The Design of Custom VLSI for Learning by Example

Aaron Lipman Woodward Yang

November 7, 1994

Abstract

Learning by example is a technique used widely in pattern classification and function approximation tasks. Neural networks and memory based approaches are common implementations of learning by example, and both have speed bottlenecks. Neural netowrks require several iterations to tune parameters based on an example database, and memory based approaches require repeated queries of close matches from an example database. The construction of a hardware "Top N" search engine which could rapidly retrieve the N closest matches to a querry point would speed up both memory based computation, and neural networks that are composed of locally tuned units. There are two strategies for retrieving the top N, brute force searching, and k-d trees. While k-d trees are efficient for low dimensional examples, their efficiency decreases exponentially with increasing dimensionality of the examples. Thus, for high dimensional example data, a brute force search is appropriate. The brute force search has a simple parallel implementation which makes it ideal for layout in VLSI, where the algorithm can realize an acceleration on the order of 10^2 *to* 10^3.

1 Introduction

Traditional digital computers are designed for computational tasks that can be described by an algorithm. In many cases, this is the most natural paradigm to solve the task at hand; word processing, database programs, and scientific computing are all good examples of tasks ideally suited to digital computers. There are also large classes of computational tasks that do not have a straightforward, or computationally tractable algorithmic implementation. Examples of these types of problems are object and character recognition, weather and financial forecasting, and the realtime control of high dimensional nonlinear systems. A characteristic common to many of these problems is that they can be implemented as a relationship to be learned from a collection of data, or *example database*, for the purpose of generalizing the relationship to the processing of new data outside the example database. The example database might be composed of images, time series data, or input/output measurements of a system. This approach is known as *learning by example*.

Thus, the process of learning by example creates an approximation which is a *generalization* of an example database. Tasks using learning by example are typically broken down into two main categories. Classification problems concern the generation of categories associated with various example vectors, or groups of example vectors. Once the feature space of these vectors has been segmented into categories, new input vectors can be identified by the region of feature space into which they map. Continuous function approximation tasks treat the examples as samples of a multidimensional function to be reconstructed in some manner.

2 Approaches

2.1 Network Based Approaches

Network based approaches, often called neural networks, include multi-layer perceptron (MLP) networks, radial basis function (RBF) networks, and Kohonen mappings, are characterized by a set of parameters that describe various units or basis functions that comprise the network. A basic neural network architecture is shown in figure 1. The input units fan a multidimensional input to intermediate units, which may be non-linear functions, or algorithmic processes, and the output of each is combined to produce a result. Neural networks require a learning phase where the parameters are adjusted such that the network can generalize properly from the example database. This can be done via deterministic or stochastic methods depending on the network, and in most cases, the true optimal parameters cannot be determined in a practical manner, so the learning phase is terminated when the network performance is at satisfactory level. There are several important considerations that affect the selection and implementation of a network. Perhaps most important is identifying which network structure is best suited to a given problem. For example, in a classification task, various networks segment the feature space differently. Some networks section it using hyperplanes, and others may use hyperspheres. The nature of the

