Complex Systems **1** (1987) 273–347

Efficient Algorithms with Neural Network Behavior

Stephen M. Omohundro
Department of Computer Science and
Center for Complex Systems Research,
University of Illinois at Urbana-Champaign,
508 South Sixth Street, Champaign, IL 61820, USA.

Abstract. Neural network models are currently being considered for a wide variety of important computational tasks, particularly those involving imprecise inputs. This paper suggests alternative algorithms for many of these tasks which appear to have much better average performance than standard neural network models. For example, these algorithms could provide a billionfold speed increase in an implementation of an associative memory with roughly human capacity. They are based on hierarchical data structures from computational geometry and use a more direct representation of information than neural networks. As in many neural network models, the modules proposed here "learn" and can adapt themselves to different statistical distribution of inputs. They can be applied to problems involving classification, clustering, dimension reduction and the learning of nonlinear mappings. They can be implemented efficiently on both serial and parallel computers, and can potentially be used in practical applications ranging from speech and optical character recognition, to robot manipulator control, data prediction, and document retreival.

1. Introduction: Why not make computers more like brains?

One of the greatest scientific challenges facing mankind today is to understand the mechanisms of intelligence. Success in meeting this challenge will lead not only to a greater understanding of biological systems but to technological advances with a dramatic impact on the quality of human life. The current effort to understand intelligence is integrating ideas and techniques from neurophysiology, psychology and computer science. Despite a large effort to develop artificial intelligence over the last twenty-five years, the only unquestionably intelligent existing systems are the nervous systems of certain biological organisms. While there have been some successes, typical engineered systems behave remarkably rigidly when compared with biological ones. Their ability to recognize objects or speech, to manipulate the physical world and maneuver in natural terrain, to adapt gracefully to new situations or environments, and to learn from experience is still far behind the capabilities of even the simplest organisms.

Several authors have argued that this low level of achievement is due to fundamental differences between the design of biological nervous systems and present day computers [69,59,5]. Brains appear to compute with a large number of simple parallel units. Information appears to be stored in a distributed fashion and many computations appear to proceed robustly when units fail or die or even when sections of brain tissue are removed. The computations performed in the brain appear to be less rigidly constrained than those in typical computer programs and have been variously termed "heuristic", "associative", or "fuzzy". It has been argued that this fuzziness might be due to the analog or stochastic nature of neurons. Memory in brains does not appear to be separate from the computation elements. Finally, brains are constructed so that they appear to learn naturally from experience.

Because of these apparent differences, a new discipline is emerging which attempts to understand brain-like computational systems. This discipline lies on the common boundary of established disciplines and has been variously termed: "computational neuroscience", "connectionism", "parallel distributed processing", and "neural network modelling". Recent years have seen a number of important advances in this area. Current research is aimed both at achieving greater biological realism in the hopes of understanding real neural systems and at developing engineering criteria for using these techniques in the design of useful systems.

Our goal in this paper is to investigate certain engineering aspects of these new ideas. Several companies and research laboratories are investing heavily in projects based in some way on neural networks. There are currently a variety of neural network software products being introduced [88,89,90,109,125], there are attempts to develop integrated circuit versions of neural networks [61], parallel computers based on network ideas are being built [49], and one company is even selling a "neuro-computer" [51]. Given this interest, it behooves us to understand the tradeoffs between using network approaches and more traditional approaches.

An obvious question is whether the apparent differences described above allow neural networks to perform different computations than traditional computers. The widespread belief in the Church-Turing thesis (which states that physical computational devices can be no more powerful Turing machines, see p. 108 of [83]) suggests that the answer is no. Analog systems can be simulated to arbitrary precision on digital computers. Recent developments in the understanding of randomness in deterministic systems [139] suggest that by using appropriate pseudo-random number generators, stochastic systems may also be arbitrarily well modelled on deterministic digital computers. The technique of time sharing allows a serial computer to emulate an arbitrary parallel computer. The issues of performance degradation under component failure are legitimately different in networks and digital machines, but the tradeoffs and rates of failure are quite different for biological and silicon components. Much current active research is aimed at investigating the design of fault tolerant computers [141].

Much of the recent interest in computational neuroscience focuses on a class of computational models which simulate certain aspects of neural networks. These models demonstrate interesting capabilities which suggest that they might form useful components in intelligent systems. In this paper we will examine algorithms for obtaining these behaviors efficiently using digital computers. It appears that certain aspects of the network models are needed only to make up for the poor engineering qualities of neural components, which are slow, of fixed connectivity, unreliable in function and prone to damage when compared with the digital components used in modern computers. Many of the desirable computational abilities of networks can be implemented much more efficiently by using good algorithms than by direct simulation.

2. Adaptive modules in brains and software

Current neural network models do not attempt to simulate entire brains. They are perhaps best thought of as representing high-level modules from which a brain might be constructed. In this section we classify the useful functions performed by current networks into four categories. Later sections will present algorithms and data structures for efficiently implementing these functions on ordinary computers. Complete intelligent systems could potentially be constructed by interconnecting high-level modules into a much coarser network than a neural net.

Modules in brains. To get a sense of the required size, number and interconnectivity of these modules for intelligent behavior, we begin by examining biological systems. Modern neurophysiology is making rapid progress in determining the anatomical and functional architecture of the brain and nervous system [62]. While there is much left to understand, one of the most fundamental features of the emerging picture is that the brain is made up of a number of functionally different areas joined together by ordered bundles of interconnecting fibers. *A priori*, one might have feared that the brain would be hopelessly complicated and every neuron would be randomly connected to other neurons throughout the system. In fact, the cerebral cortex is quite structured and much progress has been made in mapping its interconnection structure. In 1909 Brodmann decomposed the cortex into 52 different areas based on subtle anatomical differences in the size and density of cells, the layer structure, and the density of axons innervating each region [20]. More recent work has shown that the partition defined by Brodmann's areas corresponds well to partitions defined both by distinct functional behavior and by the innervation of individual bundles of interconnecting fibers. These more recent studies indicate a slightly finer decomposition than Brodmann's but the total number of areas is estimated to be at most about 200 [26].

Much progress is being made on mapping out the interconnection pattern of these areas. Aspects of this pattern give us a rough sense of what might be needed in designed systems. For example, studies of the visual system of the macaque monkey have identified twelve areas split into two major channels, one specialized for motion perception and one for form perception [30]. Each area roughly preserves the spatial layout of the retina, the interconnection graph of the areas has a hierarchical structure in which the modules fit naturally into six successive layers, and most areas have inputs and outputs to only one or two others. V2, the area with the largest number of outputs innervates five other areas and MT, the area with the largest number of inputs is innervated by four other areas. In [81] the New World monkey is described as having ten visual areas and the macaque as having seven somatosensory areas, and six auditory areas. These areas all tend to be topographically structured. The more complex systems appear to have evolved by introducing more cortical areas. An early animal like the hedgehog exhibits quite complex behavior but apparently has only two visual and two somatosensory areas.

The type of computation that can be performed in a brain is severely constrained by the limitations of neurons. The total time to perform an interesting computation such as recognizing an object is about 0.5 seconds, while an individual neuron is only capable of firing in time intervals of about 0.005 seconds. Therefore only about 100 layers of neurons from sensory input to motor output can be involved in such a computation [35]. This limit, together with physiological features described above suggests that intelligent computations can be performed in networks consisting of less than 200 modules, each of which performs a function that can be accomplished within a few layers of neurons. There are probably at most 20 to 40 modules along any path from input to output. Each module, while quite restricted in depth by the speed of neurons, may be quite wide and can perform much of its computation in parallel. There are estimated to be about 10^{11} neurons in the human brain, though many of these likely perform redundant tasks to make up for the poor behavior of individual neurons. We can hope to use these constraints on the structure of brains to estimate the computation required to perform the desired tasks.

Software engineering. One of the engineering hopes for neural network models is the development of a new kind of "extensional" computer programming. The idea is to build a software system only partially by precise specification, leaving most of the details to training by example. Unfortunately, most current networks require so much computation that they do not present a viable alternative to traditional programming. The efficient module implementations described in this paper suggest a potentially practical implementation of this idea. The high level module functions would constitute the primitives of a network programming language. Building an adaptive system for a specific task would consist of specifying the types and interconnectivity of these high level modules. A simple compiler or interpreter could construct the appropriate data structures and system code to implement the network. The modules would then be trained, either individually or as a system, on the specific inputs of interest. The resulting system could be efficient, quick to build and modify, and robust to changing environments. Such a system could be the answer to the challenge in [138]'s to go beyond current programming ideas. Similar approaches to general engineering questions are discussed in [140].

2.1 The structure of a module

At the coarsest level, a neural network module may be viewed as a black box with an input channel and an output channel. The output produced by the module on each input defines the function of the module. Let us denote by I a space whose points represent all possible inputs and by O a space whose points represent all possible outputs. These spaces might be finite or infinite collections of points and will sometimes be given extra structure such as a topology or a coordinatization by real numbers. The representation of these possible inputs and outputs in terms of neuron activity or software data structures will be an important issue for understanding how the module may be built out of real neurons in three-dimensions or implemented on a computer. We will discuss representations throughout the paper, but for now it is preferable consider these spaces abstractly and to only impose as much structure on them as is *functionally* necessary.

There are two aspects of the input and output spaces that will be critical in all that follows. The first is that in most real situations not all inputs in I are equally likely. We will assume that I is a measurable space and use $\rho(x)$ to denote the probability of the input $x \in I$ in the discrete case or a probability density defined with respect to coordinates on I in the continuous case. The second aspect is that points in these spaces will often represent the specification of the value of several quantities simultaneously. In this case the spaces have a natural product structure and are best thought of as multi-dimensional. We will introduce four classes of module which differ according to how the functionality is specified and to whether the output is continuous or discrete.

2.2 Directed and undirected learning

The desired mapping from I to O that a module implements may be specified in two basic ways. A trainer either specifies the mapping by direct specification or by presenting examples, or it is determined in some way by the set of inputs it receives. The first class corresponds to directed learning, the second to undirected learning.

Directed learning. The simplest form of directed learning hardly deserves to be called learning at all. It uses a direct specification of the mapping to describe a module's structure. This specification may take the form of a table of output values for each input, or may be an algorithm for converting an input to an output. This method of specification is analogous to the genetically specified portions of nervous systems. This probably includes virtually the entire nervous system of lower animals and includes at least the reflex loops in higher animals. This is the method by which computer programs are specified today.

Directed learning may also be based on presentation of examples. A module obtains the greatest amount information when it is allowed to ask for the output corresponding to an input which it specifies. More usually, it is presented with a given sequence of inputs taken from I according to the probability distribution ρ. In this case, the greatest information results if a teacher specifies the desired output in O corresponding to each of these inputs. Alternatively, the teacher may only indicate how far and possibly in what direction the current output differs from the desired output, or even merely whether the currently produced output is correct or incorrect. Many of the interesting recent learning algorithms for neural networks, such as back propagation [113], are based on trained learning in which a teacher presents the desired response for a variety of inputs.

Undirected learning. Undirected learning produces a mapping whose structure is determined by the set of inputs that a module receives. If the inputs are randomly distributed according to the probability distribution ρ and the exact order of the inputs is not relevant, then the desired structure of the mapping should depend only on ρ. Such a module looks for regularities and attempts to enhance them. Models of "self-organization" in network systems are often of this type.

2.3 The four basic types of module behavior

Continuous vs. discrete. Whether the input space I and the output space O are discrete or continuous often has a big effect on the function of a module and on the algorithms which implement it. Digital systems are of course only able to represent a discrete set of values but there is often still a notion of "neighboring" values. A strictly discrete space consists of independent points which have no relationship to each other. An example of such a space might consist of a set of exclusive categories without a notion of distance between them. The points of a strictly continuous space have neighborhood relationships which are locally well modelled by Euclidean space. A typical example might be the space whose points consists of configurations of a robot arm. Nearby points correspond to nearby configurations. Such a space must be modelled on a machine using only a finite number of points, but it is endowed with a metric which specifies the distance between points. Most situations are intermediate between these two extremes.

For completely specified mappings in which the exact input to output correspondence is given, any relationship between points is functionally irrelevant. Even in this case, introducing an appropriate relationship will sometimes enable us to compress the description the mapping. More generally, a neighborhood relationship on the input space I allows a module to *generalize* beyond specified examples. We will introduce modules which categorize an arbitrary input with the nearest input that it has previously seen. Inputs which are nearby are expected to have nearby causes and should elicit nearby responses in the absence of information to the contrary.

Our systems need to be insensitive to small errors in their inputs and so all of our modules will be assumed to have some kind of neighborhood relations defined on their input spaces. On the other hand, a module's behavior is quite different depending on whether it produces a fixed set of outputs or interpolates between stored outputs. We will divide the behavior of the modules we wish to study into four categories according to whether their outputs are continuous or discrete and whether their function is specified by directed or undirected learning:

1. Directed learning modules with discrete output implement the functions of associative memory and classification.
2. Directed learning modules with continuous output implement the evaluation of smooth nonlinear mappings.
3. Undirected learning modules with discrete output implement the tasks of category formation and clustering of inputs.
4. Undirected learning modules with continuous output implement dimension reduction, self-organization, and probability equalization.

These four types of behavior are representative of the useful behaviors observed in neural network models, as we shall discuss in the next few sections.

2.4 Example neural network behaviors

A wide variety of network models have been proposed and implemented. Reference [114] gives a common framework for describing many of these models. This reference identifies eight aspects of a neural network model: a set of processing units, a state of activation, an output function for each unit, a pattern of connectivity among units, a propagation rule for propagating activities throughout the network, an activation rule for combining inputs acting on a given unit to produce its new level of activation, a learning rule for changing the propagation rule with experience, and an environment within which the system operates. In this section we will examine a sampling of the proposed models in terms of the four behavioral categories introduced above.

Adeline. One of the first neuron-like models was described in [136] and falls into the directed continuous class. The output of an "Adaptive Linear Element" or Adeline, was a weighted sum of its inputs. A set of input/output pairs were repeatedly presented and the weights were incrementally modified to push the actual output closer to the desired output. A network of such units implemented a linear mapping between the input space and the output space. When the specified inputs were linearly independent, it was proven that the weights converged to values that produced the desired outputs. These units were applied to the design of communication systems such as adaptive antennas. The function performed is to learn a linear mapping from input to output given a set of input/output pairs.

Perceptron. The perceptron described in [112] is perhaps the most famous of the early models. It falls into the directed discrete class. It consisted of a sensory layer of units whose outputs were combined in a fixed way by units in an association layer and then linearly combined with variable weights and fed to a layer of response elements which used a threshold to give a binary response. The perceptron had only a finite number of outputs and was trained to categorize each input into one of these categories. When the desired input categories were separable by hyperplanes in the input space, it was proven that the weights converged to correct values. The function performed is to learn to classify inputs into categories.

Learning matrix. An early system which exhibited undirected learning was the "learning matrix" [124,123]. This system was meant to simulate classical conditioning in animals. It had two sets of input lines which were interconnected as a complete crossbar, with an adaptive element joining each pair of lines. Initially the system was restricted to binary inputs and outputs and would be in the undirected discrete category. Correlations between the two signals joined by an adaptive element caused it to decrease its resistance. If only one of the two inputs was presented, the system would produce at its output the historically most highly correlated other input. By allowing the inputs and outputs to be continuous, we get a system in the undirected continuous category which was studied in [64]. Many of the more recent proposals for associative memory including the so-called "holographic" memories are quite similar to this [137].

Competitive learning. [116] discusses a class of models which are of the undirected discrete type. The basic idea of "competitive learning" is that a set of M units each listen to the input. The unit which responds most strongly to a given input inhibits all the others. Only this unit has its weights changed to make it respond even more strongly the input. The resultant network tends to distribute the response of the M units evenly over the high-probability portion of the input space. The competitive aspect keeps the units from clumping at the highest probability input. If there is a high probability region that is not well represented by a unit, then every time the network receives an input in that region, the nearest unit will be pulled even nearer. If the input points are highly structured, then units' responses tend to correspond to clusters in the set of input samples. The function performed is to detect regularities in the input distribution and to categorize further inputs according to them.

Self-organizing feature maps. A similar type of network can implement interesting undirected *continuous* behavior. In chapter five of [65] and the references therein, and more recently in [110], a model is studied which learns to adapt its input/output mapping to the statistics of the input distribution while preserving topography. The so-called self-organizing feature maps consist of a collection of units each of which is assigned a set of neighboring units. An appropriate choice of neighbors imposes a topology on the set of units. A common choice gives the units the neighborhood relations of a two-dimensional grid. All units receive the same inputs and, as in competitive learning, the unit which responds most strongly to a given input has its weights changed in a direction to even further increase its response. The new ingredient in these models is that the strongest element's neighbors also have their weights modified. As in competitive learning, the units want to spread themselves out over the high probability regions of the input space, but in these systems they also want to respond well to inputs near those that their neighbors respond well to.

Simulations described in the references illustrate several useful behaviors. A convenient way to picture the network's response is to map each unit to the input point to which it responds most strongly. When the units are structured as a two-dimensional grid and the inputs are drawn uniformly from a two dimensional region such as a triangle, the image of the grid under this mapping is distorted so as to uniformly represent the region. When the input probability varies over the input space, the density of grid points becomes proportional to the probability density. When the output space has a lower dimension than the high probability region of the input space, it maps into it in a convoluted way reminiscent of a Peano curve. When the extra dimensions are small in extent, the output grid forms periodic stripes which oscillate in the extra dimensions. In an experiment in which the inputs were clustered into thirty-two clumps in a five dimensional space and the output grid was two-dimensional, the clumps were represented in the two dimensional space in a way that preserved many of their neighborhood relations [65]. Clumps joined by links in the five-dimensional Euclidean minimal spanning tree (discussed in section 6) were represented by neighboring units in the output grid. The functions performed are undirected learning of nonlinear mappings that perform probability equalization when the input and output spaces are the same dimension, and dimension reduction while preserving important neighborhood relations when the output dimension is smaller than the input dimension.

Control problems. In [8] a computer system is described which learns to balance a pole on its end. Much as in the modules described in section 5, this system cuts the input space into distinct regions and learns the appropriate control response in each region. The decomposition in this system is fixed, however, and not determined during the learning procedure. The behavior of these modules is that they learn to evaluate a nonlinear mapping from an input describing the current state of the pole to an output which describes the appropriate response. Reference [8] also makes an argument for building network systems out of modules which are more complex than typical neuron units.

Neocognitron. In [44] a neural network model is used to do pattern recognition. An earlier version was undirected, but this reference describes a system which is directed and discrete. There are 14,529 units organized into 9 layers. Each unit has a few hundred synapses, so the whole system has a few million. Individual layers were given hand-tuned training patterns to learn and the system appears to do a credible job of distinguishing handwritten versions of the ten arabic numerals. The behavior of this system is that it can be trained to categorize patterns.

Optimization problems. References [60,59] describe a class of neural networks with symmetric weights (i.e. the strength of the connection from i to j equals that from j to i). They show that the state space of the resulting dynamical system has a Lyapunov function (i.e. one whose value is nonincreasing in time) which they call an energy function. All orbits in such systems must approach limit points and there cannot be any periodic orbits. A computation is performed by presenting the inputs and waiting for the system to settle down to a fixed point. To solve optimization problems such as the travelling salesman problem, the authors construct the energy function to have a global minimum at the optimal legal solution. The system is started at a random state and it is hoped that it settles down to a fixed point which is near the global minimum. To encode the travelling salesman problem with n cities, n^2 totally interconnected neurons are used.

Back propagation. In [85], perceptrons were shown to be quite limited in the functions that they could compute. These limitations stem from the fact that perceptrons have only a single layer of modifiable weights. Much of the recent resurgence of interest in networks has come from the discovery of "back propagation," a weight modification rule that applies to multi-layer networks [113]. The setting is the same as for perceptrons in that input/output pairs are presented and the network is supposed to learn to produce the desired output when presented with a given input. When the system produces the wrong output, the learning rule simply changes each weight in the direction which makes the size of the error decrease as quickly as possible. The components of this steepest descent direction in weight space are computed by using the chain rule to compute the partial derivatives of the error function with respect to each weight. The implementation of this weight change requires propagating an error signal backward through the network, changing weights that had a large effect on the output more than those that did not.

Reference [113] describes a two unit system with five weights which was trained to learn the two-input "exclusive or" function. It typically took about 558 presentations of each of the four possible inputs but occasionally got stuck in an incorrect local error minima after many (in one instance 6,587) presentations of each input. Another two input system had four outputs and was supposed to have an output on the line indicated in binary on the input. A nine unit network with twenty-two weights took 5,226 presentations of each input pattern to learn the correct outputs. Networks which learn using back-propagation are of the trained discrete category and seem to be useful because they appear to generalize in a natural way on real world examples.