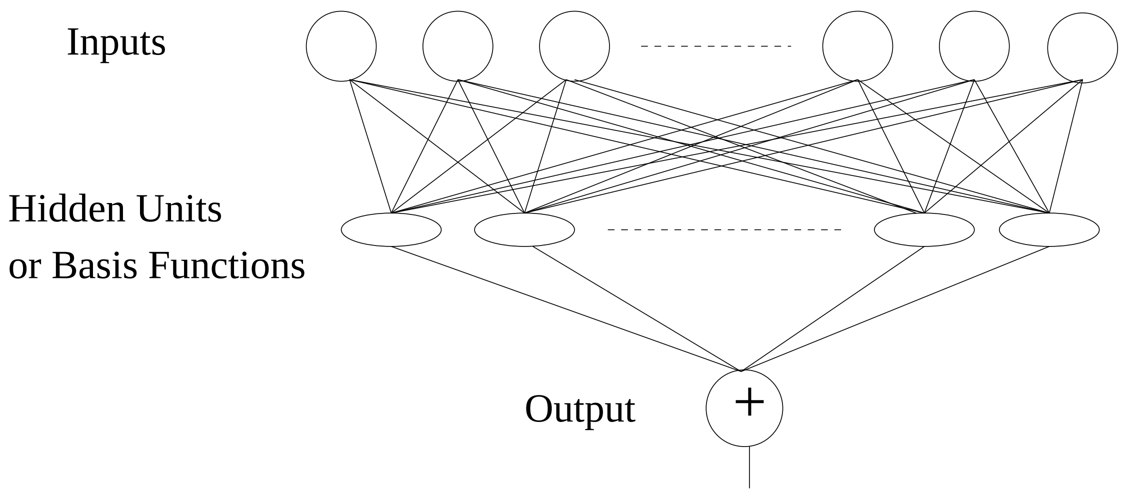


Figure 1: Sample neural network architecture.

data may dictate that the segmentation offered by one network is best suited to the problem. Unfortunately, it is not always possible to determine this in advance. A second consideration is the determination of the number of parameters to be used. In a classification task, it may be apparent that exactly n units are needed to classify n categories, but this is not always the case. In general, most algorithms for neural network learning require several thousand iterations of parameter adjustment which is a slow, computationally intensive procedure.

2.2 Memory Based Aproaches

Memory-based approaches differ from standard network approaches in that they typically are not characterized by a set of parameters. Rather, an entire example database stored in memory, and as new data is presented, examples relvant to the new input are recalled from the memory. Determining which examples are relevant varies widely with the approach taken. Usually, a distance metric, such as an L_2 norm is chosen, and the n closest points to an input $\vec{x}$, are retrieved from the memory. The number of points n needed, and the exact metric used depend on the application. A classification might require the nearest point to be retrieved using an L_2 norm, and the input will be classified in the same category as the nearest neighbor. Function approximation applications might require the retrieval of 30 or 40 nearby points, and an interpolation procedure applied to the retrieved points. A different distance metric might be used that weights the distances in each dimension separately. A memory-based approach to learning by example can also be thought of as a network where each example is a unit of some sort. The difference between the two approaches lies mostly in the practical implementation. In general, memory based approaches are characterized by having no training phase. Instead, the entire training set is stored in some sort of memory. When new inputs are presented, the memory is searched, certain relevant examples are extracted, and an output is determined by some operation on the extracted examples.

2.3 Comparison of Network and Memory Based Approaches

Network based approaches compact a training set into a number of parameters that describe the network. These parameters typically describe prototypes of the training set. The location of the basis functions in RBF networks indicate prototypes of the data, as do the weights in the competitive layer of a Kohonen feature map. In the case of MLP networks, prototype regions are identified by intersections of hyperplanes. In this sense, it is more difficult to determine what an MLP network has learned after being trained, whereas the parameters of the other networks may reveal how the network has generalized the data. The compression of information is another useful property of these networks. A series of fifty thousand training pairs might be approximated by one hundred vector quantization prototypes, or basis functions. Thus, the network has extracted the important properties of the data in a much smaller number of parameters. As a consequence, forward computation time of these networks is typically small, and tends to scale linearly with the number of units used. The practical drawback is that these networks usually require long learning phases that may require on the order of thousands to millions of iterations to converge. Furthermore, the exact number of parameters needed to adequately generalize from the training set is not always apparent. In function approximation tasks, the number of units needed to accurately approximate the function is highly dependent on the unknown function. There may be general guidelines for specific cases, but there is almost never an exact rule. Several training phases may be required for an individual training set to ensure that the parameters chosen are nearly optimal. Also, network based approaches are typically unable to adapt to a dynamic training set. Some applications require the incorporation of new data that arrives continuously, and perhaps the discarding of old data. This would require the network to learn continuously, and possibly 'unlearn' generalizations culled from old data. Most networks are designed to learn from a static training set, and cannot cope efficiently with new data. Many of these complications are alleviated with memory based approaches. It is simple to add new data and remove old data simply by updating the memory. Furthermore, it is not necessary to choose a number of parameters, or perform a long training phase. However, the forward computation speed of memory-based approaches is typically quite slow, with a potentially exhaustive search of the memory required, and possibly, a fitting of the retrieved nearby points, which may be computationally intensive. Also, the exact number n of nearby or 'relevant' points that must be retrieved is not always straightforward but is relatively easy to adjust. Proper selection of a distance metric is crucial to the performance of any memory based approach. The similarities between the two approaches are based on the fact that they both provide a look-up table operation. In fact, as mentioned previously, a memory-based approach can be described as a learning network with unit per example, and an associated algorithm that processes certain units to produce a result. In fact, an RBF network with a basis function per example is very much a memory-based approach.