NETtalk. An important example of a back propagation system is described in [120]. This system learns to map strings of seven successive characters in English text into the phoneme that should be pronounced at the middle letter. The system has 309 units and 18,629 weights. It reached ninety per cent accuracy on a sample of 1,024 words after about 30,000 word presentations. A network with 10,000 weights took 0.5 seconds to process each letter on a VAX 11/780 with a floating point accelerator.

2.5 The amount of computation required by neural networks

In this section we discuss the amount of computation required to implement simulations of neural networks on current serial computers. Section 9 compares these performance estimates with the performance of the algorithmic modules described in later sections, in both serial and parallel implementations.

Geometric analysis. To understand how the computation performed by a neural network might be done more efficiently, we will begin with a geometric analysis of Boolean networks. Consider networks made up of units which form a linear combination of their inputs and produce a one or a zero depending on whether the sum exceeds a threshold. A full network is a collection of these units with k real-valued inputs and some number of boolean outputs. Let us assume that the network has a layer of input units which each receive some subset of the k input values and that to these are connected a network of other units forming an acyclic graph (i.e. the network is a purely feedforward system).

Let us focus on a single output. The set of weights and the structure of the network leading up to this output unit define a particular computation which assigns a one to some inputs and a zero to others. With the inputs taken to be real numbers in a specified range, it is natural to form a geometric picture of this computation. The k input values together define a point in the k-dimensional input space I. The output space O has only the two points one and zero. The network may be thought of as defining the subset S of I which consists of those inputs which cause the unit under consideration to produce an output of one. The function performed by the network is to determine whether a given input is in S or not. The subset S changes as we alter the values of the weights in the network.

To understand the structure of the subset S, consider the computation performed by a single unit. Letting W_i represent the weight on the ith input and T represent the threshold, the set S consists of all points whose coordinates x_i in the input space satisfy:

$$\sum_i W_i x_i > T. \tag{2.1}$$

This inequality is satisfied by a region of input space which is bounded by the hyperplane:

$$\sum_i W_i x_i = T. \tag{2.2}$$

(A hyperplane in an N-dimensional vector space is a linear subset of dimension $N-1$. Such a hyperplane divides the embedding space into two distinct regions.) As we vary the weights W_i and the threshold T, this hyperplane moves about the input space.

Each of the units in the input layer of the network similarly defines such a hyperplane in the input space. The entire network *beyond* the layer of input units can only perform a Boolean computation on the outputs of the input layer. It receives boolean values from the input layer and it produces a single output which is either zero or one. While this Boolean function may be quite intricate, all of the interesting *geometric* structure is defined by the first layer of units. The collection of possible sets S consists of all combinations of unions and intersections of the half spaces defined by the input units. We may imagine the input space as being cleaved by the input unit hyperplanes. After all cuts have been made, the resulting polyhedral pieces are the components from which we make the set S. The different Boolean functions that may be performed by later units correspond to including and excluding different pieces in the set S. The network has the representational capacity to distinguish inputs only up to the granularity of the pieces defined by the hyperplane decomposition.

An important class of computations classifies inputs into one of several categories. The different output units could represent membership in the different categories. In a learning session, the hyperplanes would be moved around so as to better represent the desired categories. We can now begin to understand the inefficiency in performing this computation using a direct network simulation. The input is processed by every input neuron in every computation, which geometrically amounts to comparing the input point with each hyperplane to determine on which side it lies. Once each comparison is made, the collection of answers is used to determine membership in the set S.

Usually, many of these comparisons will be superfluous. For example, consider the case where the input space is two-dimensional and there are two vertical hyperplanes (which are just lines in two dimensions). Knowing that an input point lies to the left of the lefthand hyperplane immediately implies that it is to the left of the righthand hyperplane and there is no need to do a second comparison. In computer science, such situations as this often yield to the technique of recursive decomposition. Every time a piece of information is determined about the input, it is used to prune away unnecessary further computations. The total amount of computation required for a given answer can be far less than that required by a brute force approach which performs the same computation on every input presentation.

Capacity of Hopfield nets. Reference [58] presents a neural network which stores associative memories by arranging the dynamics so that memories correspond to attractors. With N totally interconnected neurons, it was found that $0.15N$ memories could be stored before "error in recall is severe." This storage capacity has been confirmed in studies which examined the asymptotic capacity as both N and the size of the network approach infinity [4].

Capacity of learning matrices. Reference [77] analyses the number of units needed in single layer learning matrices. If r associations are to be stored, each with an average of m_i of its input bits equal to one and m_o of its output bits equal to one, then the number of units must be greater than or equal to $1.45rm_im_o$. There must be at least $1.45m_im_o$ times as many units as there are stored memories. Any direct serial implementation of this scheme will require an amount of storage and take a time to retrieve a single memory which are both a substantial multiple of the number of memories stored.

Computation time. To simulate a totally connected neural network with n neurons, we must do n^2 multiplications and additions to form the linear combination of the inputs. These n sums must then be passed through the nonlinearity. The analyses in [77,58,4] and elsewhere suggest that at least $O(n)$ neurons are needed to store $O(n)$ memories, yielding $O(n^2)$ computation time and storage space. In later sections, we present a variety of algorithms which use $O(n)$ space, have a search time of $O(\log_2 n)$, and yet implement neural-like behaviors from each of the four categories defined above.

Learning time. Reference [128] studies how the required number of presentations scales with the total number of memories in networks which learn by back-propagation. The study examines learning of a particular function and so should provide a lower bound on the time to learn an arbitrary set of memories. As discussed above, a network has a maximal memory capacity beyond which the learning time is infinite. Before that limit is reached, however, the number of presentations required for learning appears to scale as the 4/3 power of the number of memories to be stored over a wide range. It would be of great interest to study this kind of scaling relationship for other types of networks and learning algorithms.

Million memory goal. It is of interest to estimate the memory capacity needed to achieve human-like performance on tasks of interest. To get a sense for the absolute upper bounds, we may consider the total amount of information input to the nervous system during a lifetime. In thirty years of life, there are roughly a billion seconds. There are roughly 10^7 afferent fibers sending input to the brain. Each of these has a data bandwidth of roughly 100 bits per second. The entire sensory input over the course of a lifetime is therefore about 10^{18} bits. If a person stores a memory every 10 seconds, then during waking hours there is only time for roughly fifty million memories by adulthood. The total amount that a person could write, if they wrote full time during their whole life is about a gigabyte (10^9). The most that they could read is about a terabyte (10^{12}) and the most that they could see in high resolution color video is about a petabyte (10^{15}).

A number of indicators suggest that the actual number of entries that must be stored for human-like performance on various tasks is of the order of a million. There are about 350,000 words in large English dictionaries, but typical individuals know only about 50,000 of them and only about 10,000 are used in everyday speech and writing. The probability distribution of these words is very closely approximated by Zipf's law which for the first 8,000 English words says that the nth most common word has probability approximately equal to $0.1/n$. The number of objects a person can identify with a name or short phrase is therefore probably less than a million. A good player in the game of "twenty-questions" can usually identify an arbitrary object and an optimal set of questions can only distinguish $2^{20} \approx 10^6$ objects. It also appears reasonable to estimate that a person learns less than an average of 100 new things per day during the 10,000 days into adulthood. Many of the parallel nerve bundles which communicate information from one area of the brain to another, such as the optic nerve, have on the order of a million fibers. These estimates suggest that we should be considering modules with a storage capacity of approximately one million items. Section 9 makes performance comparisons based on this million memory goal.

3. Tools for module construction and analysis

We will use a variety of powerful algorithmic tools to build efficient software versions of the modules described in the last section. Much of the theoretical work in computer science has focused on obtaining good *worst case* bounds in the asymptotic limit as the size of problems gets arbitrarily large. Unfortunately, somewhat arcane data structures with large numbers of intricate components are often required to deal with worst case situations. These situations are often extremely rare in practice and the intricacy of the data structures makes them difficult to implement and inefficient for realistic problem sizes. We will be concerned with simple data structures that have good *average* case performance with respect to the probability distribution ρ on the input space.

The first part of this section presents data structures for representing one-dimensional data. The second part extends these to the multidimensional setting. The third part presents some statistical ideas that are useful in analyzing the average behavior of these structures and in adapting structures to the underlying probability distribution. Finally, the last part discusses algorithms for building and maintaining these structures.

The data structures we are concerned with will be used to store labelled data in a way that allows it to be efficiently accessed by using various properties of its label. In database literature the label is called a key. We will denote the space of possible labels by I as above, and will also refer to it as the embedding space or the key space. We will denote the size of this space by $|I|$ and will use N to denote the number of entries which are to be stored. We will sometimes assume that the keys are selected according to the probability distribution ρ defined on I. Throughout the study of data structures, there is a basic tradeoff between building a structure by using properties of the embedding space I or by using properties of the actual data to be stored.

A key aspect of modern digital computers is the memory system which consists of a large number of storage locations that may be accessed by a numerical label called the address. An address decoder in the memory converts the numerical address into a signal at the corresponding storage location and the content of the memory at that location is placed onto a data bus where it is accessible to the rest of the system. This operation is a very powerful one and is not directly available in a neural network, though several researchers are working to build networks with this functionality.

In modern software systems, a single conceptual entity is often represented by several adjacent memory locations. These locations hold the different components of the entity and the entity as a whole may be assigned the address of its first component. Components may contain the numerical address of other entities and in this way *data structures* composed of many entities may be constructed. Such addresses are called *pointers*. A common way to allow an arbitrary number of entities to be accessible from a single address is to form them into a *linked list*. One component of each entity is a pointer to the next entity in the list and the pointer in the last entry in the list contains a special value called "null" (which is often taken to be 0).

The entities composing all of the data structures we use here will represent subsets of the input space I. Different data structures will decompose I into subsets in different ways and will link these subsets together into different structures. The general goal will be to build structures that adapt themselves to the underlying probability distribution so as to support fast access with small structures.

3.1 Arrays, hashing, tries, and trees

Arrays. The high level abstraction of the basic capability of computer memory to access addressed elements is the *array* data structure. Each integer in a fixed range is associated with a piece of data. An $|I|$ entry array A requires $|I|$ storage locations and allows access to an arbitrary element in fixed time. The idea of the array data structure appears in the very earliest works on computers [45].

Let us begin by examining the problem of storing data elements whose keys are drawn from a one-dimensional space I. We will assume that I consists of the integer keys in the range $[0, |I|)$. If the number of data items N is equal to the size of the key space $|I|$, and the data is labelled by the N integers from 0 to $N-1$, then one could not hope for a better structure than the array. To read or alter the data associated with the key j, we directly access array location j denoted by $A[j]$ in constant time.

The need for more sophisticated structures arises when the number of actual data elements to be stored is much smaller than the number of possible elements $|I|$. In this case, using one memory location for each possible key is wasteful of space. The basic idea of the more sophisticated data structures is to let different locations in the structure represent subsets

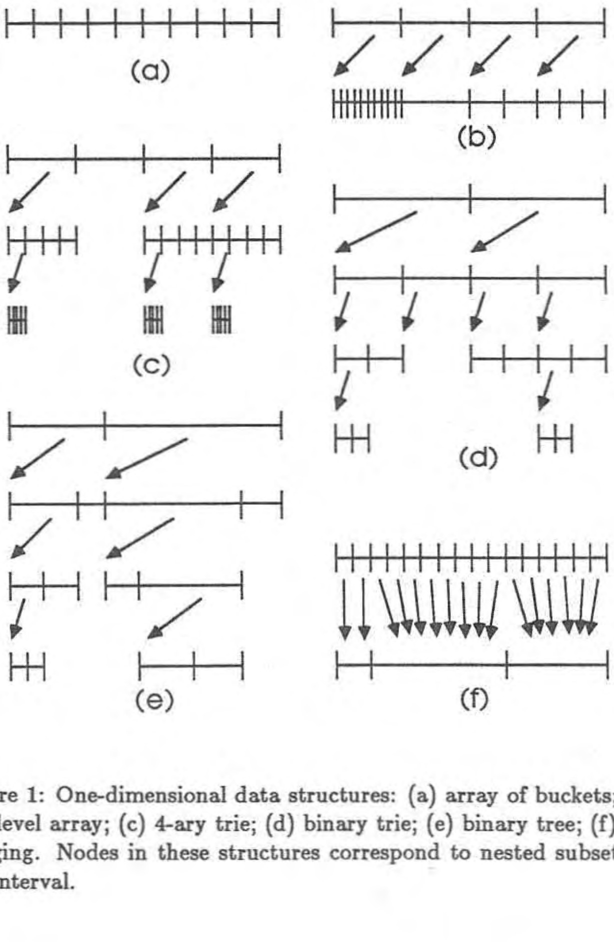

Figure 1: One-dimensional data structures: (a) array of buckets; (b) two-level array; (c) 4-ary trie; (d) binary trie; (e) binary tree; (f) bin merging. Nodes in these structures correspond to nested subsets of the interval.

of the key space I rather than individual entries. Such locations will often also store pointers to locations representing smaller subsets. The search procedure for a point traverses these pointers to access locations in the data structure representing smaller and smaller subsets of I whose entries satisfy a desired key property. When the reached subset of I is small enough, the few data entries in that subset may be directly examined.

Arrays of buckets. As a first data structure of this type, we consider arrays whose entries correspond to subintervals of I rather than points (Figure 1a). For example, instead of letting j run from 0 to $|I|-1$ and using the array entry $A[k]$ to represent the single key j, we let the array index run from 0 to $M-1$, and let the jth entry represent the subset of keys in the range:

$$\left[\left\lceil \frac{j|I|}{M} \right\rceil, \left\lceil \frac{(j+1)|I|}{M} \right\rceil\right), \tag{3.1}$$

($[,)$ means the interval including its left endpoint, but excluding its right, and $\lceil x \rceil$ means smallest integer larger than or equal to x.) Array locations must now sometimes hold multiple elements, and so we will refer to them as *buckets* or *bins*. The multiple elements in a bucket are typically stored as a linked list.

If the number of elements stored in each array location is small, we get almost the same efficiency of access as in a complete array but potentially with many fewer array locations. If we choose $M = N$, there are as many buckets as data elements. The ideal situation would then occur if the data elements were evenly spaced in I. There would then be one element per bucket, giving fast access, and yet only as many array locations as data elements. In the worst situation, all elements would end up in a single bucket. If the elements are drawn from a uniform distribution, then the behavior approximates the ideal situation on average. The number of elements in a bucket is described by the binomial probability distribution

$$P(j) = \frac{N!}{j!(N-j)!} p^j (1-p)^{N-j}, \tag{3.2}$$

where $p = 1/M$ is the probability of a single entry falling into a given bucket. On average, a bucket will contain N/M elements, which is 1 if $M = N$.

More sophisticated structures may be thought of as attempts to achieve this ideal average behavior even when the input distribution is non-uniform. The problem with cutting I uniformly into bins is that bins in regions where ρ is large will have too many data elements, while buckets in regions where ρ is small will have too few.

Hashing. A classical method of handling this problem is *hashing* in which a key is first sent through a randomizing hashing function (such as $j \mapsto \alpha j \bmod M$) before being assigned to a bucket. If the hashing mapping is sufficiently mixing, it will take an arbitrary smooth probability distribution into an approximately uniform one which has good binning behavior. Unfortunately, hashing works by destroying the neighborhood relations of the space which will be of great importance to us. The subsets of I corresponding to individual bins are extremely fragmented sets with pieces taken uniformly from the whole range of I.

Two-level arrays. An approach which preserves the neighborhood relations of the data partitions the space so that the bins are smaller in regions where ρ is large and larger where ρ is small. Most standard probability distributions vary rather slowly and have only a few peaks in the space I. In this case, a two level binning process can be used to fit the data well. As above, we split the space I evenly into M buckets, though now M need only be large enough to capture the scale of large variation of ρ (say ten buckets). We further uniformly divide each bucket b into M_b intervals instead of putting the data elements directly into them. M_b is chosen so that buckets with a larger number of elements are subdivided into more sub-buckets, while buckets with only a few elements may not be subdivided at all (figure 1b). The first level buckets store the value M_b and a pointer to the array of its sub-buckets. To access an element, we use the quotient of its key and $|I|/M$ to determine which first level bucket it lies in. The remainder, along with the stored value of M_b for that bucket, determines the location of the second level bucket. The access time (two steps) is almost as fast as for an array with a uniform distribution and for smooth enough ρ we can keep the bucket occupancy at 1 on average while using only about as many buckets as there are data elements.

Tries. To adapt the bucket structure to the input distribution even more finely, we can consider structures with more than two levels. An m-way *trie* is a structure which decomposes each bucket into m pieces or not at all depending on how many points there are in the bucket (figure 1c). Densely populated regions may have several levels of splitting, whereas sparse regions may not be split beyond the first level. A bucket that is further decomposed contains a pointer to the array of buckets at the next level. To search for a point, the quotient of its key and $|I|/m$ is used to determine which bucket it lies in and the remainder is used for further computations if any. If the bucket is not further decomposed, the desired point will be stored in it directly (typically in a linked list). Otherwise the remainder is used to determine the bucket at the next level to look in, and so on.

A particularly important structure is the 2-way or *binary* trie [39] (figure 1d). Each node is split into either two halves (accessed through left and right pointers) or not at all. At the ith level, the choice whether to go left or right is determined by the ith bit of the key expressed in binary. In a trie all buckets at the ith level have the same size and the adaptation to the distribution ρ occurs only in the choice as to whether a bucket is further decomposed or not. One property of this structure is that the path through the structure is determined only by the address of a key and not by which other elements are stored. A variant called the Patricia structure replaces paths from the root which do not branch by single pointers and is very effective for storing long strings [86].

Trees. If we further allow a region to be decomposed into different sized regions, we get a data structure known as a *tree* (figure 1e). Particularly important is the binary tree, in which each region is decomposed into two pieces or not at all. To specify the place at which a region is cut, we store a *discrimination value* with each node. The top level of the tree is called the root and represents the entire space I. To search for a piece of data, we begin by comparing its key with the root's discrimination value to determine whether to proceed to its left or right child. For a given number of nodes, a tree can adapt itself more finely to an input distribution than can the structures which use address computation. Regions are often decomposed until there is only one entry in each leaf bucket. If at each stage we cut a bucket so that half its entries are assigned to the left child and half to the right, then the depth of the tree will be $\lceil \log_2 N \rceil$. Such a tree is said to be *perfectly balanced.* Research on data structures has discovered many techniques for dynamically keeping a binary tree approximately balanced under insertion and deletion. A review of these structures may be found in [80]. With randomly chosen elements, balance is maintained automatically on average.

Bin merging. Trees adapt well to the structure of the data, but because retrieval requires a sequence of comparisons along the path from the root to the desired leaf, they are often slower to use than structures which require only a computations on the key. In some situations the number of buckets is fixed or we would like each bucket to contain at least some minimum number of elements. This situation arises both in parallel computing when a buckets are assigned to processors and in disk based systems when buckets are assigned to disk blocks. A useful approach for this situation which we call "bin merging", uses a finely partitioned array to decompose I for fast access, but allows many contiguous entries in the array to have pointers to the same bucket (figure 1f). The subset corresponding to a bucket may then be tuned to the resolution of the array, but the number of buckets can be as small as desired. In this way we get some of the best features of both trees and arrays. This structure is useful in the implementation of the grid file structures described in section 3.2. It is also the key to "extendible hashing" [31], a dynamic storage scheme which guarantees data retrieval in only two disk accesses and is therefore competitive with B-trees, a balanced tree scheme which is the current standard.

We have discussed two basic approaches to forming the state space decomposition given a collection of states. In the first approach, illustrated by arrays, hashing, and tries, the structure of the pieces is determined by properties of the state space and the data to be represented merely determines which pieces appear in the decomposition. In the other approach, illustrated by trees, even the shape of the pieces which make up the decomposition of state space is determined by the stored data. The first approach often allows for very fast retrieval, more regular decompositions, and simple dynamic behavior. The second approach is more adapted to the data, leading to fewer nodes and better manipulation characteristics. Traditionally, trees have been the preferred data structure, but [92] argues that they may soon be overshadowed by the other class.

3.2 Multi-dimensional data structures

In this section we describe generalizations of these basic one-dimensional structures to handle multi-dimensional data. Many of these generalizations arose in the solution of problems in the field of computational geometry. This young discipline seeks to design efficient algorithms for solving geometric problems and has seen explosive growth in the last ten years [105]. One critical aspect of this growth has been the development of data structures and algorithms capable of dealing efficiently with entities naturally described in spaces with dimensions larger than one. To date, most of this work has focused on algorithms with good worst case behavior. These tend to be based on intricate data structures which typically only work well on two-dimensional problems. Some progress has been made, however, on higher dimensional structures with good *average* case behavior when the inputs are reasonably distributed. These structures not only have excellent average behavior, but are extremely simple and efficient to implement. They will be used to implement the adaptive modules described in later sections.

Grids. The most natural multi-dimensional generalization of the array of buckets is a multi-dimensional array of buckets. If I is a k-dimensional space, we may store its points in a k-dimensional array which we shall call a *grid* (figure 2a). We uniformly partition each dimension exactly as we did in the one-dimensional case, except that different dimensions can be cut into a different number of pieces. To access a point, we apply the one-dimensional computation to each coordinate to determine the k indices of the bucket that it falls into. The buckets in this case correspond to hyper-rectangular regions in I of a uniform size and shape which are aligned with the coordinate axes. As in the one-dimensional case, this structure is ideal for uniformly distributed data. If there are N data points and N grid buckets, there will be an average of one data point per grid bucket. In this case the grid requires only as many storage locations as there are data points and entries may be accessed in constant time (or time $O(k)$ if one allows the dimension to grow asymptotically). As in the one-dimensional case, the more sophisticated structures are needed to achieve this ideal behavior for non-uniform probability distributions.

Adaptive grids. We will refer to the next class of structures as *adaptive grids* [93,50,107]. As in ordinary grids, the data points are stored in a k-dimensional array. Here, though, the sizes of the one-dimensional buckets which partition each axis are chosen to produce a finer decomposition in high-probability portions of the space (figure 2b). We may use any of the structures described in the last part to decompose each coordinate axis into buckets. The adaptive grid itself is a k-dimensional array, in

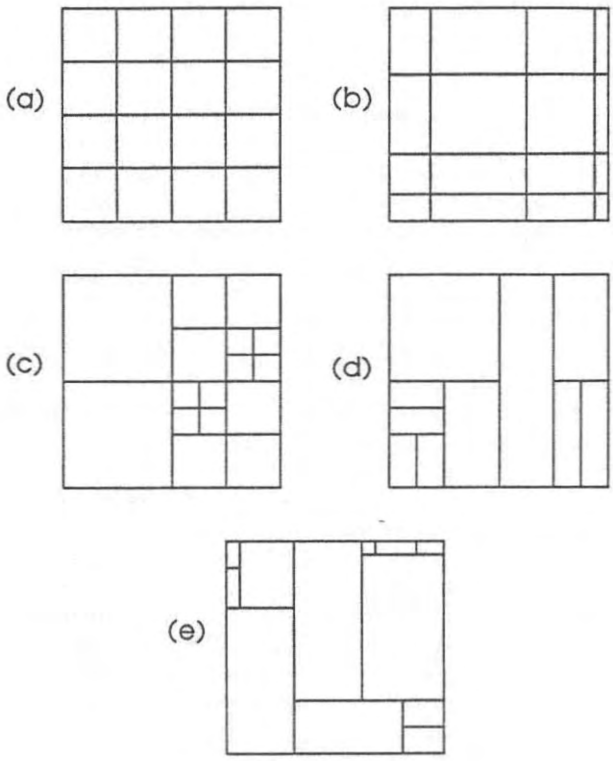

Figure 2: Multi-dimensional data structures: (a) grid; (b) adaptive grid; (c) quad trie; (d) *k-d* trie; (e) *k-d* tree. Shown is the partition of a two dimensional region by the lowest level buckets of each kind of structure.

which each dimension has an index that runs from zero to one less than the number of final buckets in the corresponding one-dimensional structure. To access a point, each of its coordinates is sent through the appropriate one-dimensional structure and is assigned the index of the final bucket reached there. These k integer indices are used to access the array location representing the bucket which contains the point. The buckets here are hyper-rectangular, but their shape and size varies over the input space I. The bin merging technique is useful here because the number of final buckets in the one-dimensional structures has a dramatic effect on the number of buckets in the resulting adaptive grid.

Adaptive grids have fast access times, are easy to implement, are fairly easy to alter dynamically, and can adapt well to distributions that vary gradually over the input space. Unfortunately, they can require many more buckets than necessary when representing distributions with a complicated local structure. The basic problem is that partitioning one of the one-dimensional buckets has a global effect on the grid, causing the partitioning of all the buckets which intersect the hyperplane orthogonal to the partitioned dimension. This effect is especially bad when I is high dimensional. Later structures will respond more locally and therefore more gracefully to features in the probability distribution.

There is a very useful technique which we call *partial summation* which is applicable to both adaptive and ordinary grids and which is particularly effective in two and three dimensions. This technique is similar to one used for range counting in computational geometry (see page 37 of reference [105]) but it appears to be much more useful in the current context. We often wish to make a multi-scale study of the distribution of a quantity f which has a value in each grid bucket. The technique allows us to obtain the total amount of f contained in an arbitrary hyper-rectangle in constant time. It begins with a preprocessing pass through the grid along each dimension in which the partial sums of f are accumulated along each dimension. For example, in two dimensions the horizontal pass forms the array whose (i, j)th entry is

$$\sum_{k=0}^{j} f(i,k) \tag{3.3}$$

and then the vertical pass forms the array P with (i, j)th entry

$$P(i,j) = \sum_{l=0}^{i} \sum_{k=0}^{j} f(l,k). \tag{3.4}$$

Regardless of how large the grid is, we may now find the sum of f taken over an arbitrary hyper-rectangular collection of buckets in constant time. In two dimensions, the sum over the rectangular region with diagonal corners (i_1, j_1) and (i_2, j_2) is

$$P(i_2,j_2) - P(i_2,j_1) + P(i_1,j_1) - P(i_1,j_2). \tag{3.5}$$

Because this takes only four additions, we may treat obtaining the total amount of a quantity in an arbitrary rectangle as a primitive operation. A similar idea is used in [95] to efficiently find scale-independent peaks in histograms. We have used the technique successfully on images to provide very fast scale-independent segmentation and analysis. In this capacity, the technique forms the basis for a fast optical character reader [97].

Multi-level grids. There are two natural generalizations of the one-dimensional trie structure. The first is a multi-level grid. The top level grid coarsely partitions I into equal sized evenly spaced hyper-rectangles. Those top-level buckets which contain a large amount of probability are again uniformly partitioned by a grid. This partitioning continues until each leaf bucket has few enough points. As in the one-dimensional case, the most important of these decompositions simply splits each dimension in half. In one dimension this provides each decomposed bucket with two children and the structure was called a binary trie. In k-dimensions, each split bucket will have 2^k children. We may decompose an image or other two-dimensional space by using a *quad-tree* [117,37] which is more properly called a *quad-trie* (figure 2c). This is a tree whose internal nodes have four children corresponding to the four quadrants of the square represented by the parent. Similarly, three-dimensional space may be decomposed using an *oct-tree* [33]. This is a tree whose internal nodes have eight nodes representing the eight octants of the cube represented by the parent. In higher dimensions this kind of structure tends to use a large amount of memory for a given accuracy of representation.

***K-d* tries.** The other generalization of the one-dimensional trie is the *k-d trie* [98]. The most important case is the binary *k-d* trie (figure 2d). This is a multi-level structure which splits dimensions in half, but unlike the multi-dimensional grids, a bucket is split along only one dimension at a time. Along with a left and right pointer as in the 1-d trie, each node of a k-trie contains an integer between 1 and k called the dimension number which specifies which dimension is split. Some authors require the dimension number to cycle through the dimensions, so that all nodes at level L are cut along dimension L mod k. By conceptually combining each set of k successive levels, we see that such cyclic trees are almost the same as multi-dimensional grids. The real representational power of a general *k-d* trie arises from the ability to finely partition those dimensions in which a large amount of variation occurs while only coarsely partitioning the other dimensions.

As in the one-dimensional case, the fact that the cut locations always halve the buckets simplifies both searching and dynamic structure alteration [98]. Whether to go left or right at a given level is determined by the $(j+1)$st bit of the correct component of the input vector, where j is the number of levels above which were split on the same dimension. As in the one-dimensional case, *k-d* tries do not adapt themselves as precisely to the data as the corresponding tree structure.

***K-d* trees.** Perhaps the most important of the high-dimensional structures is the *k-d tree* [11,10] (figure 2e). As with a *k-d* trie, a *k-d* tree corresponds to a decomposition of a k-dimensional Euclidean space into hyper-rectangles. It is an ordinary binary tree, but each internal node has room for both a discrimination value and a dimension number. The root of the tree represents the entire space I. The dimension number associated with the root specifies the first dimension along which the space is cleaved. The discrimination value specifies the location of the cut along this dimension. If the root's dimension number is i and its discrimination value is v, then the decomposition of the space is by the hyperplane defined by $x_i = v$. Its left child represents the half-space of all points satisfying $x_i \leq v$ and its right child the half-space of all points satisfying $x_i > v$. In general each node of a *k-d* tree corresponds to a hyper-rectangular piece of the space in which some of the dimensions may be infinite. The set of nodes at a given level of the tree correspond to a set of hyper-rectangles which partition the space. The leaves of the tree correspond to hyper-rectangles forming the finest partition of the space.

Searching a *k-d* tree for the leaf bucket containing a query point is easily accomplished in a single traversal of the tree from root to leaf. At each node, we compare the value of the point's coordinate in the dimension specified in the node with the discrimination value stored in the node. The traversal proceeds to the left or to the right depending on the outcome of this comparison. A well-built tree will have only $\log_2 n$ levels when there are n hyper-rectangles in the leaf partition. The search time is then logarithmic in the number of stored entries.

There are several schemes to make *k-d* trees that support insertions and deletions while preserving their useful properties. Such dynamic *k-d* trees may be implemented by periodically rebuilding portions that violate desired constraints [101,102]. k-dimensional versions of B-trees have also been designed [118,100]. It has been suggested that in very high dimensional spaces, better performance is obtained by combining key coordinates together, effectively making the dimension of the tree smaller [28]. We will investigate modules which reduce dimension in section 7.2.

Hybrid structures. A wide variety of hybrid structures may be constructed by viewing the input space as a product of smaller spaces and using different combinations of the above structures to decompose the smaller spaces. For example, one might think of a $2n$-dimensional space as the product of two n-dimensional spaces each with their own *k-d* tree decompositions. The product space is partitioned into regions consisting of all products of a leaf region from the first tree with a leaf region from the second tree. *K-d* trees can only represent a distribution well if they are based on enough sample points. Forming a *k-d* tree using only a subset of the dimensions is the same as projecting the probability distribution defined on the whole space onto the smaller space coordinatized by the chosen dimensions. Even if the representation of original distribution is too sparse for an effective decomposition of the whole space, the representation of the projected distribution may be quite adequate.

Data and the input space. The fundamental idea of all of these data structures is to connect input space properties with data properties. An incoming query is described in terms of input space coordinates and the data structure's job is to convert this to something in terms of stored data. Each node represents both a set of data elements and a region of the input space. The region of the input space is what we have been calling the bucket defined by the node. In the multi-level structures, the structure of the bucket is determined as we proceed from the top of the structure down. The stored data points corresponding to a node are obtained by starting at the node and following all paths to levels below it. The spatial structure is more accessible at the top and the elements are more accessible at the bottom. The structures based on address computation do not have many levels and closely associate the data points with the bucket regions.

3.3 Probability distributions and computation

The power of the algorithms described in this paper is that they create data structures which adapt themselves to the distribution of the data that they must store independently of the form of that distribution. The science of making inferences from data described by an unknown distribution is known as nonparametric statistics. There are two fundamental probability distributions that arise in analyzing the behavior of points drawn from an arbitrary distribution: the beta distribution and the Poisson distribution. This section will describe these distributions and will use them to discuss computational methods for estimating quantities needed in building data structures.

Binomial distribution. Nodes in each of the data structures we have discussed correspond to subsets of the input space I which we have called bins or buckets. We must understand both how to choose the buckets and how query points are likely to fall in them once they are chosen. If we have a fixed bin $B \subset I$, and we draw N points from I according to ρ, we may ask what the probability is that j points will be chosen from B. The probability that a single point will be drawn from B is the total probability density contained in B:

$$P(B) = \int_B \rho(x)dx. \tag{3.6}$$

The probability that j points fall in B is then described by the binomial distribution:

$$p_N(j) = \frac{N!}{j!(N-j)!}P(B)^j(1-P(B))^{N-j}. \tag{3.7}$$

Beta distribution. In tree structures, we must choose real partitioning parameters on the basis of observed data points to build buckets with good properties. We therefore need to understand how the distribution of contained points varies as we look at successively larger members of a one-parameter family of regions. For simplicity, let us assume that the probability distribution ρ does not vanish anywhere in the input space I, but is otherwise arbitrary. Let us consider an arbitrary one-parameter family of nested subsets R_γ of S. These sets are parameterized so that:

$$R_{\gamma_1} \subset R_{\gamma_2} \text{ iff } \gamma_1 \leq \gamma_2. \tag{3.8}$$

In this case, we may redefine the parameter γ to be the total probability contained in a subset (i.e. the measure of the set):

$$\gamma = \int_{R_\gamma} \rho(x)dx. \tag{3.9}$$

We assume that $R_0 = \emptyset$, the empty set, and that $R_1 = I$, the entire set. For example, the family R_γ might be a set of nested spheres or hyper-rectangles about a point $x \in S$.

Let us now draw N points from I according to the distribution function ρ. We use this set of sample points to choose the subset R_γ which is the smallest set in the family that still contains one of the sample points. What is the distribution $p_1(\gamma)$ of the γ's we get by applying this procedure repeatedly? $p_1(\gamma)d\gamma$ is the probablility that $R_{\gamma+d\gamma} - R_\gamma$ (the complement of R_γ in $R_{\gamma+d\gamma}$) contains one point and all $N-1$ others lie in $S - R_\gamma$. By the way in which we defined γ, the probability of a point lying in $R_{\gamma+d\gamma} - R_\gamma$ is $d\gamma$. There are N different points which can be chosen to lie in this interval. The probability for a single point to lie in $S - R_\gamma$ is $1-\gamma$, implying that:

$$p_1(\gamma)d\gamma = N\ d\gamma\ (1-\gamma)^{N-1} \tag{3.10}$$

or equivalently:

$$p_1(\gamma) = N(1-\gamma)^{N-1}. \tag{3.11}$$

In general we will be interested in $p_n(\gamma)$ which we define as the probability that R_γ is the smallest of the R's which contains exactly n sample points. $p(\gamma)d\gamma$ is the probability that one of the points lies in the shell $R_{\gamma+d\gamma} - R_\gamma$, $n-1$ of the points lie inside R_γ and $N-n$ of the points lie outside of it in $S - R_\gamma$. Introducing the combinatorial factor which counts the number of ways in which we may assign points to these tasks, we obtain:

$$p_n(\gamma)d\gamma = \frac{N!}{(n-1)!1!(N-n)!}\gamma^{n-1}d\gamma(1-\gamma)^{N-n}, \tag{3.12}$$

so

$$p_n(\gamma) = \frac{N!}{(n-1)!(N-n)!}\gamma^{n-1}(1-\gamma)^{N-n}. \tag{3.13}$$

This is the Beta distribution [76]:

$$\beta(\gamma) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}\gamma^{a-1}(1-\gamma)^{b-1} \tag{3.14}$$

where $\Gamma(a)$ is the gamma function, $a = n$, and $b = N - n + 1$.

The beta function integral:

$$\int_0^1 x^{m-1}(1-x)^{n-1}dx = B(m,n) = \frac{\Gamma(m)\Gamma(n)}{\Gamma(m+n)}, \tag{3.15}$$

allows us to find the moments of p_n:

$$\langle \gamma^j \rangle_{p_n} = \frac{N!(j+n-1)!}{(N+j)!(n-1)!}. \tag{3.16}$$

In particular the mean is

$$E(\gamma) = \langle \gamma \rangle = \frac{n}{N+1}, \tag{3.17}$$

the standard deviation is

$$\sigma^2(\gamma) = \langle \gamma^2 \rangle - \langle \gamma \rangle^2 = \frac{n(n+1)}{(N+2)(N+1)} - \frac{n^2}{(N+1)^2} = \frac{n(N-n+1)}{(N+2)(N+1)^2}, \tag{3.18}$$

and two moments we will need later are

$$\langle \gamma^{-1} \rangle = \frac{N}{n-1}, \tag{3.19}$$

and

$$\langle \gamma^{-2} \rangle = \frac{N(N-1)}{(n-1)(n-2)}. \tag{3.20}$$

If we take a large number of sample points N compared to the number of points in a region n, the mean and standard deviation asymptotically approach:

$$E(\gamma) \sim \frac{n}{N} \tag{3.21}$$

and

$$\sigma^2(\gamma) \sim \frac{n}{N^2}. \tag{3.22}$$

Density estimation. We often need to estimate the value of a probability density $\rho(x)$ at a specified point x. Such an estimate might be used to construct an efficient data structure as when a binary tree is constructed to make each child is equally likely. In other situations the estimated distribution itself contains the essence of the desired information as when sample clusters are chosen on the basis of it.

Binning method. One common technique for estimating ρ begins by decomposing the sample space into B equal sized bins of volume V. A histogram of the points in these bins is then formed by counting how many of the N sample points lie in each bin. If N_i samples lie in the ith bin, then the probability density is estimated as:

$$\rho_b(x) = \frac{N_i}{NV} \tag{3.23}$$

where i is the bin containing the point x of interest. There are a number of difficulties with this technique. Because the estimated distribution ρ_b predicts the same density at each point in a bin, any features smaller than a bin will be lost. If the bins are too big, they will wash out important fine structure in the distribution ρ. On the other hand if the bins are too small, in regions of low probability the expected number of points in a bin can fall below one. In this case a low probability region will be represented by one bin with an estimated density that is too high surrounded by empty bins with estimated probability zero. Without knowing much about the probability density, it is difficult to choose proper-sized bins. If N is too small there may be no way to keep the bins from being too small in some regions and too large in others. The process of binning throws away all information about where sample points lie within bins, suggesting that better procedures might exist. More discussion of the problems of binning may be found in [96,106,42].

Let us analyze the performance of bin estimation on a uniform distribution with probability density:

$$\rho(x) = \rho_0. \tag{3.24}$$

The statistics of the binning algorithm are governed by the binomial distribution. The probability of a single sample lying in the bin of interest is $\rho_0 V$ where V is the volume of a bin, as above. After choosing N samples, the probability $p(n)$ of having chosen n samples from the bin of interest is:

$$p(n) = \frac{N!}{n!(N-n)!}(\rho_0 V)^n (1-\rho_0 V)^{N-n}. \tag{3.25}$$

The mean number of samples in our bin is then

$$\langle n \rangle = N\rho_0 V \tag{3.26}$$

and the standard deviation is:

$$\sigma^2(n) = N\rho_0 V(1-\rho_0 V). \tag{3.27}$$

As above, we define the estimating distribution:

$$\rho_b = \frac{n}{NV}. \tag{3.28}$$

This estimate is again correct on average:

$$\langle \rho_b \rangle = \rho_0. \tag{3.29}$$

In this case the standard deviation of the estimate is:

$$\sigma^2(\rho_b) = \frac{1}{N^2V^2}\sigma^2(n) = \frac{\rho_0(1-\rho_0 V)}{NV}. \tag{3.30}$$

Volume method. An estimation technique which underlies many the structures discussed in this paper considers not how many points lie in a given volume, as in the bin technique, but rather considers what volume is needed to contain a certain number of points [72]. For example, we may estimate $\rho_v(x)$ of ρ at the point x as the inverse of the volume of the smallest sphere centered at x which contained n sample points times roughly n/N. We will see that a better estimate is:

$$\rho_v(x) = \frac{n-1}{NV_n(x)} \tag{3.31}$$

where $V_n(x)$ is the volume of the smallest ball centered at x which contains n points. In low probability regions, the ball will be large and so based on a reasonable number of sample points. In high probability regions, the ball will be small and thus capable of high spatial resolution. Because each estimate is based on n points, one expects the estimates to be equally supported by evidence in all regions. Because the ball size is the smallest possible consistent with the desired amount of evidence, the technique adapts itself to a distribution as best it can with N sample points. We shall see that it does reasonably well even with a very small number of samples.

In the constant density case, the actual probability contained in a region of volume V is $\rho_0 V$. The beta distribution describes the probability γ contained in the region V_n. The probability distribution for V_n is then:

$$p(V_n) = \rho_0 \frac{N!}{(n-1)!(N-n)!}(\rho_0 V_n)^{n-1}(1-\rho_0 V_n)^{N-n}. \tag{3.32}$$

The mean volume is:

$$\langle V \rangle = \frac{n}{(N+1)\rho_0}. \tag{3.33}$$

With the definition above for ρ_v, we see that:

$$\langle \rho_v \rangle = \left\langle \frac{n-1}{NV} \right\rangle = \frac{\rho_0(n-1)}{N}\langle \gamma^{-1} \rangle = \rho_0. \tag{3.34}$$

This justifies the definition of ρ_v. With samples drawn from a uniform distribution, the above procedure will produce estimates of that distribution which are correct on average. Let us compute the standard deviation of these estimates:

$$\sigma^2(\rho_v) = \langle \rho_v^2 \rangle - \langle \rho_v \rangle^2 = \frac{\rho_o^2(n-1)^2}{N^2}\langle \gamma^{-2} \rangle - \rho_0^2 = \rho_0^2 \left(\frac{N-n+1}{N(n-2)} \right). \tag{3.35}$$

It is of interest to compare the relative deviations σ^2/ρ^2 of the estimated probability density for the two techniques. For the binning technique, we find:

$$\frac{\sigma^2}{\langle \rho_b \rangle^2} = \frac{1-\rho_0 V}{NV\rho_0} = \frac{1}{NV\rho_0} - \frac{1}{N}. \tag{3.36}$$

For the volume technique, we find:

$$\frac{\sigma^2}{\langle\rho_v\rangle^2} = \frac{N-n+1}{N(n-2)} = \frac{1}{n-2} - \frac{n-1}{N(n-2)}. \tag{3.37}$$

If N is less than $1/V\rho_0$, then the deviation of the binning technique is as large as the estimate, which is indicative of the all or none binning noise we discussed above. When ρ varies, this problem will occur in the regions where ρ is small. The relative deviation using the volume technique, however, is independent of the density and goes to zero with increasing n. By letting n grow with N, the volume technique can do as well as the bin technique even in high probability regions. Asymptotically, n should be chosen so that $n \to \infty$ but $n/N \to 0$ as $N \to \infty$. Reference [72] suggests that $n = \sqrt{N}$ is a good choice and [42] gives a functional form for the optimal choice of n under certain assumptions.

In section 4.1 we will discuss methods for finding a point's n nearest neighbors in time $O(n)$. For small n these methods can be used to give very fast density estimates. For large n, as required by the optimal choice above we suggest using the partial summation technique from section 3.2. Nothing in the derivation above required the use of the spherical volumes implied by considering the n nearest neighbors. We can just as well use the volume of the smallest hypercube centered at the point of interest which contains n sample points. If we store the points in an adaptive grid, the partial summation trick lets us determine the number of sample points in a hyper-rectangular region in constant time. Using a binary search, we can find that volume which contains n points by looking at a maximum of $\log_2 N$ hypercubes. We have found this technique useful for estimating various densities in images. It also gives an efficient means for implementing the histogramming technique in [99] and gives insight into what the peaks found by the method in [95] actually are.

Median estimation. In building trees and *k-d* trees, we would like to choose discriminators at the median of the underlying distribution in the region corresponding to a node. This implies that a query search which reaches that node is equally likely to choose the left or right branch. In this situation we want to cut a hyper-rectangle perpendicular to one of its axes. To apply the analysis leading to equation 3.13 we choose the nested family of regions to be the left region in the hyper-rectangle as the cut plane goes from left to right. We would like to choose that cut for which γ, the total probability content of the left portion, is equal to $1/2$. We must make this choice on the basis of N input samples in the region of interest. From equation 3.17, the expected γ if we cut at the nth sample point is $n/(N+1)$. If N is odd, then choosing the middle point (i.e. $n = (N+1)/2$) will give $\langle\gamma\rangle = 1/2$ with standard deviation

$$\sigma^2(\gamma) = \frac{1}{4(N+2)} \sim \frac{1}{4N}. \tag{3.38}$$

Notice that of all n's this is also the choice for which the estimated volume has the lowest standard deviation. If the density is uniform near the median, then when N is even, we can choose the cut to be half way between the center two points with similar results.

Boundary estimation. A problem which is of great importance to classification is the estimation of the boundary of a region from N points drawn from it. An important special case is a hyper-rectangular region with a uniform probability distribution. One could estimate the region by choosing the smallest hyper-rectangle which contains all the sample points, but it is intuitively clear that this is likely to cut off some of the probability around the outside. If we project the sample points onto one of the axes, we can compute how far the leftmost point in a sample of N points drawn uniformly from an interval of length L is likely to be from the left end of the interval. We let the nested family of regions be sub-intervals starting at the left end of the interval. Let us denote the length of the smallest of these intervals which contains n samples by L_n. From equation 3.33, the mean value of L_n is

$$\langle L_n \rangle = \frac{nL}{N+1} \tag{3.39}$$

and the standard deviation is

$$\sigma^2(L_n) = \frac{L^2 n(N-n+1)}{(N+2)(N+1)^2}. \tag{3.40}$$

A simple way to estimate L is to measure the distance between the ith and jth points and assume this distance is equal to:

$$\langle L_i - L_j \rangle = \frac{(i-j)L}{N+1}. \tag{3.41}$$

For example if x is the distance between the leftmost and rightmost points, then we estimate the width of the region as

$$L \approx \frac{(N+1)x}{N-1}. \tag{3.42}$$

If the density is really constant, we can get much better estimates by utilizing all the sample points. If the density varies, however, we would like to estimate how far the interval extends beyond the leftmost sample by using only samples near the left end. Notice that the extension of the interval beyond the leftmost point is L_1 and that this is equal on average to the distance between the first and second points $L_2 - L_1$. A reasonable procedure to apply when the density varies is to extend each boundary of the hypercube out from the first sample point it hits by a distance equal to that between the first and second sample points.

Error bounds. Many of the algorithms in later sections create structures from sample inputs that are meant to deal well with future inputs drawn from the same distribution. A useful proposition from [132], based on the Chernoff inequality may often be used to bound the probability that there is a large set of inputs from an unseen part of the input space. $L(h, S)$ is defined as the smallest number of independent trials, each with a probability at least h^{-1} of success, such that the probability of fewer than S successes is less than h^{-1}. For integers $S \geq 1$ and real numbers $h > 1$, the proposition states that $L(h, S) \leq 2h(S + \ln h)$. If there are S categories, by choosing L sample points, the proposition guarantees with probability h^{-1} that the categories from which no samples have been drawn have a combined probability of less than h^{-1}.

Bin filling time. When the categories are equally likely, we may do much better than this bound on average. If we draw samples from S equally likely categories, then after we have samples from j of the categories, the probability of sampling a new category is $(S - j)/S$. The mean number of trials required to achieve this success is $S/(S - j)$. The average total time to draw a sample from each category is then:

$$\sum_{j=0}^{S-1} \frac{S}{S-j} = SH_S, \tag{3.43}$$

where $H_S = \sum_{i=1}^{S}(1/i)$ is the Sth harmonic number. It is easy to see that $\ln S < H_S \leq \ln(S + 1)$ by comparing the sum of $1/x$ with its integral. The average number of trials needed to draw one sample out of each of S equally likely bins is therefore less than or equal to $S(\ln S + 1)$.

3.4 Building data structures

We now consider the problem of building the data structures that we have described. Let us begin with the one-dimensional structures. As in choosing a representation, in the building of structures there is also a subtle interplay between using the structure of the stored data and the structure of the embedding space.

Bucket sort. The address-based structures are extremely easy to build. In most cases, a single pass is made through the data in which each entry is simply inserted into the proper location. For example, once an array is declared and filled with null pointers, we may fill it in a single pass through the data. The array location is obtained directly from the key of the entry, and the data is inserted at the head of the linked list stored there. When the data is uniformly distributed, each list will have only a few entries. It is useful to note that in this case we may make a single pass through the array to read the data entries off in sorted order. The discussion of the binomial distribution in section 3.3 shows that uniformly distributed data may therefore be sorted in only linear time.

Distribution sorts. In building the two level array, we must decide on the number M_b of sub-buckets into which each top level bucket b should be split. In [94] it is suggested that the data first be subsampled to obtain statistics on which to base the choice of the M_b. Every nth element is used to increment a counter in the corresponding top level bucket. The counter value in each bucket divided by the size of the subsample, estimates the probability of the corresponding bucket. The discussion of bin estimation in section 3.3 tells us how good this estimate is likely to be for a given distribution. A single linear pass through the top level array can then parcel out the desired total number of sub-buckets according to these estimates. The computed M_b's are stored in each top level array slot along with a pointer to an array of size M_b. A single pass through the data can now insert each element into the proper sub-bucket. Again, in a single pass the data may be read out in sorted order. This procedure can be used to sort data that is distributed according to a variety of distributions in linear time. In all cases [94] finds this algorithm to be substantially faster than the fastest implementation of quicksort.

Building tries. Tries may be built by sequentially inserting each element. One begins by initializing the root array with null pointers. Elements are added to the buckets determined by their addresses until a bucket gets larger than a desired size. A new node on the next level is then created and the entries in the bucket are redistributed into the correct locations. The average time to create a trie with N nodes is $N \log_2 N$. If the branching number of a trie is a power of two, then the array locations needed in inserting an element are given by substrings of the binary representation of the address. The structure of the trie is independent of the order in which elements are inserted.

Building trees. To build a binary tree, we must choose the partitioning values at each node in addition to inserting the data elements. To obtain optimal search times, we would like the probability of going to the left at each node to be equal to that of going to the right. As we showed in section 3.3, this behavior is best approximated by building a balanced tree, i.e. the buckets should be divided in such a way that half of the elements contained in them are to the left of the partition and half are to the right. This suggests an approach to building a balanced tree in which we first sort the data items. Let us assume that sorting produces an N element array of pointers to data elements in which the pointer stored in the ith position points to the ith largest element. The tree may be built recursively. If there is only one data element, the tree consists of a leaf with a pointer to that element. Otherwise, the root partitioning value is be taken to be the element stored at array location $\lfloor N/2 \rfloor$, the left pointer points to the tree built over the elements with indices less than the partitioning element and the right pointer points to the tree built over elements with indices greater than the partitioning element. For the sort, we may use one of distribution sorts described above, or quicksort which has a structure quite similar to that of the tree.

Quicksort. In building up the tree, the partitioning value is the median of the stored values. Instead of sorting all the elements to begin with, we may attempt to find this median value directly. This approach will be useful in building multi-dimensional trees. To understand how to find the median efficiently, we must first consider the mechanism underlying quicksort [56]. In quicksort we choose a random element e and partition the original set into those elements less than or equal to e and those greater than e during a single pass through the data. The two partitioned pieces are recursively sorted using quicksort and the resulting lists are concatenated. A randomly chosen element is a rough estimate of the median of the set. We can increase the likelihood that the choice is close to the median by randomly choosing three or more elements and partitioning on the middle element. If the partition value really is near the median, then the two halves will each have roughly half the elements. In this case, the recursion ends after $\log N$ stages. Each element is examined during each stage causing the entire sort to take average time $N \log_2 N$.

Median finding. If we are only interested in finding the median value, we can get linear time performance using a very similar procedure [2]. Again we partition the elements into those greater than and those less than a randomly chosen element. By counting how many items are in each group, we can determine which group contains the median element. If less than half of the elements are in the lower partition, the median is in the upper group and otherwise it is in the lower group. In this situation we need only recursively consider one of the two partitions. If it is the lower one, we look for its $\lfloor N/2 \rfloor$nd element. If it is the upper one, we look for the element whose rank is $\lfloor N/2 \rfloor$ minus the number of elements in the lower partition. On average, we again have $\log_2 N$ stages, but now each stage only examines about half the elements examined during the previous pass. The average total number of steps is $N + N/2 + N/4 + \ldots$ which is of order N. The same approach may be used to find the nth largest element in linear time, for arbitrary n.

The same idea may be extended to finding the median in probability of a set of elements which are explicitly assigned probability weights. Denote the probability of each data element d by $p(d)$. Such quantities may have been collected during a statistics gathering phase. We want to find an element whose probability of being larger than a randomly chosen element is as close as possible to $1/2$. We apply the same approach as above except that during each stage we count the total probability instead of the number of elements. For example, at the first partition we choose the smaller half if its total probability is greater than $1/2$ and the larger half otherwise. Again the median may be found in linear time on average.

To build a binary tree using the median selection approach, we again proceed recursively. If there is a single element, the root just points to it. Otherwise, we select the median at the first stage and partition the elements into those smaller than it and those larger than it. The algorithm is applied independently to the two halves and the root is made to point to the two resulting trees. The time for this procedure is again $N \log_2 N$ since there are $\log_2 N$ stages and 2^i medians are found in sets of size $N/2^i$ at the ith stage. The actual amount of work required is a constant factor greater than in the quicksort approach, but this approach generalizes to building multi-dimensional trees.

Building grids. The address based multi-dimensional structures can be built just like the one-dimensional ones. Grids and multi-level grids can be built just like arrays and tries. To build adaptive k-dimensional grids, we build k one dimensional structures, one based on each of the coordinates. The array is then allocated and the points are inserted by sending their coordinates through the one-dimensional structures. One can think of this as projecting the points onto each of the axes and choosing equiprobability partitions for this projected distribution of points. For k-d tries, the splitting dimension needs to be chosen for each node. The issues here are the same as for k-d trees, which we discuss next.

Building k-d trees. The k-d tree will be used in many of the module algorithms. In typical applications the leaves will store a the set of sample data points that lie inside the corresponding hyper-rectangle. The partitioning of the tree is chosen to reflect the probability distribution of the stored points. In the applications, the tree should be approximately balanced and each leaf region should be approximately cubical and should contain approximately the same amount of probability density. The volume of a leaf region at a given point should be approximately inversely proportional to the probability density there. If this can be accomplished, the tree lets us access data according both by where it is and by how it is distributed. This will allow us to implement a variety of useful associative functions in the next section.

Given a set of points drawn from a probability distribution, we can build a k-d tree with the above properties if there are enough points to represent the distribution well, i.e. the distribution should not vary much between neighboring data points. We may build a perfectly balanced tree by a simple recursive procedure that is exactly analogous to the one-dimensional procedure. There are several heuristics available which may be used to decide along which dimension a given node should be cut. These heuristics amount to choosing that dimension in which the points are most spread out. Near the top of the tree, the distribution is likely to vary considerably within the region represented by a node. The most spread dimension may be defined as the dimension in which the data points have the greatest standard deviation or as the one in which the nth smallest component of any point and the nth largest component of any point are furthest apart. Both of these statistics (and a variety of others) can be determined in linear time and need be based on only a subsample of the data. The techniques of section 3.3 allow one to estimate how well these statistics are likely to represent the underlying distribution. Another idea is to form a coarse grid (say three on a side) and in a single pass through the data determine the number of entries in each grid block. These sums may then be used to estimate the most spread dimension. As we get closer to the leaves of the tree, the probability density becomes uniform within the hyper-rectangles corresponding to nodes. In this case the dimension in which the distribution is most widely spread is simply the longest dimension of the hyper-rectangle.

Once the splitting dimension is chosen, we would like to choose the discrimination value so that half the probability inside the parent's hyper-rectangle lies to one side of the partition and half to the other side. We may find this value applying the one-dimensional median finding algorithm or its weighted version to the chosen coordinate of the stored points. If we allow the recursion to proceed to the same depth everywhere, we obtain a perfectly balanced *k-d* tree. Each leaf will have the same number of points in its corresponding hyper-rectangle (plus or minus 1 if the number of points is not a power of 2). This means that with high probability the leaf regions contain roughly equal portions of the underlying probability distribution ρ.

If the number of points is large enough that the probability distribution is approximately constant in the hyper-rectangles associated with the lower nodes, then we can guarantee that the leaf hyper-rectangles will be approximately cubical. Because the density in the node regions becomes constant near the leaves, the construction is well approximated by an algorithm which splits the longest side of a hyper-rectangle exactly in half on each cut. If the initial ratio of longest to shortest side is α, then in $k \log_2 \alpha$ cuts we are guaranteed that no side is less than half the length of any other side. This is because if there are two sides of length a and b, then in $|\log_2(a/b)|$ splits, the longer side will become shorter than the other side but not less than half its length. This condition persists if we continue to cut the larger of the two in half. We get the bound above in higher dimensions if we always cut the longest side in half.

4. Directed discrete modules: associative memory

In this section we consider modules in which the input to output mapping is specified by a "teacher" and whose outputs are drawn from a discrete set. We can think of the N possible outputs as representing N distinct "memories." The set of inputs which produce a given output may be thought of as the stimuli which are associated with that memory. One of the possible outputs may represent the absence of any memory associated with the given input. We shall call such an output the null output.

In the simplest circumstance, the teacher provides $N-1$ input-output pairs which are to be associated and the response to any other input is to be null. We may implement such a map using any of the data structures discussed in section 3.1. The specified inputs are stored in the data structure along with their desired outputs. When the key of an input request is found in the structure, the accompanying value is output, otherwise the null value is output. The only issues are the efficiency of access and the efficiency of storage. Using hash tables we obtain the best possible results: retrieval in constant expected time and $O(N)$ storage.

The situation is slightly more interesting when the input-output pairs are presented dynamically. In this case the presentation of input-output pairs to be learned is interspersed with retrieval requests. Much of the development of sophisticated dynamic tree balancing algorithms was stimulated by the requirements of exactly this kind of dynamic data base [80]. If the module is supposed to have a fixed capacity, then criteria must be chosen to decide which memory to forget when a new element is to be stored. Such a choice might be based on query request statistics that can be stored along with each item. Typical forgetting rules are to forget the least recently accessed memory or the least often accessed memory. A number of interesting data structures have been designed which are self organizing in the sense that the most popular items are easiest to retrieve and the least likely items get harder and harder to access (say by moving onto disk) until they are finally forgotten. One of the nicest of these is the splay tree. This is an ordinary binary tree, but when an item is accessed the tree structure is altered so as to make the accessed item become the root. The analysis of this tree's behavior shows that on average the tree remains balanced and the less likely an item is to be queried, the farther it tends to reside from the root [122].

4.1 n nearest neighbor generalization

A module generalizes when it produces an output other than null on inputs which were not directly associated with that output during training. One of the best techniques for generalization produces the output associated with the stored input which is nearest the unknown input in the input space I. Reference [25] shows that the probability of making an error by using this procedure is less than twice the probability of making an error using any other procedure based only on sample points. A generalization of this idea looks at the n nearest neighbors and lets them vote on the output. In the next section we will consider procedures that interpolate between the outputs of the n nearest neighbors.

Computing the n nearest neighbors of a point has traditionally been viewed as an extremely desirable, but very expensive operation. Many systems compute the distance from the given point to *every* stored point and then choose the n points with the smallest distances. Many speech recognition systems, for example, compare an incoming word with every word in their dictionaries. Reference [43] suggested using the branch and bound technique for cutting down the number of needed comparisons. The data points were to be first clustered and the search for near neighbors begun in a single cluster. When the nth closest point already seen was closer than the boundary of a cluster, there was no need to examine the points in that cluster.

Reference [40] then suggested that k-d trees were an efficient and natural mechanism for implementing the clustering. This reference proves that when the nodes are partitioned along the most spread dimension, the expected number of leaf buckets which intersect the smallest sphere containing a point's n nearest neighbors is only $O(n)$. The branch and bound technique is easily implemented, since the distance to a hyper-rectangle is the distance to its nearest corner. The logarithmic search to find the leaf bucket containing an input point takes time $\log_2 N$. The time to find the n nearest neighbors is therefore only of order $(\log_2 N) + n$. The nearest neighbor is likely to be in a bucket near the query point's leaf bucket.

Reference [63] suggests using what we have called an adaptive grid to cut the nearest neighbor search time down to $O(1)$. This reference suggests using hashing to access the correct grid location but does not say how this might be done. It appears that the bin merging technique discussed in section 3.2 is ideal for this application.

We have found that in many circumstances, obtaining the exact nearest neighbor is not important and that a nearby neighbor may suffice. In this case, we do not need to implement the branch and bound check. If there are enough sample points, the leaf regions of the *k-d* tree are approximately cubical and each contain the same number of samples. The nearest neighbor of a point is very likely to lie in the same bucket as the sample point or in the neighboring bucket on the side closest to the sample point. By choosing the nearest neighbor out of the points in just these two buckets, we make the search algorithm extremely fast and trivial to implement without sacrificing much of its utility.

An interesting use for the ability to quickly retrieve the n nearest neighbors arises when we consider a network of modules. As evidenced by the disease of epilepsy, one of the problems with networks of mutually excitatory cells (such as the pyramidal cells in cortex) is the tendency toward an instability which leads to explosive firing activity. As suggested in [18] many of the reciprocal bundles of fibers in the brain may function to create a negative feedback loop among the computational areas. Early areas along the input-output pathway should generate just the amount of output that later areas can use. When later areas begin to recieve too much input, they dampen out the earlier areas via the feedback paths. The same idea may be applied to a network of modules. Each module would receive a "level of activity" input commanding it to produce a set of outputs corresponding to successively further nearest neighbors of the current input. If the system is working on a set of inputs with a straightforward and unambiguous interpretation, it will quickly generate appropriate outputs. If the input is more ambiguous, then later modules should increase the activity of the earlier ones to allow looser and looser interpretations of the input. It appears that this kind of analog feedback might be quite useful when applied to information flow even in digital systems.

4.2 Classification heuristics

The n nearest neighbor approach always classifies an input into some category. In many circumstances, some inputs should produce the null output. For example, we have built an optical character recognition system that should produce null if it is presented with a character which is too unlike any it had been trained on. The system begins its computation by quickly extracting several scale-independent parameters from the pattern of pixels which make up a letter. It must classify an unknown letter from these parameters using a data structure constructed on the basis of labelled sample points in the parameter space. Each character is represented by a swarm of data points and these should determine disjoint classification regions. The nearest neighbor approach will categorize every query with the stored point nearest to it.

One way to restrict the extent of the classification regions is to search for points in a fixed radius of the sample points. Efficient algorithms for this task are discussed in [16] and [17]. Unfortunately, the optimal radius to search within varies from region to region. High probability categories with small regions are likely to be fairly densely covered with sample points while low probability categories with large regions are likely to be only sparsely covered.

We have used several variants of the *k-d* tree construction algorithm presented in section 3.4 to deal with this situation. The first difference between these algorithms and the *k-d* algorithms is that we choose the splitting dimension and the discrimination value to optimally separate the categories in addition to evenly dividing up the probability. In the optical character recognition example, a cut through a hyper-rectangle is given a score based on how many character categories are completely to one side or the other of the cut, how evenly the hyper-rectangle is cut, and how evenly split any categories that are cut are.

This cutting procedure proceeds until a hyper-rectangle contains only entries from a single class. The resulting leaf hyper-rectangles are then shrunk down on the enclosed samples using the boundary estimation technique discussed in section 3.3. Thus the second point of departure from standard *k-d* trees is that the leaf regions do not cover the entire space. In some situations it may be important to do the shrinking at each level of the tree, because the optimal splitting dimension according to some heuristics may change after shrinking. The classification regions are represented as unions of hyper-rectangles and points which fall outside of any leaf region are classified as null. Classification of a point requires only a single traversal of the tree and is therefore extremely fast. A statistical justification for a similar approach to classification is presented in reference [19]. Section 6 discusses similar data structures for categorizing data without a teacher.

The fast nearest neighbor classification technique should be useful for real-time speech recognition. A standard approach to speech recognition [71] is to compute the distance between an unknown word and each stored sample using dynamic time warping. This uses a dynamic programming algorithm to choose the optimal distortion of the time axis when matching two time series. Unfortunately, the use of warping means that there is no obvious way to apply the branch and bound technique discussed above and an unknown word must be compared with every stored sample. An alternative approach is parameterize a time series by arc-length in the parameter space instead of by time [130]. This eliminates the warping step; to compare two words, we need only compare two curves in parameter space, independent of their time parameterization. This allows us to introduce a distance measure between words which satisfies the triangle inequality. The samples may be stored in a *k-d* tree and the nearest match may be found in logarithmic time. Speech recognition also appears to be an ideal candidate for the shrunken hyper-rectangle classification technique.

Incremental learning In this section we briefly consider incremental learning. We assume that the module is corrected whenever it misclassifies an input. A useful heuristic is to use nearest neighbor classification and to only store those data points which are misclassified. Any of the dynamic multi-dimensional tree techniques mentioned in section 3.2 can be used to hold the points. If memory space is an important issue, the data structure can be periodically "cleaned up" by removing redundant data points.

How many data points need to be seen before the classifications are likely to be correct? As a first estimate, we may ask how many samples need to be drawn to get representatives from all the likely types. As discussed in section 3.3, if there are S types and $2h(S + \ln h)$ samples then with probability h^{-1} the unchosen types have a combined probability of less than h^{-1}. As we saw in section 3.3, when the categories are equally likely we need only choose $S(\ln S + 1)$ samples on average to draw one from each category. If we decompose the classification regions into equi-probable hyper-rectangles using the algorithm above, then applying these results to the number of hyper-rectangles in the decomposition gives the order of the expected number of samples needed to achieve the desired level of decomposition.

4.3 Other generalization behaviors

Depending on the application, there are a variety of other useful ways to access hierarchically stored information. A common situation with multi-dimensional data is that only some of the dimensions will be specified. For example, if we are searching a visual database to do object recognition, then when an object is occluded by another object some of the features will be unavailable. A partial-match search is a search which returns all stored items which agree with the query in the specified dimensions and have arbitrary values in the unspecified dimensions. Range searching is a further generalization in which one wants to retrieve all stored data whose coordinates lie in specified ranges in each dimension. A range search operation returns all data points which lie in a specified hyper-rectangle. A variety of structures which support range searching are discussed in reference [14]. *K-d* trees may be used for efficient range searching by examining only those leaves which intersect the query hyper-rectangle. One proceeds from the root by taking the left branch, the right branch, or both depending on which intersect the query region. For cubical range regions, the number of entries that must be examined is of order $\log_2 N$ plus the number of points returned.

Another useful kind of query combines range searching with nearest neighbor searching. Some dimensions are given specified ranges and in others we are to find the closest m points. This kind of query can be implemented on a *k-d* tree in an obvious way. The branch and bound is carried out only in the nearest neighbor directions and all bins intersecting the specified ranges in the range dimensions are examined. This type of query is used in an alternative approach to learning and evaluating nonlinear mappings which we will discuss at the end of the next section.

A final important type of query is partial match when the component in each dimension is chosen from unordered sets with only a few elements. For example, the inputs might be Boolean vectors in which the dimensions correspond to features and the components of the vector specify the presence or absence of each feature. A query specifies the values of some subset of the components and the search is supposed to return all stored entries which have these values in the specified locations. Because the space is discrete, the geometric approach discussed above does not immediately help. Reference [111] suggests concatenating a few bits from each component of the key to make up a bin label. A partial match search will specify some of the bits in the bin label, and the search examines the contents of all the bins with the specified bits by letting the other bits run over all possibilities. Reference [3] investigates the optimal number of bits to choose from each key when the queries are governed by a known distribution. Further practical considerations are dicussed in the references [73,74]. Another approach is to store the records in a *k-d* trie. This has the advantage that if the specified bits are near the top of the trie, large chunks of the data base are pruned. Reference [13] discusses the choice of trie discriminators in this case.

If there are a large number of binary dimensions in the key and each has a low probability of being true, then using a bin label made by *or*ing dimensions together is useful. If the query specifies that a certain dimension is set, then we need not consider keys in bins whose label does not have the corresponding bit set. This situation arises, for example, when searching a database of documents for those documents which contain a given set of words or when searching a visual database for those objects which contain a given set of feature pairs joined in a specified relation.

5. Directed continuous modules: smooth mappings

In this section we discuss an important class of modules for building systems that interface with the physical world. These modules can efficiently represent, evaluate, and learn nonlinear mappings from one space to another. Such a capability is of critical importance in applications such as dynamic, adaptive control of robot manipulators or autonomous vehicles in changing environments and the decoding of visual and auditory data into geometric information about the objects which generated it.

A *mapping* assigns to each point in the input space I (also called the domain of the mapping) a point in the output space O (also called the range of the mapping). We will assume that the domain and range are subsets of Euclidean space. If the domain has dimension k and the range has dimension n, then R^k and R^n denote the Euclidean spaces in which they are embedded. We specify a point in either of these spaces by giving its coordinates, k real numbers for a domain point and n for a range point. The software or hardware modules we wish to construct take as input the k numbers specifying a domain point and should produce as output the n numbers specifying a range point.

In the simplest of mappings the input and output are linearly related. If v_1 is sent to u_1 and v_2 is sent to u_2, then for any real numbers α and β for which $\alpha v_1 + \beta v_2$ is in the domain, $\alpha v_1 + \beta v_2$ is sent to $\alpha u_1 + \beta u_2$. Most mappings are not linear, but linear mappings often make useful approximations. For smooth mappings, if u is sent to v, then small deviations from u map approximately linearly into small deviations from v. The smaller the deviation, the better the approximation (by Taylor's theorem). The linear map of deviations is called the linearization or the *Jacobian* of the original nonlinear map at the point u. In many physical systems, the origin in R^k represents the absence of activity and the origin in R^n represents the absence of response. In this case the response to small inputs will be approximately linear.

The idea proposed here is to represent an arbitrary nonlinear mapping by a piecewise linear mapping. We have seen many effective data structures which hierarchically decompose the input space and support efficient identification of the partition region containing a query point. In this section we will approximate an arbitrary mapping by one which is linear within each region and will choose the decomposition to adapt to the nonlinearities in the map. In most situations of interest, the mappings are very smooth, curving only slightly in most regions with occasional singularities or discontinuities scattered about the input space. We will adapt the mapping representation to have high resolution in regions of high curvature and low resolution elsewhere. We will present the representation algorithms in terms of a desired error ϵ. The smaller the desired error, the finer the required decomposition of the input space. We will estimate how many decomposition regions are needed as a function of the desired error.

In a complete intelligent system, the mapping units may only need to give a very crude approximation to the actual mapping. Different mapping modules will often be used in conjunction with each other and they will often operate on partially redundant data. The ensemble response of a collection of units can be much more accurate than any individual unit. In another common circumstance, the mapping units will be embedded in feedback loops. As long as their outputs predict roughly the correct direction for a process to proceed, the feedback will vary the input so as to eventually cause the desired response.

It is worth contemplating the maps we must presumably use in even so simple a task as picking up an object. We probably have a variety of components in our visual systems which together map the visual input associated with the object into our internal model of its position in three-dimensional space. This map is likely formed by combining mappings based on a variety of visual cues such as stereopsis, size information, shading information, information from the relative image motions under small head movements, depth of focus of the eyes and relative angle of the eyes. The mapping is not precise, as is indicated by our inability to exactly estimate the distance to the object. Based on proprioceptive information we map the sensed joint angles and muscle extensions of our arm into its position in our internal model of the three-dimensional world. The desired trajectory and final position of the arm must then be computed and mapped into muscle control coordinates. An estimate of how accurate this map is without the benefit of visual feedback may be obtained by reaching with closed eyes. With visual feedback we correct for inaccuracies in the muscle to space map. Once we come into contact with the object, tactile information is also mapped into the space where the goal state is computed.

Vision-based Robotics. We are currently using the techniques described in this section to develop a vision-based robot manipulator system that learns the visual impact of its actions. A video camera produces an image of the robot arm from which the basic parameters describing the visual position and angle of each limb are extracted using the image processing techniques described in section 7. The system also controls the joint angles of the arm. Using the algorithms in this section, it will learn the mapping between the joint angles and the parameters of the arm in the image. In this way it will be able to use visual feedback in tasks such as picking up objects and following moving objects at a specified distance.

5.1 One-dimensional mappings

Let us begin by understanding mappings from a one-dimensional space into another one-dimensional space. The one-dimensional problem arises in any higher dimensional context when all but a single degree of freedom is constrained. Some examples of one-dimensional maps that might be important to learn include the relationships between: the angle of an arm joint and the distance to its tip, the observed intensity of a patch of surface and the angle between its surface normal and the direction to a light source, the response of a tactile sensor and the force applied to it, the Doppler pitch shift of a sound source and the rate at which the source approaches, the divergence of the optic flow in an image as an object approaches (i.e. the rate at which the size of objects grows as they approach) and the time to impact, and the rate of air flow needed to produce a given pitch.

Choose coordinates in the domain so that the inputs lie in the unit interval $[0,1]$. Let the nonlinear mapping from $[0,1]$ to R^1 be denoted by f and the value at a point $x \in [0,1]$ by $f(x)$. We wish to construct an approximation to f which has an error of less than ϵ with high probability. We will denote the approximate function by $\tilde{f}$. We will choose the approximation to be equal to f at N "knot" points: $x_1 = 0, x_2, \ldots, x_{N-1}, x_N = 1$. $\tilde{f}$ will be defined between these points by linear interpolation. If $x_i \leq x \leq x_{i+1}$, then

$$\tilde{f}(x) \equiv f(x_i) + (x - x_i)\frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}. \tag{5.1}$$

We will choose N and the points x_i so as to keep the error below ϵ. If possible, we would like to keep N as small as possible consistent with this requirement so as to minimize the storage space and search time required for the data structure.

Any of the data structures discussed in section 3.1 may be used to efficiently implement evaluation of the approximation $\tilde{f}$. Given a point $x \in [0,1]$ we would like to quickly find the subinterval $[x_i, x_{i+1}]$ in which it lies. To build a static structure representing $\tilde{f}$ using comparison based searching, we could build a balanced binary tree with the x_i in sorted order as leaves. We may either directly store the value of f at each point with the corresponding leaf and perform the interpolation computation on each retrieval (equation 5.1) or we may store the coefficients

$$a = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} \tag{5.2}$$

and

$$b = f(x_i) - \frac{x_i}{x_{i+1} - x_i}(f(x_{i+1}) - f(x_i)) \tag{5.3}$$

along with the left endpoint of each interval so that we need only compute

$$\tilde{f}(x) = ax + b \tag{5.4}$$

to complete an evaluation. As was discussed in section 3.1, the time to find the correct interval is $O(\log_2 N)$. Because the interpolation takes time of $O(1)$, the cost to evaluate $\tilde{f}$ using this technique is $O(\log_2 N)$. We might also use the address computation structures described in section 3.1 to find the correct interval in constant time. In this case the data structure might require us to use suboptimal intervals and so require more of them.

5.2 The required number of intervals in one dimension

Let us now discuss the number N of intervals required to keep the error below ϵ. Most of the mappings that arise in practice have a slowly varying slope. For example, one typical source of nonlinearity is the trigonometric relationship between lengths and angles. This shows up for example in relating joint positions and angles or object size and orientation. In examples of this type a sinusoidal relationship typically causes the slope to vary only from 1 to -1 over the entire range of the angular variable. In many other examples, the nonlinearity arises only as a small correction to a linear map. In these examples as well as many others, the slope varies slowly over the range of the variables. We may take account of this by assuming that the rate of change of the slope, i.e. the second derivative of the map, is bounded by a constant C. We will compute a number of desired quantities in terms of this constant.

What is the maximum error in approximating a function f which satisfies

$$\left|\frac{d^2 f}{dx^2}\right| \leq C \tag{5.5}$$

by linearly interpolating its values at the two points $x = x_1$ and $x = x_2$? The error $e(x)$ at each point x is given by:

$$e(x) = \left| f(x) - f(x_1) - (x - x_1)\frac{f(x_2) - f(x_1)}{x_2 - x_1} \right|. \tag{5.6}$$

We would like to determine the maximum of e over all points x in the interval for that f for which it is largest. It is clear that we can get larger errors if e is entirely of one sign over the interval. Let us take it to be positive. $e(x)$ then satisfies the following:

$$e(x_1) = 0, \tag{5.7}$$

$$e(x_2) = 0, \tag{5.8}$$

and

$$\left|\frac{d^2 e}{dx^2}\right| = \left|\frac{d^2 f}{dx^2}\right| \leq C. \tag{5.9}$$

Let us show that the function with the largest error has constant second derivative $-C$ over the range. Consider the slope of e. This must vanish at the maximum of e and can increase at a rate no greater than C as we approach the endpoints of the interval. The height of the peak in e is the integral of the slope over this range. Thus we must maximize the area under a curve of bounded slope. This maximum is achieved when the slope equals the bound. Therefore the worst possible error is given by the quadratic function:

$$e(x) = -\frac{C}{2}x^2 + \frac{C}{2}(x_1 + x_2)x - \frac{Cx_1x_2}{2}. \tag{5.10}$$

The largest error occurs at the midpoint:

$$x = \frac{x_1 + x_2}{2} \tag{5.11}$$

and has the value

$$\frac{C}{8}(x_2 - x_1)^2. \tag{5.12}$$

For this error to be less than the desired ϵ, we must have:

$$|x_2 - x_1| < \sqrt{\frac{8\epsilon}{C}}. \tag{5.13}$$

This then tells us that the largest number of intervals we could possibly need is:

$$N = \frac{1}{|x_2 - x_1|} = \sqrt{\frac{C}{8\epsilon}}. \tag{5.14}$$

This grows more slowly than the inverse of the error as $\epsilon \to 0$.

We may obtain a better estimate for the required number of points if we look in the asymptotic limit as $\epsilon \to 0$. In this case, the spacing between neighboring points shrinks. Eventually, the samples become so dense that the second derivative is approximately constant between any two points. In this case the argument above gives the spacing, where at the point x we take C to be f's second derivative at the point x. The better estimate is:

$$N(\epsilon) = \frac{1}{\sqrt{8\pi\epsilon}} \int_0^1 \sqrt{\left|\frac{d^2 f}{dx^2}\right|} dx. \tag{5.15}$$

This holds even when the second derivative is not restricted to be less than some fixed bound. The asymptotics are only valid, however, for choices of ϵ which get smaller and smaller for larger and larger second derivatives.

It is easy to see that isolated discontinuities in the function or its derivatives do not change the required number of knot points asymptotically. We need only include two nearby points in the case of a discontinuity and only the corner point in the case of a discontinuous change in slope.

5.3 Static methods for choosing 1-D knot points

There are several strategies for building a data structure to represent a desired mapping. In the best of situations, a module may request the correct output for arbitrary inputs. This occurs, for example, when a system is attempting to learn a mapping which describes the sensory result of a motor action. By trying out the action, the system may sense the result. Another useful instance arises when there is a complex analytical description of a desired mapping which is too time consuming to evaluate during actual running. We might want to replace the analytic function by an approximate mapping which can be evaluated very quickly. In this "static" case, the system would be allowed a learning phase during which it evaluates the analytical expression at arbitrary points. Later in the section we will investigate the dynamic situation in which the system must make use of a given random sequence of input-output pairs.

In the static case we might hope to approach the optimal number of knot points derived in section 5.2. We may, in fact, achieve this value asymptotically for small ϵ by making a single linear sweep over the interval, determining successive knot points in order. For small ϵ the function is well approximated over an interval by the first three terms in its Taylor series approximation taken at the center of the interval. Let us denote the center of the interval of interest by:

$$\bar{x} = \frac{x_i + x_{i+1}}{2}. \tag{5.16}$$

For $x_i \leq x \leq x_{i+1}$, f is asymptotically well approximated by:

$$f(x) \approx f(\bar{x}) + \left.\frac{df}{dx}\right|_{x=\bar{x}} x + \frac{1}{2}\left.\frac{d^2 f}{dx^2}\right|_{x=\bar{x}} x^2. \tag{5.17}$$

As we have seen, when we linearly interpolate this approximation between the values at the endpoints, the worst error occurs at the midpoint $\bar{x}$. One strategy is to probe for the next knot point x_{i+1} by attempting to make the following function vanish:

$$f(\bar{x}) - f(x_i) - (x - x_i)\frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} - \epsilon. \tag{5.18}$$

To the extent that the approximation is valid, this will give the largest legal interval starting at x_0. There are a variety of techniques for finding zeros of functions numerically [106]. The basic idea is to use binary search or interpolation search to hop from side to side of the zero in ever shrinking jumps until it is approximated sufficiently accurately. We may allow for the inaccuracies in this process by searching for an ϵ slightly less than the desired one. An approach which generalizes well to higher dimensions is to successively halve the intervals until the approximation is good enough.

Sometimes we will be interested not in finding an approximation to within a fixed error but rather in finding the best approximation possible with given resources (such as memory space). This is probably the task that neural systems face, where a fixed number of neurons must be used to accomplish tasks as well as possible. As we saw above, given N points we should distribute them according to the distribution:

$$\nu\rho(x) = \nu\sqrt{\left|\frac{d^2 f}{dx^2}\right|}, \tag{5.19}$$

where ν is chosen to satisfy the normalization constraint:

$$N = \nu \int_0^1 \rho(x)dx. \tag{5.20}$$

We may apply the technique which was introduced in the context of probability distributions in section 3.4 to this situation. The idea is to estimate $\rho(x)$ using an input sample and to base the partition of the interval on this estimate. There are a variety of numerical techniques which may be used to estimate $\rho(x)$ using values of the function at points in a neighborhood of the point where the estimate is desired.

The simplest approach to choosing knot points would begin by estimating $\rho(x)$ at m points y_i evenly spaced in the interval. We would then compute R, the sum of ρ in all intervals:

$$R = \sum_{i=1}^{m} \rho(y_i). \tag{5.21}$$

We would then parcel out knot points to intervals according to the density of ρ that they contain. Let us denote the number of points assigned to interval i by N_i. The following expression gives each successive N_i in terms of the previous ones:

$$N_i = \left\lfloor \frac{\sum_{j=1}^{i} \rho(y_i)}{R} - \sum_{j=1}^{i-1} N_j \right\rfloor . \tag{5.22}$$

The N_i may be uniformly distributed in their intervals to give the knot points.

As we saw in section 3.3, we may do better by estimating $\rho(x)$ with more points in regions where it is larger. We may accomplish this by recursively decomposing the interval into halves which are expected to have an equal number of knot points. The estimation and decomposition procedure is repeated until there are N intervals. Because the same number of samples is taken from each of the segments of the partition at a given level of recursion, smaller segments (which have a higher density of ρ) will have a larger sample density than large segments.

Incremental methods. A module will often have to form the data structure representing a mapping without having control over the input-output pairs it is presented with. In this case the structure must be built incrementally as the points arrive. The most straightforward approach would simply store each input point in the data structure along with the corresponding output. When an input request arrives, the two closest stored inputs would be retrieved and their corresponding outputs used to interpolate. Any of the dynamic data structures suggested in section 3.1 could be used to maintain the points. If the inputs are random, a simple binary tree is probably sufficient in practice, but it may need occasionally to be rebuilt to maintain balance.

If the inputs are randomly chosen from a uniform distribution then we can estimate the number of inputs needed to achieve a certain accuracy. In equation (5.14) we gave the maximum number N of uniform sized intervals that could be required to achieve a desired accuracy of $\sqrt{C/8\epsilon}$. In section 3.3 we showed that the average number of samples needed to choose one from each of N equally likely bins is less than $N(\ln N + 1)$. The average number of samples needed to obtain an error bound of ϵ is therefore less than

$$\sqrt{\frac{C}{8\epsilon}} \left(\frac{1}{2} \ln \frac{C}{8\epsilon} + 1 \right) . \tag{5.23}$$

A technique to adaptively store more points in regions where they are needed is to first test each input-output pair to see if the current structure predicts the output to within the desired error limit. The pair is added to the data structure only if the current structure does not predict it sufficiently well. Faster convergence may be obtained if the error requirement a pair must satisfy to avoid inclusion is more stringent than the desired error requirement. A good choice is to require prediction to within $\epsilon/2$ for exclusion. For mappings well approximated by linear ones, this technique can drastically cut down on the number of stored points.

When the sample distribution is non-uniform, we may wish to know how many samples are required to get prediction within ϵ with high probability. We saw in section 3.3, that if we draw $2h(N + \ln h)$ points, then of the N bins, the probability the missed bins have a combined probability greater than h^{-1} is less than h^{-1}. If we want the error to be less than ϵ on all but a part of the space which has probability less than ϵ with a probability greater than $1 - \epsilon$, we need only use

$$\sqrt{\frac{C}{2}}\epsilon^{-3/2} - \frac{2}{\epsilon}\ln\epsilon \tag{5.24}$$

samples. Note that this is typically larger than the average number required when the distribution is uniform.

5.4 Multi-dimensional mappings

The k-dimensional situation is very similar to the one-dimensional one. We will approximate a smooth nonlinear mapping from one multi-dimensional space to another as a piecewise linear mapping. The input space will be decomposed into regions on which the approximation is linear. We saw in section 4 that the n nearest neighbors of a query point in a stored set of N points could be found in average time $\log_2 N + n$ by using k-d trees. This result suggests the first approach. We store a collection of input-output pairs using a k-d tree. To process a query point, we begin by finding its $k+1$ nearest neighbors. The approximate output value at the query point is given by linear interpolation between the output values at these $k+1$ nearest neighbors. Linear interpolation on a k dimensional space is described by a constant offset and $k \times k$ matrix for each component of the output vector. The interpolant may be computed from the stored values by standard techniques (e.g. reference [106]). We may either explicitly compute the interpolation coefficients for each region at query time or precompute them and store them along with the sample points. An intermediate choice is to initially compute interpolants when queries demand them, but to cache the computed results. For large k the number of regions may make this storage prohibitively expensive.

Number of points. The analysis of the required number of stored data points is quite analogous to the one-dimensional case. The $k+1$ nearest points to a query typically define a simplex (i.e. a generalized tetrahedron) surrounding the point. As in the one-dimensional case, given a bound on the second derivatives, we would like to estimate how many stored points are needed to achieve an error bound of ϵ. Let us again denote by C the bound on the second derivative of the output vector when computed along any straight line in the input space. For real-valued outputs, C will be the largest eigenvalue of the Hessian matrix of the mapping. For a given choice of $k+1$ points, the error $e(x)$ representing the difference between the original mapping and the linear approximation is a function with the same second derivative bound C which vanishes at the points in question. The error is a distance in the output space, the constant C is the ratio of a distance in the output space to the square of a distance in the input space, and the scale d of the stored points is a distance in the input space. Dimensional analysis immediately tells us that the scale d needed must be of order $\sqrt{\epsilon/C}$. The maximum number of stored points needed in k dimensions is therefore $N \sim 1/d^k \sim (C/\epsilon)^{k/2}$.

To work out a specific example, let us assume that the stored points lie on the vertices of a uniform grid with linear spacing d. In this case, the sets of points with the same $k+1$ nearest neighbors (called the faces of the kth order Voronoi diagram [121]) within a grid cube are the 2^k subcubes of side $d/2$ made by partitioning the grid cube in half along each dimension. The function with the maximum error inside one of these subcubes has this maximum at the center of the grid cube. We may bound the error by first bounding the value at the center of the $k-1$ simplex determined by taking all but the closest vertex. In this simplex, the maximum has zero first derivative and so we may use the same method as was used above for the one-dimensional case. Once we compute the maximum at the center of the simplex, we can bound the growth along the diagonal of the cube starting at the closest corner, passing through the center of the simplex and finally reaching the maximum value at the center of the cube. Carrying out these calculations gives the bound

$$d \leq \sqrt{\frac{8\epsilon}{C}|3k-4|}. \qquad (5.25)$$

Notice that when $k = 1$ we get the one-dimensional bound. In the unit cube there are $N = 1/d^k \sim \epsilon^{-k/2}$ of these grid cubes as ϵ vanishes. Making estimates exactly as in the one-dimensional case shows that in an average of order

$$-\epsilon^{-k/2}\ln\epsilon \qquad (5.26)$$

samples, we will have drawn one from each grid cube. For nonuniform distributions we may bound the probability of having this density of samples as above.

Cheaper versions. A coarser, but cheaper, alternative to linearly interpolating between the $k+1$ nearest neighbors is to use the same linear approximation throughout an entire leaf region of the k-d tree. If the leaves contain more than $k+1$ points, then the linear approximation may be chosen to be a least squares fit. In this case, the matrices and offsets could naturally be stored in the leaf. An approximation error could be maintained and when it exceeds a preset threshold, the leaf could be split. We may either split leaves by cutting across the direction in which the estimated second derivative is largest, or, as in section 4, we may split the leaf along the longest dimension, to yield roughly cubical leaves. As in the one-dimensional case, we may assume that the sample points are all presented at once, and we may build an approximately optimal tree by recursive splitting or incrementally adding samples, including them only when they are not already predicted sufficiently well.

An even cheaper version would just assign to a point the same output as its nearest neighbor. A more refined version would assign the average (perhaps weighted by a function of distance) of the n nearest neighbors. Since the kth order Voronoi diagram typically has many more regions than sample points, we get a finer approximation to the function without the expense of interpolation.

Learning the graph. There is another approach to learning, representing, and evaluating mappings. In mathematics it is common to replace a mapping from I to O by its *graph*, a surface in the product space $I \times O$. This product space has a dimension which is the sum of the dimensions of I and O. Each of its points represents a pair of points in the original spaces, one from I and one from O. An input-output training pair is thus a point in $I \times O$. We may store the training points by decomposing the whole product space $I \times O$ with a k-d tree. To find the output corresponding to a new input, we use the mixed nearest neighbor and range search technique discussed in section 4.3. This version is capable of learning relationships which are not mappings, and allow one input to correspond to several outputs.

5.5 Applications of mapping modules

There are a variety of important applications for piecewise linear mapping modules. These include learning geometric relationships, projecting out unwanted information, combining data from different sources, and learning to predict spatial and temporal sequences.

Object and state recognition. Many recognition tasks must deal with objects that have different states. Sometimes one wants to identify a specific object independent of its state and sometimes one wants to identify the state independent of the specific object. For example, speech recognition tries to classify words independent of the speaker, while speaker recognition tries to classify the speaker independent of the speech. Similar dichotomies are evident in problems such as handwriting recognition *versus* handwriter recognition, face identification *versus* facial expression identification, determination of illumination *versus* determination of surface reflectivity, identification of textures independent of viewing geometry *versus* using identification of the geometry from the texture variation, identification of object color *versus* illumination color, etc. An important class of smooth nonlinear mappings projects out one of these components to give the other. Often what is thrown away in one of the projections is exactly what is kept in the other. For example, in speech recognition one might want to do time warping and pitch renormalization to convert a spoken word into a more speaker independent form. The parameters of this transformation may be excellent clues as to the identity of the speaker. Similarly, one might translate, rotate and scale the image of an object to put it into a canonical form for recognition. The parameters of this transformation identify the geometric placement of the object. The mappings which perform these projections are a very important class and the algorithms in this section could be used to learn them. We are currently building a system based on these modules which both learns to discriminate textures and to extract geometric information from the variation of a single texture [57].

Sequence prediction. A vitally important task for biological organisms is the prediction of spatial and temporal patterns. Much behavior consists of performing appropriate actions after perceiving sequences of events. This includes communicative functions such as speech and birdsong and locomotive functions such as walking and swimming. Similarly, much of the activity of science consists of forming models to make predictions about future events on the basis of observed data. Reference [32] will report on independent work which shows that piecewise linear maps can be used in prediction of physical systems. The nonlinear mapping modules discussed in this section can learn to perform prediction if they are presented with several successive states in a sequence as their inputs. If a sequence is predictable on the basis of the presented number of states, then the simple learning algorithms discussed above will produce a piecewise linear predictor for it. If there are several "branches" which a sequence may take, depending on information not available in the observed previous states, then we may use the graph learning technique to obtain a complete list of likely predictions in any state. This type of system can in particular be used to represent and learn the behavior of deterministic and non-deterministic finite state machines. It would be interesting to investigate whether this kind of mechanism is sufficient for grammar acquisition.

Information combination. A final application for mapping modules is learning to combine multiple sources of information optimally. Much of the perceptual information used in machine vision, for example, is intrinsically ambiguous when taken alone. It appears that the ambiguity in one modality may be resolved by information from other modalities. For example, there are a large number of visual cues for extracting object shape including shading, contour, texture, surface contour, motion, and stereopsis. A fundamentally important task is to combine these different kinds of information in computing an interpretation of an image. Current attempts at doing this tend to simply use rather *ad hoc* weighted voting schemes, but one would really like a system to *learn* the appropriate response to different combinations of inputs. The mapping modules described in this section appear ideal for this task.

6. Undirected discrete modules: category formation

In this section we discuss undirected algorithms which produce a discrete set O of outputs. Let us denote the number of non-null outputs a module can produce by $|O|$. The set of inputs which produce a given output may be thought of as a class or a category. These modules must create categories on the basis of the observed statistics of their inputs. This type of task is often called *clustering.* We would like the regions corresponding to each category to be roughly cubical and to divide up the input probability distribution evenly. We would like enough outputs so that the probability density in each region is approximately uniform. These are almost exactly the goals for the *k-d* tree leaves described in section 3.4. Thus the process of building a *k-d* tree gives a natural clustering algorithm if we consider the points in the leaf regions as clusters.

Shrink and split. In practice, we will often want to use the variant suggested in section 4 in which the leaf hyper-rectangles are shrunk to a size given by the criteria in section 3.3. The leaf regions will then demarcate the high probability portion of the input space. The fineness of the demarcation is determined by the number of samples and the depth of the *k-d* tree. In fact, the shrunken regions at each level of the *k-d* tree are a natural hierarchical clustering of the input space. We call this approach to hierarchical representation the *shrink and split* technique.

Image analysis. We have found the shrink and split technique to be useful in the analysis of images [97]. Using the partial summation technique discussed in section 3.2, partitioning and shrinking operations can be done in logarithmic time. We have built an optical character reader which finds the rectangular boxes surrounding characters by using this kind of structure. We are also building a system for understanding line drawings which uses the technique to obtain a hierarchical representation of a drawing.

Color maps. Reference [48] uses a similar technique to find an optimal color map for a color image. The colors in the image are histogrammed in the three-dimensional color space. A *k-d* tree is built over the color space based on the histogram, and color map representatives are chosen from the leaves. Use of the partial summation technique presented in section 3.2 could probably speed up the tree construction in this algorithm substantially. Reference [99] uses a similar technique to analyse histograms and to implement high dimensional Hough transforms.

Speech coding. The technique of vector quantization, which is widely used in speech coding, compression, and recognition [75] quantizes a time series by decomposing the space coordinatized by successive samples into regions which are labelled by codes. A powerful body of theory has been developed to analyse optimal quantization choices but it appears that algorithms based on *k-d* tree variants could speed up many procedures substantially. They could be used both to decompose the space into quantization regions and to code and decode entries.

Minimal spanning trees. There are several useful clustering techniques based on the Euclidean minimal spanning tree of the set of input points [143]. This is the tree whose straight line edges joining all the points in the set have a total length smaller than or equal to that of any other such tree. References [15,91] discuss the use of *k-d* trees to construct the Euclidean minimal spanning tree over N nodes in average time of order

$N \log_2 N$ independent of the dimension k. The algorithm is based on Prim's which considers edges joining pairs of points in order of increasing length. An edge is added to the partially constructed tree if it does not create a closed loop. The resulting tree joins all the points and is the minimal spanning tree. The *k-d* tree algorithms make use of the fast nearest neighbor ability to choose the next edge.

A natural way to cluster points in a Euclidean space depends on a real parameter. Given a parameter value, join all pairs of points which are closer to each other than this value, and consider the clusters to be the connected components of the resulting graph. A hierarchical structure is introduced as the parameter is varied from infinity down to zero. Clusters break into two when the parameter goes below the shortest edge joining the two halves. It is easy to see that the critical edges in this process are exactly the minimal spanning tree edges. Thus the hierarchical pattern of clusters can be determined directly from the minimal spanning tree by removing edges in order, from longest to shortest. In the next section we will discuss mappings which preserve the minimal spanning tree of a set of points.

7. Undirected continuous modules: self-organization

We come finally to modules whose behaviors are analogous to neural networks models which exhibit self organization. The outputs that these modules produce are continuous in character and the mapping that they implement is determined by the distribution of inputs. We will discuss algorithms for two particularly interesting behaviors. In the first kind, the output space has the same dimension as the input space and the mapping imposes a distortion that tends to make the probability density uniform. In the second kind, the output space has a lower dimension than the input space and the mapping attempts to preserve the neighborhood relationships of high-probability regions.

7.1 Probability equalization modules

There are a variety of applications for modules which equalize the probability of states in their output spaces. In systems composed of many modules, such modules adjust the internal representation of states so that most of the input space of later modules is focused on inputs that occur with sizable probability. This renormalization also makes the Euclidean distance metric used by later modules in nearest neighbor calculations act more like a conceptual distance than an arbitrary coordinate distance. A well-known feature of human psychology is that people are better able to discriminate between pairs of items, like faces or speech sounds, which have been repeatedly encountered than between those which have not. An equalizing mapping makes the probability of a region approximately equal to its volume. Information theory tells us that a process which localizes states to fixed volumes gets the maximal amount of information when the probability distribution is uniform.

Histogram equalization. In one dimension, the probability equalization operation is often called histogram equalization and is widely used in image processing. In this application, the appearance of a monochrome image can often be greatly improved by mapping its intensity values so that the number of pixels with each possible intensity is roughly equal. The standard technique begins by histogramming the intensities of the pixels. Dividing the total number of pixels by the number of intensities gives the number of pixels that should have each intensity in the final image. A single sequential pass through the histogram is made, partitioning the old intensities every time a bin's worth of pixels is passed. This process will join weakly-represented intensities into a single intensity and spread the pixels of strongly-represented ones among several intensities. Several heuristics have been introduced to split up a large number of pixels representing a single intensity value [103]. This procedure is particularly effective in lowering the intensity resolution of an image with as little degradation as possible.

There are several algorithmic approaches one might take to probability equalization in higher dimensions. We will present two that have the advantage of extreme algorithmic simplicity and efficiency. The first technique is based on the adaptive grid structure and is continuous. The second technique is based on *k-d* trees and does a better job of equalization but can be slightly discontinuous along some edges.

Grid equalization. If we have an adaptive grid, whose cells have been chosen so as to have approximately equal probability, we can achieve equalization by merely mapping the cell with label $(i_1, \ldots, i_d)$ to the cell with the same label in a uniform grid. The hyper-rectangular regions within a cell are linearly mapped into the hyper-cubical cells of the uniform grid. This mapping is clearly continuous and piecewise linear and it equalizes probability to the extent that the original grid regions were of equal probability. Notice that the mapping may be described by d one-dimensional equalization mappings; one applied to each of the coordinates. The density that one of these mappings equalizes is the total density projected to the relevant coordinate. The final density that is produced by this operation has uniform projections to each of the coordinate axes. Unfortunately, iterating the procedure does not improve the equalization performance of this technique.

***K-d* equalization.** A *k-d* tree may also be viewed as a *k-d* trie by ignoring the discrimination values stored at each node and only taking account of the stored dimension numbers. A *k-d* tree cuts a hyper-rectangle so that half the probability is on each side of the cut. A *k-d* trie cuts a hyper-rectangle so that half the volume is on each side of the cut. If we create a balanced tree, then each leaf of a *k-d* tree contains the same amount of probability, while each leaf of a *k-d* trie contains the same volume. The probability equalization mapping which we are proposing here sends the points in each leaf in the *k-d* tree to points in the corresponding leaf in the *k-d* trie with the same structure. The hyper-rectangles are mapped by linearly scaling each dimension appropriately. We have seen that *k-d* tree leaves are approximately cubical with a volume inversely proportional to the probability density they contain. If ρ varies smoothly, then neighboring *k-d* tree leaves will undergo approximately the same transformation and the mapping will be approximately continuous along the edge joining the leaves. In practice, to find the image of point under the mapping, we first search for it in the *k-d* tree. We need to produce the d components of the output point. At a particular node in the traversal of the *k-d* tree, if the dimension number is i, then whether we go left or right determines the next bit in the ith component of the output. When we finally reach the leaf, we need merely tack the binary expansions of the relative position of the point along each dimension of the leaf hyper-rectangle onto the ends of the partially specified output components. This operation is very fast, and does a good job of equalizing probability. The mapping produced is one-to-one and onto (assuming that the boundaries of the leaves are handled correctly) and quite close to continuous if there are enough leaves in the *k-d* tree.

7.2 Dimension reduction

There are many uses for a module which reduces the dimension of a space while preserving as many of the neighborhood relationships between high probability regions as possible. As we discussed in section 5.5, there is a need to combine data from different modules throughout intelligent systems. A natural approach is view the output of two modules as a single point in the product of the two output spaces. The dimension of this product space is the sum of the dimensions of the original two output spaces. To keep the input spaces from getting larger and larger in successive layers of modules, we need to reduce the dimension of the space while preserving the important relationships within it. A more mundane use for these algorithms is to collapse multi-dimensional data structures to two dimensions so that their contents may be viewed on a screen or page.

***K-d* tree approach.** A simple approach to dimension reduction uses a *k-d* trie decomposition of the input space. The idea is to identify some subset of the input dimensions with each output dimension. We reinterpret the *k-d* trie decomposition of I as a *k-d* trie decomposition of O by replacing each split dimension with the corresponding output dimension. To find the output coordinates of an input point, we traverse the tree, at each juncture adding bits to the corresponding output coordinate value. Finally, we project the input leaf region onto the output leaf region to get the low order bits of the output coordinates. This approach preserves much of the hierarchical and neighborhood structure determined by the original partition. Nested node regions in the input space are still nested and pairs formed by splitting a region in the input correspond to pairs formed by splitting the corresponding region in the output.

Category representation. Another use for dimension reduction arises in the categorization modules described in section 4. Modules which categorize their inputs as belonging to one of M disjoint subsets of the input space must choose an encoding for outputting the category. The most straightforward encoding simply labels the M classes with integers from one to M without regard to any structure possessed by the classes. A similar encoding would utilize M output lines, each one representing one class. These encodings ignore any relationships that exist among the classes. If there is a metric on the input space, we may define a notion of which classes are near to which other classes. If the encoding preserves aspects of these relationships, later modules can use them. One approach to implementing such an encoding chooses a representative point in each of the classes whose coordinates are output when a class is chosen. If these representatives are typically situated in their classes, the metric relationships between them will be representative of those between the classes. The disadvantage of this scheme is that it preserves the dimensionality of the input space. The dimension reduction technique presented above does an excellent job of reducing this dimensionality if the *k-d* trie is built so that each leaf contains only one classification point. It is of interest, though, to determine theoretical limits on the preservation of the metric relationships between a set of points under dimension reduction.

Two-dimensional representations. There are a variety of indications that two-dimensional encodings are particularly important in biological nervous systems. The major pathways from sensory systems to the cortex, from the cortex to motor systems, and from one cortical area to another are made up of parallel bundles of nerve fibers which tend to preserve topographic relationships from one end to the other. Such ordered projections are presumably easy to grow and to specify genetically. When such parallel bundles are used to transmit information from one region to another, the data is represented in the pattern of activity across its two-dimensional cross section. Because of the limited extent of the dendritic arborization of the neurons receiving input from this bundle, neighboring fibers will tend to activate similar responses. The metric structure of three dimensional space may well impose itself on the neighborhood relations between represented concepts. As we discussed in section 2, the neocortex itself is a two-dimensional sheet organized into two-dimensional areas.

The three-dimensional nature of the world also causes much of an animal's sensory input to be naturally two-dimensional. An animal has a two-dimensional bounding skin enclosing its body and the somatic sensory input is organized in terms of this two-dimensional topography. Visual input arrives along the two-dimensional sphere of rays centered at the lens of the eye and is focused on the two-dimensional sheet of photoreceptors making up the retina. This two dimensional topography is preserved (though distorted) in the surface of the lateral geniculate nucleus and the first few visual areas of cortex. A number of two-dimensional maps are superimposed in the superior colliculus. Auditory information about the direction of a sound source is two dimensional for the same reason that visual directions are. The different directions in which the eye can point are represented in a two dimensional map which is aligned with the auditory and visual maps.

Even in cases where there is no obvious geometric reason for it, information is often represented in two-dimensional form. In speech understanding, the vowels naturally decompose the two dimensional space coordinatized by the frequencies of the first two formants of a sound. Similarly, distinct colors naturally decompose a two-dimensional space (which is linearly independent of overall intensity). In the primary visual cortex, orientation appears to be mapped against occular dominance to form two dimensional patches (hypercolumns) which code for combinations of these two quantities.

Collapsing minimal spanning trees. It is intuitively clear that the neighborhood relations of a two-dimensional representation are much more able to represent complex relationships than those of a one-dimensional representation. The question naturally arises as to what extent higher dimensional relationships can be encoded in two dimensions. We saw in section 6 that the minimal spanning tree of a collection of points is useful in describing how the points are clustered. Given M points in n dimensional space, to what extent can we find corresponding M points in two dimensions, such that the spanning trees are the same? Points in n dimensions which are joined by a minimal spanning tree arc should correspond to points in two dimensions which are joined in the two-dimensional minimal spanning tree. We will show that any collection of M points in an n dimensional space, with a Euclidean minimal spanning tree whose vertices each have degree less than 6, may be projected onto M points in the plane such that the Euclidean minimal spanning trees are taken to one another.

We will prove this theorem by induction on the number of vertices M. The inductive statement is that an arbitrary tree whose vertices each have degree less than six can be embedded in the plane as a Euclidean minimal spanning tree in such a way that given positive real numbers d and ϵ and a specified vertex v of degree less than five, the vertex v is placed at the point $(d, 0)$, the entire tree lies inside a circle of radius ϵ centered at $(d, 0)$ and v is the closest point in the tree to the origin $(0, 0)$. If we can prove the inductive hypothesis, we will have proven the theorem by choosing the tree to be the desired minimal spanning tree in n dimensions. We place one of the leaves of the tree at the origin. v is chosen to be the parent of that leaf. Since the original tree had degree less than six, without the chosen leaf v has degree less than five. Inductively, the tree minus the leaf may be placed in a disc centered at (0,1) as a Euclidean minimal spanning tree in such a way that the origin is closer to v than any other vertex. By Prim's algorithm for minimal spanning tree construction [2] the whole Euclidean minimal spanning tree is obtained by adjoining the origin to v as desired. The base case for the induction is trivial. If the tree has only a single vertex, assign it to the point $(d, 0)$ and all conditions are satisfied.

To establish the induction, we must construct a planar embedding of the tree obeying the inductive conditions based only on the existence of such embeddings for strict subtrees. Without loss of generality we assume that the degree of the chosen vertex v is four. The lower degrees proceed in an identical manner but are simpler. If we view v as the root of the tree, it has four subtrees as children. We will inductively assume that each of these children can be embedded with parameters which are to be determined. In particular, we will force each of these five subtrees to lie entirely within its own disc of radius ϵ_2 which will be determined as we proceed. Each disc will be centered at a distance $\epsilon/2$ from v. As long as $\epsilon_2 < \epsilon/2$, the entire tree will then lie inside the desired radius ϵ disc. We must choose the angular placement of the subtrees and the bound ϵ_2 in such a way that each subtree is closer to v than to any of the other subtrees and that the origin is closer to v than to any of the subtrees.

Let us first ensure that no part of the discs containing the subtrees is closer to the origin than v (i.e. than the distance d). It would be easy to impose this requirement if we forced the subtree discs to lie to the right of the vertical line $x = d$. Unfortunately this does not leave enough room to make the discs farther from each other than from v. We will therefore allow the discs to extend an angle γ beyond the line $x = d$ both above and below v. We must choose γ so that no point of any disc is closer than v to the origin. We will first impose the requirement that $\epsilon_2 < \epsilon/4$. Then the subtree discs lie inside an annulus defined by the circles of radius $\epsilon/4$ and $3\epsilon/4$. The portion of this annulus, bounded by γ which is closest to the origin is on the inner radius (i.e. $\epsilon/4$) and at the angle γ from the vertical. The cosine rule shows that this point is a distance d from the origin (i.e. the same distance as v) when $\sin\gamma = \epsilon/8d$. Since $\sin\gamma < \gamma$ for positive angles, we may choose $\gamma = \epsilon/8d$.

We now choose the angular positions of the centers of the four discs. We need to leave room so that the entire outer two discs stay within γ of the vertical and the angle between the centers of the discs must be larger than $\pi/3$ (otherwise we cannot fulfill the distance requirement between discs). We choose to center the uppermost disc at an angle $3\epsilon/32d$ counterclockwise from vertical and the next three centers at successive clockwise angles of $\pi/3+\epsilon/16d$. This places the last disc also at an angle of $3\epsilon/32$ from vertical.

Finally, we must choose the radii ϵ_2 so that the two requirements are fulfilled. The outermost discs stay inside of the line defined by γ if $\epsilon_2 < (\epsilon/2)\sin(\epsilon/32d)$. We also need to ensure that the distance between the closest points in two discs is larger than the largest distance from a point in a disc to v. If the angle between disc centers had been $\pi/3$, they would have formed equilateral triangles with v and the distance between them would have been $\epsilon/2$. Since the actual angle is $\pi/3 + \epsilon/16d$, the distance is larger. Let us denote the amount by which it is larger by ξ. The closest points on two discs are then $\epsilon/2+\xi-2\epsilon_2$ apart. The largest distance from v to a disc point is $\epsilon/2+\epsilon_2$. We get the required relationship if $\epsilon_2 < \xi/3$. Thus by choosing ϵ_2 to be the smallest of the inequalities we have given for it, all requirements are satisfied. By Prim's minimal spanning tree algorithm, the minimal spanning tree of the point set we have defined will be isomorphic to the original given tree. This completes the induction.

It is easy to see that no graph with a vertex of degree 7 or two neighboring vertices of degree 6 can be properly embedded. Similar results hold for embedding minimal spanning trees in other dimensions. The requisite bound on degree is determined by the properties of sphere packing in the given dimension.

8. Parallel algorithms

One of the apparent advantages to performing computations with neural networks is their natural parallelism. Each network unit performs its computation based only on its current inputs and each unit can proceed independently of the others. The idea appears particularly attractive because the function performed by individual units is very simple. This suggests that it should be possible to inexpensively build large arrays of units. Unfortunately, the completely interconnected networks used in most models are not practical to build when the number of units is large. In this section we will show that there are parallel algorithms for architectures with large numbers of simple components which are capable of implementing the efficient algorithms we have described in previous sections. The speedup over a direct neural network implementation is almost proportional to the number of processors. It thus appears that for a given amount of hardware, one may do much better than direct emulation of neural networks.

There is a natural way to build a coarsely parallelized implementation of the systems built from efficient modules that we have been discussing. We simply assign a processor to each module or to a set of modules and the pattern of communication between processors is exactly the pattern of interconnection of the modules. A major problem with many parallel computers is that the amount of time spent communicating data between processors completely swamps the time spent doing useful computations. One advantage of modular networks is that the paths of communication are well defined and sparse, and there is a fairly large amount of work to be done within a module for each piece of data that is input or output.

To get a higher degree of parallelism, we must parallelize the function of a single module. We will show how to parallelize the basic operations we have discussed when there are $O(N)$ queries to be handled using $O(N)$ simple processors which together store a data structure with $O(N)$ pieces of data.

8.1 Routing, sorting, and parallel prefix

There are many different designs for massively parallel machines with a large number of simple processors connected by a fairly dense interconnection network [9,49,131,119,104]. There are three fundamental parallel operations that most of these machines can support well and which form useful primitives for designing and implementing parallel algorithms. The technique of "virtual processors" [129] allows individual processors to transparently simulate any number of processors. In this way, the three operations described here can be implemented to work with any number of elements.

Perhaps the most basic operation is *routing*, whereby a piece of data in one processor is transferred to another processor whose address is specified along with the data. A routing algorithm must resolve the inevitable collisions which occur when many pieces of data are simultaneously transferred. Most designs use a variant of the algorithms proposed in [134,70] which take logarithmic average time.

The second important parallel operation is sorting. One of the first algorithms for this task is the Batcher sort [9], which takes $\log_2^2 N$ time but is very efficient to implement on hypercube, shuffle-exchange and cube-connected cycles networks. More recent sorts take logarithmic time [22,108] and appear to be amenable to practical implementation.

The final operation has come to be known as the parallel prefix operation [67,66,119]. It takes an associative binary operation $\oplus$ and a vector of elements drawn from the domain of the operation, and produces the vector of all prefixes of the given vector combined using the given operation. Prefixing $\oplus$ over the vector (A, B, C, D, E) results in the vector $(A, A \oplus B, A \oplus B \oplus C, A \oplus B \oplus C \oplus D, A \oplus B \oplus C \oplus D \oplus E)$. By making different choices for $\oplus$, a variety of useful operations can be put into this form. There are several logarithmic time implementations of prefix and it can be implemented very efficiently on the standard interconnection networks. For example, on the Connection Machine computer, a parallel prefix operation using the underlying hypercube connections is as fast as a single random routing operation [129]. A simple general implementation uses $\log_2 N$ routing stages. At the tth stage the ith processor sends its current partial result forward to the $i + 2^{t-1}$th processor where it is combined using $\oplus$ with the partial result that is already there.

Useful choices for $\oplus$ include "minimum" or "maximum" to find the largest or smallest elements and "plus" to sum elements. Given a vector composed of ones and zeroes in which the ones mark the elements of a set, prefixing "plus" will enumerate the elements of the set. By first enumerating a set and then routing the set elements, treating the enumeration value as an address, we may pack the elements of a set into consecutive processors. If the binary operation is chosen to be $A \oplus B = A$, then parallel prefix will spread the value of the first element of a vector to all elements. If the elements have two components, the second of which is Boolean valued and treated as a marker, a similar operation can be constructed to spread values or combine them within the ranges of consecutive processors delineated by the markers.

8.2 Parallel module implementation

Let us now consider the parallel implementation of some of the data structures underlying the adaptive modules described in previous sections. We begin with a coarse decomposition of the data and the queries which is appropriate for a system with an intermediate number of fairly powerful processors. We then consider algorithms appropriate for a system with a large number of very simple processors.

If the input requests are distributed over the input space according to the same distribution as stored data, it is natural to assign a processor to each leaf bucket in a structure which has as many equiprobable leaf buckets as there are processors. Assume the incoming queries are distributed randomly to the processors. Each processor uses the addresses of the queries it receives to compute the address of the processor corresponding to the bin that the query should be assigned to. A routing operation can then send all the queries to their correct bins. Because the inputs are distributed according to the same distribution as the data, and the bins are equiprobable, each processor will receive approximately the same workload. Internally, the processors use serial algorithms to process the queries according to a further decomposition of the bin they represent. This technique is tunable to any degree of parallelism by choosing the number of bins represented by processors.

If the queries are not well described by the stored data distribution, then the above scheme can end up with a few processors handling all the work while others are idle. This is more and more likely to happen as the number of processors increases and the decomposition of the space becomes finer. Let us see how by combining the three fundamental operations, we can distribute the work over all the processors, no matter how the queries are distributed.

Let us begin with the one-dimensional case. Assume that the input space is an interval which is broken into N bins. These are represented by "bin descriptors" which contain the left coordinate of the bin and the value which should be returned when an input key falls into the bin. The technique described above would assign a bin to each processor along with its value. When there are many queries that fall into a bin, the corresponding processor becomes a serial bottleneck. The technique we use here begins by sorting the bin descriptors by coordinate together with the key values of the queries. If we assume that descriptors are less than queries with the same key value, then we end up with all queries that fall in a given bin residing in consecutive processors headed by the bin descriptor. We make up a vector which contains the value to be returned in those processors which contain a bin descriptor and a special value "null" in those processors which contain a query. The parallel prefix operation which sets $A \oplus B$ to A if B is "null" and to B otherwise, will spread the return values from each bin descriptor to all query elements which lie in that bin. If query elements have retained their initial addresses, then a routing operation returns them there along with the correct value to return. Notice that as many processors are devoted to a bin as there are queries which fall into that bin (plus one for the bin descriptor). If all queries fall into a single bin, almost all processor resources are devoted to that bin.

Almost the same technique works when we use an adaptive grid to decompose a multi-dimensional space. We again use a descriptor for each grid element. The first step sorts these together with the queries on the basis of the first component of the key. This will split the processors into segments corresponding to the decomposition of the first coordinate. Each segment begins with descriptors for all the grid elements with a given first coordinate. In the next stage, we would like to sort elements on their second coordinate separately within each segment. This partitions the processors even more finely, on the basis of the first two coordinates. Proceeding in this manner, we end up with each bin in the grid represented by a consecutive set of processors beginning with the descriptor and followed by all queries which lie in the bin. We again use parallel prefix to spread the return value to the queries and a routing cycle to return them to their original processors.

We must now describe how the segments are sorted within themselves. The processor at the left end of a segment can determine this by checking that it contains a descriptor and its left neighbor either contains a descriptor with a different first coordinate or a query value. We can then use the prefix technique of spreading values between markers to give each element in a segment the value of the left end of that segment. Prepending this value to the front before sorting the entire machine on the second coordinate will have the effect of sorting within segments. Shorter quantities to prepend can be obtained by using a "plus" prefix to enumerate the heads of the segments, and then spreading this segment number throughout the segment to be prepended at the next stage.

Finally, let us give the analogous algorithm for searching for query points in a *k-d* tree. The only added ingredient in this case over the grid case is that different segments must be sorted on different coordinates according to the dimension number specified in the descriptor at the front of the segment. A descriptor is formed for each node of the *k-d* tree and is given coordinate values equal to the those of the point with minimal coordinates contained in the hyperrectangle corresponding to the node. In sorting, nodes higher in the tree are treated as smaller than lower nodes with equal coordinates. On the ith pass, the queries will be segmented according to the bins determined by the nodes of the *k-d* tree at the ith level. Used up parent nodes will end up at the front of segments and are not included in any further segment computations. During each stage, the dimension number is spread from the descriptor at the front of each segment throughout the segment. The segments are enumerated and the segment number is spread throughout the segment and prepended to the coordinate of the correct dimension. A sort is performed to give the segmentation corresponding to the next level of the *k-d* tree. When the bottom of the tree is reached, each leaf descriptor spreads the desired return value to the query points in the same segment. These may then be routed back to their initial positions.

Most of the operations we have discussed in this paper may be parallelized in a similar way. Because we sort at each level, the time to perform the search on an $O(\log_2 N)$ depth *k-d* tree is $O((\log_2 N)^2)$. In this time we satisfy $O(N)$ queries on an $O(N)$ processor machine. In the best of circumstances, a parallel implementation of an $O(N)$ unit neural network with $O(\log_2 N)$ connections per neuron on an $O(N)$ processor machine would take time $O(\log_2 N)$ to process a *single* query. The neural network processors must do a fair amount of floating point arithmetic. The operations we have used are simply and efficiently implemented on the one bit wide Boolean processors used in the Connection Machine [49]. Even assuming that the time per operation is equivalent, we obtain a speed-up of $O(N/\log_2 N)$ per query by using the algorithmic approach. In future machines with 10^8 processors, this represents a speed-up of over a million.

9. Conclusions

Let us compare the performance of the *k-d* tree structures of section 3.2 with the learning matrix neural network discussed in section 2.5 on the task of associative storage and retrieval of one million entries, each specified by a few real coordinates. We motivated this number in section 2.5 as giving the order of magnitude for human performance. To build a *k-d* tree with 10 entries per bucket, will take about 17 stages in which each of the 10^6 items is examined, or about 10^8 operations. To access a given element takes 17 comparisons and a final search through 10 elements. The nearest neighbor of a given element is likely to be in the same bucket or a neighboring one, and so will only take a few times longer to find.

To do associative retrieval with a learning matrix, we must first expand the dimension of the representation of each data item using a randomizing function so that the memories are likely to be linearly independent. As we discussed in section 2.5, [77] showed that a learning matrix requires a number of units equal to 1.45 times the number of memories times the product of the average number of set bits in an input and an output. References [58,4] show that similar networks require about seven times as many completely interconnected units as there are memories. Let us nonetheless underestimate the required number of totally interconnected neurons to store a million memories to be one million. As discussed in section 2.5 such a network will require on the order of 10^{12} operations to produce an output every time it is presented with an input. This is to be compared to the few hundred steps needed by the *k-d* algorithm. In use, we would expect the algorithmic approach to beat the simulation approach by a factor of at least a billion. Even if the network is much less than totally interconnected, one would expect a speed up of over a million.

About 10^{12} operations are also required to update the weights according to a given input and it will also require about 10^{12} memory locations to store these weights. As we discussed in section 2.4, learning even only a few items in typical neural networks takes thousands of presentations of each input. Reference [128] shows that in at least one example of back propagation the learning time goes as the 4/3 power of the number of inputs. Let us underestimate the required number of presentations of each memory to be of order a thousand. Estimating 10^3 presentations of each of the 10^6 memories, and 10^{12} operations to update the weights for each presentation, we obtain a total learning time of 10^{21} steps. This is 10^{13} times longer than the *k-d* tree algorithm's learning time.

The parallel algorithms which we presented in section 8 speed up the serial algorithms by a factor of $N/\log_2 N$ by using N processors (though more sophisticated algorithms may be able to achieve a factor of N speedup). Parallel hardware to simulate a neural network can at best achieve the optimal speed up of a factor of N with N processors. On a parallel machine with one million processors, the above comparison is therefore only weakened by a factor of $\log_2 10^6 = 20$. Parallel digital hardware to simulate a neural network could be sped up by a factor of ten million if it were used to implement one of the parallel algorithms instead. Notice that we have not even taken into account the fact that the floating point network simulation is likely to require more hardware per processor than the simple operations used in the algorithmic approach.

Why are the algorithms so much faster than the networks? Both of them work by effectively partitioning up the input space and causing desired outputs to be produced on inputs in each partition. The neural networks must evaluate the activity of every neuron and must consider the effect of every weight each time they are presented with an input. The algorithmic approach only looks at those stored values along a path of logarithmic depth. In learning, the networks update each and every weight on every input. The data structures only modify the parameters of those regions which are relevant in determining the output on the given input.

This same reasoning about performance appears to apply equally well to modules implementing behaviors from each of the four categories we have discussed. For behaviors more complex than associative memory, however, it is not as clear exactly what functions neural networks are actually computing. It would be of great interest to compare both the computational and cognitive performance of neural network implementations of systems like NETtalk [120] with their algorithmic counterparts. Even if digital simulations of neural networks are not as cost-effective as the corresponding algorithmic systems, their study is important both for the understanding of biological systems and possibly for future technologies based on components with different tradeoffs than current digital ones. If an inexpensive analog technology is found that can directly implement the neural network operations, then this argument might be significantly weakened.

Let us conclude with a somewhat speculative estimate how far current computers are from the human brain under the assumption that the algorithms used by the brain can also be sped up by the factor 10^9. Assume there are an average of 10^3 synapses per neuron and that there are 10^{12} neurons, each with an average firing rate of 100 spikes per second. Let us further assume that the effective computation done at a synapse during each potential firing period is equivalent to a 32 bit operation on a modern computer. The brain's computation rate is then equivalent to 10^{17} instructions per second. With a 10^9 algorithmic speedup, we would only need a 100 MIP (million instructions per second) machine. This level of performance should become quite common in the next few years.

10. Acknowledgements

I would like to thank Doyne Farmer, Bartlett Mel, Norman Packard, Rob Shaw, Gerry Tesauro, and Stephen Wolfram for many stimulating discussions about these ideas, David Ballman for a feat of digital archeology which rescued these bytes from the ruins of a file system. This work was supported in part by the National Center for Supercomputer Applications.

References

[1] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, "A Learning Algorithm for Boltzmann Machines," *Cognitive Science*, **9:1** (1985) 147–169.

[2] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, (Addison-Wesley, 1974).

[3] Alfred V. Aho and Jeffrey D. Ullman, "Optimal Partial-Match Retrieval When Fields Are Independently Specified," *ACM Trans. on Database Systems*, **4:2** (1979) 168–179.

[4] Daniel J. Amit, Hanoch Gutfreund, and H. Sompolinsky, "Storing Infinite Numbers of Patterns in a Spin-Glass Model of Neural Networks," *Physical Review Letters* **55:14** (1985) 1530–1533.

[5] James A. Anderson, "Networks for fun and profit," *Nature*, **322** (1986) 406–407.

[6] Dana Angluin and Carl H. Smith, "Inductive Inference: Theory and Methods," *Computing Surveys*, **15:3** (1983) 237–269.

[7] D. H. Ballard, G. E. Hinton, and T. J. Sejnowski, "Parallel visual computation," *Nature*, **306** (1983) 21–26.

[8] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson, "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems," *IEEE Transactions on Systems, Man, and Cybernetics*, **13:5** (1983) 834–846.

[9] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the AFIPS Spring Joint Computer Conference* **32** (1968) 307–314.

[10] Jon Louis Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM*, **18:9** (1975) 509–517.

[11] Jon L. Bentley, "Multidimensional Binary Search Trees in Database Applications," *IEEE Transactions on Software Engineering*, **SE-5:4** (1979) 333–340.

[12] Jon Louis Bentley, "Multidimensional Divide-and-Conquer," *Communications of the ACM*, **23:4** (1980) 214–229.

[13] J. L. Bentley and W. A. Burkhard, "Heuristics for partial-match retrieval in database design," *Information Processing Letters* **4:5** (1976) 132–135.

[14] Jon Louis Bentley and Jerome H. Friedman, "Data Structures for Range Searching," *Computing Surveys*, **11:4** (1979) 397–409.

[15] Jon Louis Bentley and Jerome H. Friedman, "Fast Algorithms for Constructing Minimal Spanning Trees in Coordinate Spaces," *IEEE Transactions on Computers* **C-27:2** (1978) 97–105.

[16] Joh Louis Bentley and Hermann A. Maurer, "A Note on Euclidean Near Neighbor Searching in the Plane," *Information Processing Letters* **8:3** (1979) 133–136.

[17] Jon L. Bentley, Donald F. Stanat, and E. Hollins Williams, Jr., "The Complexity of Finding Fixed-radius Near Neighbors," *Information Processing Letters*, **6:6** (1977) 209–212.

[18] Valentino B. Braitenberg, *Vehicles: Experiments in Synthetic Psychology*, (MIT Press, 1984).

[19] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone, *Classification and Regression Trees*, (Wadsworth International Group, Belmont, California, 1984).

[20] Korbinian Brodmann, *Vergleichende Lokalisationslehre der Grosshirnrinde in ihren Prinzipien dargestellt auf Grund des Zellenbaues*, (Barth, Leipzig, 1909).

[21] Walter A. Burkhard, "Partial-Match Hash Coding: Benefits of Redundancy," *ACM Transactions on Database Systems*, **4:2** (1979) 228–239.

[22] Richard Cole, "Parallel Merge Sort: Extended Abstract," preprint New York University (1986).

[23] Douglas Comer, "Heuristics for Trie Index Minimization," *ACM Transactions on Database Systems*, **4:3** (1979) 383–395.

[24] L. N. Cooper, F. Liberman and E. Oja, "A theory for the acquisition and loss of neuron specificity in visual cortex," *Biological Cybernetics* **33** (1979) 9–28.

[25] T. M. Cover and P. E. Hart, "Nearest Neighbor Pattern Classification," *IEEE Transactions on Information Theory*, **IT-13:1** (1967) 21–27.

[26] F. H. C. Crick and C. Asanuma, "Certain aspects of the anatomy and physiology of the cerebral cortex," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 2: Psychological and Biological Models*, edited by J. L. McClelland & D. E. Rumelhart, (MIT Press, 1986).

[27] J. R. Driscoll, N. Sarnak, D. D. Sleator, R. E. Tarjan, "Making data structures persistent," *Proceedings of the 1986 ACM Symposium on the Theory of Computing* (1986) 109–121.

[28] C. M. Eastman and S. F. Weiss, "Tree Structures for High Dimensionality Nearest Neighbor Searching," *Information Systems* **7:2** (1982) 115–122.

[29] H. Edelsbrunner and J. van Leewen, "Multidimensional data structures and algorithms, A bibliography," IIG, Technische University Graz, Austria, Report 104 (1983).

[30] David C. Van Essen and John H. R. Maunsell, "Hierarchical organization and functional streams in the visual cortex," *Trends in Neuroscience* (1983) 370–375.

[31] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong, "Extendible Hashing—A Fast Access Method for Dynamic Files," *ACM Transactions on Database Systems*, **4:3** (1979) 315–344.

[32] Doyne Farmer and John Sidorwich, "Predicting Chaotic Time Series," in preparation.

[33] B. Faverjon, "Obstacle Avoidance Using an Octree in the Configuration Space of a Manipulator," in *Proceedings of the IEEE International Conference on Robotics* (1984).

[34] J. A. Feldman, "Dynamic connections in neural networks," *Biological Cybernetics*, **46** (1982) 27–39.

[35] J. A. Feldman and D. H. Ballard, "Connectionist Models and Their Properties," *Cognitive Science*, **6** (1982) 205–254.

[36] Ysmar V. Silva Filho, "Optimal Choice of Discriminators in a Balanced *K-d* Binary Search Tree," *Information Processing Letters*, **13:2** (1981) 67–70.

[37] R. A. Finkel and J. L. Bentley, "Quad Trees: A Data Structure for Retrieval on Composite Keys," *Acta Informatica* **4** (1974) 1–9.

[38] Philippe Flajolet and Claude Puech, "Tree Structures for Partial Match Retrieval," in *The 24th Annual Symposium on Foundations of Computer Science*, (IEEE Computer Society Press, Los Angeles, CA 1983) 282–288.

[39] E. Fredkin, "Trie Memory," *Communications of the ACM*, **3:9** (1960) 490–499.

[40] Jerome H. Friedman, Jon Louis Bently, and Raphael Ari Finkel, "An Algorithm for Finding Best Matches in Logarithmic Expected Time," *ACM Transactions on Mathematical Software*, **3:3** (1977) 209–226.

[41] Jerome H. Friedman, Forest Baskett, and Leonard J. Shustek, "An Algorithm for Finding Nearest Neighbors," *IEEE Transactions on Computers* (1975) 1000–1006.

[42] Keinosuke Fukunaga and Larry D. Hostetler, "Optimization of k-Nearest-Neighbor Density Estimates," *IEEE Transactions on Information Theory* (1973) 320–326.

[43] Keinosuke Fukunaga and Patrenahalli M. Narendra, "A Branch and Bound Algorithm for Computing k-Nearest Neighbors," *IEEE Transactions on Computers* (1975) 750–753.

[44] Kunihiko Fukushima, Sei Miyake, and Takayuki Ito, "Neocognitron: A Neural Network Model for a Mechanism of Visual Pattern Recognition," *IEEE Transactions on Systems, Man, and Cybernetics*, **SMC-13:5** (1983) 826–834.

[45] H. H. Goldstine and Joh von Neumann, *Planning and coding of problems for an electronic computing instrument, Part II, Vol 1* (1947); reprinted in A. H. Taub, ed., *John von Neumann – Collected Works, Volume 5*, (Pergamon Press, 1963).

[46] G. H. Gonnet, *Handbook of Algorithms and Data Structures*, (Addison-Wesley 1984).

[47] D. O. Hebb, *Organization of Behavior*, (John Wiley & Sons, 1949).

[48] Paul Heckbert, "Color Image Quantization for Frame Buffer Display," *Computer Graphics* **16:3** (1982) 297–304.

[49] Danny Hillis, *The Connection Machine*, (MIT Press, 1986).

[50] Klaus Hinrichs, "Implementation of the grid file: Design Concepts and Experience," *Bit*, **25** (1985) 569–592.

[51] Hecht-Nielsen Neurocomputers, 5893 Oberlin Drive, San Diego, CA 92121.

[52] G. E. Hinton, "Learning distributed representations of concepts," *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, (Hillsdale, New Jersey: Erlbaum, 1986) 1–12.

[53] G. E. Hinton and J. A. Anderson, *Parallel models of associative memory*, (Hillsdale, New Jersey: Erlbaum Associates, 1981).

[54] G. E. Hinton and T. J. Sejnowski, "Optimal perceptual inference," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, (Washington, D. C., 1983) 448–453.

[55] G. E. Hinton and T. J. Sejnowski, "Learning and relearning in Boltzmann machines," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 2: Psychological and Biological Models*, edited by J. L. McClelland & D. E. Rumelhart, (MIT Press, 1986).

[56] C. A. R. Hoare, "Quicksort," *Computer Journal*, **5:1** (1962) 10–15.

[57] Darrell Hougen and Stephen M. Omohundro, "On Learning to Discriminate Textures," in preparation.

[58] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the National Academy of Science, USA* **79** (1982) 2554–2558.

[59] J. J. Hopfield and D. W. Tank, "Neural Computation of Decisions in Optimization Problems," *Biological Cybernetics* **52** (1985) 141.

[60] J. J. Hopfield and D. Tank, "Computing with neural circuits: A model," *Science*, **233** (1986) 624–633.

[61] R. Colin Johnson, "Neural Net Speech System In Works," *Electronic Engineering Times*, (March 23, 1987) 10.

[62] Eric R. Kandel and James H. Schwartz, *Principles of Neural Science, Second Edition*, (Elsevier Science Publishing Co., Inc., 1985).

[63] Baek S. Kim and Song B. Park, "A Fast k Nearest Neighbor Finding Algorithm Based on the Ordered Partition", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **8:6** (1986) 761–766.

[64] T. Kohonen, "Correlation Matrix Memories," *IEEE Transactions on Computers*, **C-21** (1972) 353–359.

[65] T. Kohonen, *Self-Organization and Associative Memory*, (New York: Springer Verlag, 1984).

[66] Clyde P. Kruskal, Larry Rudolph, and Marc Snir, "The Power of Parllel Prefix," in *Proceedings of the 1985 International Conference on Parallel Processing*, (IEEE Computer Society Press, Washington, D. C., 1985) 180–185.

[67] R. E. Ladner and M. J. Fischer, "Parallel Prefix Computation," *Journal of the Association of Computing Machinery* (1980) 831–838.

[68] D. T. Lee and Franco P. Preparata, "Computational Geometry–A Survey," *IEEE Transactions on Computers*, **C-33:12** (1984) 1072–1101.

[69] Eric J. Lerner, "Why Can't a Computer be More Like a Brain?", *High Technology*, August (1984) 34–78.

[70] G. F. Lev, N. Pippenger, and L. G. Valiant, "A Fast Parallel Algorithm for Routing in Permutation Networks," *IEEE Transactions on Computers*, **C-30** (1981) 93–100.

[71] Stephen E. Levinson, "Structural Methods in Automatic Speech Recognition," *Proceedings of the IEEE* **73:11** (1985) 1625–1650.

[72] D. O. Loftsgaarden and C. P. Quesenberry, "A Nonparametric Estimate of a Multivariate Density Function," *Annals of Mathematical Statistics* **36** (1965) 1049–1051.

[73] J. W. Lloyd, "Optimal Partial-Match Retrieval," *Bit* **20** (1980) 406–413.

[74] John W. Lloyd and K. Ramamohanarao, "Partial-Match Retrieval for Dynamic Files," *Bit* **22** (1982) 150–168.

[75] John Makhoul, Salim Roucos, and Herbert Gish, "Vector Quantization in Speech Coding," *Proceedings of the IEE* **73:11** (1985) 1551–1588.

[76] Edward B. Manoukian, *Modern Concepts and Theorems of Mathematical Statistics*, (Springer-Verlag, 1986).

[77] J. L. McClelland, "Resource Requirements of Standard and Programmable Nets," in D. E. Rumelhart and J. L. McClelland, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 1: Foundations*, (MIT Press, 1986) 460–487.

[78] J. L. McClelland and D. E. Rumelhart, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 2: Psychological and Biological Models*, (MIT Press, 1986).

[79] W. S. McCulloch and W. H. Pitts, "A logical calculus of ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, **5** (1943) 115–133.

[80] Kurt Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, (Springer-Verlag, Berlin, 1984).

[81] Michael M. Merzenich and Jon H. Kaas, "Principles of Organization of Sensory-Perceptual Systems in Mammals," *Progress in Psychobiology and Physiological Psychology*, **9**, 1–42.

[82] R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, eds., *Machine Learning: An Artificial Intelligence Approach, Vols. I and II*, (Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986).

[83] Marvin L. Minsky, *Computation: Finite and Infinite Machines*, (Prentice-Hall, Inc., 1967).

[84] Marvin Minsky, *The Society of Mind*, (Simon and Schuster, 1986).

[85] M. Minsky and S. Papert, *Perceptrons*, (MIT Press, 1969).

[86] Donald R. Morrison, *Journal of the ACM* **15** (1968) 514–534.

[87] V. B. Mountcastle, "An organizing principle for cerebral function: The unit module and the distributed system," in *The Mindful Brain*, edited by G. M. Edelman & V. B. Mountcastle, (MIT Press, 1978).

[88] Nestor Inc., 1 Richmond Square, Providence, Rhode Island 02906.

[89] Neural Tech, 177 Goya Road, Portola Valley, CA 94025.

[90] Neurocomputers, Gallifrey Publishing, P. O. Box 155, Vicksburg, Michigan 49097.

[91] O. Nevalainen, J. Ernvall, and J. Katajainen, "Finding Minimal Spanning Trees in a Euclidean Space," *Bit* **21** (1981) 46–54.

[92] J. Nievergelt, "Trees as data and file structures," In *CAAP '81, Proc. 6th Colloquium on Trees in Algebra and Programming,* E. Astesiano and C. Bohm, Eds., *Lecture Notes in Computer Science 112,* (Springer Verlag, 1981) 35–45.

[93] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure," *ACM Transactions on Database Systems,* **9:1** (1984) 38–71.

[94] M. T. Noga and D. C. S. Allison, "Sorting in Linear Expected Time," *Bit* **25** (1985) 451–465.

[95] Lawrence O'Gorman and Arthur C. Sanderson, "The Converging Squares Algorithm: An Efficient Method for Locating Peaks in Multidimensions," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* **6:3** (1984) 280–288.

[96] Stephen M. Omohundro, *Geometric Perturbation Theory in Physics,* (World Scientific Publishing Co. Pte. Ltd., Singapore, 1986).

[97] Stephen M. Omohundro, "Fast Image Segmentation and Analysis Using Partial Sums," in preparation.

[98] Jack A. Orenstein, "Multidimensional Tries for Associative Searching," *Information Processing Letters,* **14:4** (1982) 150–157.

[99] Joseph O'Rourke and Kenneth R. Sloan, Jr., "Dynamic Quantization: Two Adaptive Data Structures for Multidimensional Spaces," *IEEE Trans. on Pattern Analysis and Machine Intelligence,* **6:3** (1984) 266–279.

[100] Mohamed Ouksel and Peter Scheuermann, "Multidimensional B-Trees: Analysis of Dynamic Behavior," *Bit* **21** (1981) 401–418.

[101] Mark H. Overmars, *The Design of Dynamic Data Structures,* Lecture Notes in Computer Science, Number 156, (Springer-Verlag, 1983).

[102] Mark H. Overmars and Jan van Leeuwen, "Dynamic Multi-Dimensional Data Structures Based on Quad- and K-D Trees," *Acta Informatica,* **17** (1982) 267–285.

[103] Theo Pavlidis, *Algorithms for Graphics and Image Processing,* (Computer Science Press, Rockville, Maryland, 1982).

[104] F. P. Preparata and J. Vuillemin, "The cube-connected-cycles: A versatile network for parallel computation," *Comm. of the ACM* **24:5** (1981) 300–309.

[105] Franco P. Preparata and Michael Ian Shamos, *Computational Geometry, An Introduction*, (Springer-Verlag, 1985).

[106] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling, *Numerical Recipes: The Art of Scientific Computing*, (Cambridge University Press, New York 1986).

[107] Mireille Regnier, "Analysis of Grid File Algorithms," *Bit* **25** (1985) 335–357.

[108] J. H. Reif and L. G. Valiant, "A logarithmic time sort for linear size networks," *Proceedings of the Fifteenth Annual ACM Symposium on the Theory of Computing* (1983) 10–16.

[109] Revelations Research, 4261 Sherwoodtowne Blvd., Mississauga, Ont L4Z 1Y5, Canada.

[110] H. Ritter and K. Schulten, "On the Stationary State of Kohonen's Self-Organizing Sensory Mapping," *Biological Cybernetics* **54** (1986) 99–106.

[111] Ronald L. Rivest, "Partial-match Retrieval Algorithms," *SIAM Journal of Computing*, **5:1** (1976) 19–50.

[112] F. Rosenblatt, *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*, (Spartan Books, Washington D. C., 1961).

[113] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Internal Representations by Error Propagation," in D. E. Rumelhart and J. L. McClelland, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 1: Foundations*, (MIT Press, 1986) 318–362.

[114] D. E. Rumelhart, G. E. Hinton, and J. L. McClelland, "A General Framework for Parallel Distributed Processing," in D. E. Rumelhart and J. L. McClelland, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 1: Foundations*, (MIT Press, 1986) 45–76.

[115] D. E. Rumelhart and J. L. McClelland, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 1: Foundations*, (MIT Press, 1986).

[116] D. E. Rumelhart and D. Zipser, "Feature Discovery by Competitive Learning," *Cognitive Science* **9** (1985) 75–112.

[117] Hanan Samet, "The quadtree and related hierarchical data structures," *ACM Computing Surveys* **16:2** (1984) 187–260.

[118] Peter Scheuermann and Mohamed Ouksel, "Multidimensional B-Trees for Associative Searching in Database Systems," *Information Systems*, **7:2** (1982) 123–137.

[119] J. T. Schwartz, "Ultracomputers," *ACM Transactions on Programming Languages and Systems* **2:4** (1980) 484–521.

[120] Terrence J. Sejnowski and Charles M. Rosenberg, "Parallel Networks that Learn to Pronounce English Text," *Complex Systems* **1:1** (1987) 145–168.

[121] M. I. Shamos and D. Hoey, "Closest-point problems," *Sixteenth Annual IEEE Symposium on Foundations of Computer Science*, (1976) 151–162.

[122] D. D. Sleator and R. E. Tarjan, "Self-Adjusting Binary Trees", *Proceedings of the 15th ACM Symposium on the Theory of Computing*, (1983) 235–245.

[123] K. Steinbuch, "Die Lernmatrix," *Kybernetik* **1** (1961) 36–45.

[124] K. Steinbuch, U. A. W. Piske, *IEEE Transactions* **EC-12** (1963) 846.

[125] Synaptics, 2860 Zanker Road, Suite 105, San Jose, CA 95134.

[126] Markku Tamminen, "The Extendible Cell Method for Closest Point Problems," *Bit*, **22** (1982) 27–41.

[127] Robert Endre Tarjan, *Data Structures and Network Algorithms*, CBMS-NSF Regional Conference Series in Applied Mathematics, Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania (1983).

[128] Gerald Tesauro, "Scaling Relationships in Back-Propagation Learning: Dependence on Training Set Size," *Complex Systems*, this issue.

[129] Thinking Machines Corporation, 245 First St., Cambridge, MA, Connection Machine promotional literature.

[130] N. Z. Tishby, private communication.

[131] J. D. Ullman, *Computational Aspects of VLSI*, (Computer Science Press, Rockville, Md., 1984).

[132] L. G. Valiant, "A Theory of the Learnable," *Communications of the ACM*, **27:11** (1984) 1134–1142.

[133] L. G. Valiant, "Learning Disjunctions of Conjunctions," *Proceedings of the International Joint Conference on Artificial Intelligence*, (1985).

[134] L. G. Valiant and G. J. Brebner, "Universal schemes for parallel communication," *Proceedings of the Thirteenth Annual ACM Symposium on the Theory of Computing* (1981) 263–277.

[135] C. J. D. M. Verhagen, R. P. W. Duin, F. C. A. Groen, J. C. Joosten, and P. W. Verbeek, "Progress Report on Pattern Recognition," *Rep. Prog. Phys.*, **43** (1980) 786–831.

[136] B. Widrow, "Generalization and Information Storage in Networks of Adaline Neurons," in *Self-Organizing Systems 1962*, ed. M. C. Yovits, G. T. Jacobi, and G. D. Goldstein, (Spartan Books, Washington D. C., 1962) 435.

[137] David Willshaw, "Holography, Associative Memory, and Inductive Generalization," in *Parallel Models of Associative Memory* edited by Geoffrey E. Hinton and James A. Anderson, (Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1981) 83–104.

[138] Terry Winograd, "Beyond Programming Languages," *Communications of the ACM*, **22:7** (1979) 391–401.

[139] Stephen Wolfram, *Theory and Applications of Cellular Automata*, (World Scientific Publishing Co. Pte. Ltd., Singapore, 1986).

[140] Stephen Wolfram, "Approaches to Complexity Engineering," *Physica* **22D** (1986) 385–399.

[141] Chuan-lin Wu and Tse-yun Feng, eds., *Interconnection Networks for Parallel and Distributed Processing*, (IEEE Computer Society Press, Los Angeles, CA, 1984).

[142] L. A. Zadeh, *IEEE Transactions on Systems, Man, and Cybernetics*, **SMC-3** (1974) 38.

[143] Charles T. Zahn, "Graph-Theoretical Methods for Detecting and Describing Gestalt Clusters," *IEEE Transactions on Computers*, **C-20:1** (1971) 68–86.