In general, all approaches to learning by example perform some type of look-up table operation. An input vector can be thought of as an address into a lookup table, with the corresponding output residing at the memory location pointed to by that address. Most tasks have address spaces that are much larger than the capability of any lookup table. For example, imagine a character recognition application that uses an 8×8 grid where each square in the grid takes on a value of 1 or 0. It is possible to classify every possible input character by generating a lookup table with 2^{64} entries, each corresponding to a possible grid configuration. This is not realizable, because such a table would require on the order of 10^{19} bytes, (more memory than all the hard disks in the world put together). Instead, outputs are generated by generalizing from an example database, which fills in the missing entries in the look-up table. This incorporates a notion of locality. Examples that lie in close proximity to an input vector will be the most influential in determining the output. Typically, there is a relatively small number of examples in a database, usually on the order of 10^2 to 10^5, whereas the dimensionality of the examples might range from two to several thousand dimensions. Given the fact that in traditional function approximation problems, the number of samples needed to reconstruct a function rises exponentially with dimension, it may seem impossible to perform learning in high dimensional spaces with few examples. Fortunately, most of the examples tend to define a lower dimensional manifold in the higher dimensional space, and the generalization is usually performed in this manifold. The implication is that several of the dimensions are either non independent, or are fairly constant across the training set. There are methods for reducing the dimensionality of the training set, such as principle component analysis (PCA), which can sometimes be used to pre-process the examples, and form a lower dimensional training set, but PCA can be extremely computationally intensive, and is not always practical.

3 The Top-n Approach

3.1 Motivation

Neural networks, and memory based computation perform useful tasks, but each can be cumbersome. Neural networks require long learning phases, and memory based computation requires exhaustive memory searches. Is there are hardware approach that can accelerate both methods? The bottleneck in most memory-based approaches can be seen as the retrieval of the n most "relevant" examples. A software or hardware solution that could accelerate this search would directly speed up these memory-based approaches. Furthermore, as discussed previously, the notion of locality is important to many network-based approaches. RBF networks, vector quantization, and Kohonen Mappings are all composed of units that respond to local example clusters. MLP networks though, do not fall into this category. Typically in these networks, parameters are tuned across all the examples in an example database, even though only a small fraction of the examples may be relevant to the parameter being tuned. Thus, rapid retrieval of local examples could accelerate the parameter adjustment of locally tuned units by greatly reducing the amount of computation needed to adjust each parameter. We refer to this approach as the "Top-n" approach. The search for top n can be performed using a brute force method, where every example in the database is investigated. There are also algorithmic approaches to accelerating the search by sorting the data using multi-dimensional decision trees. The merits of both approaches will be discussed.

3.2 k-d Trees

A k-dimensional tree, or *k-d tree* is a multidimensional extension of a decision tree, used to sort multi-dimensional data [1]. Each level in the tree contains a decision along one of the dimensions of the data. Typically, the decision on the top of the tree corresponds to a splitting of the data along the first dimension, second level of the tree to the second dimension, and so on, repeating with the first dimension after the last dimension has been split. Figure 2 shows an example of a k-d tree partitioning for 2-dimensional data. The tree on the right shows the decomposition of the data

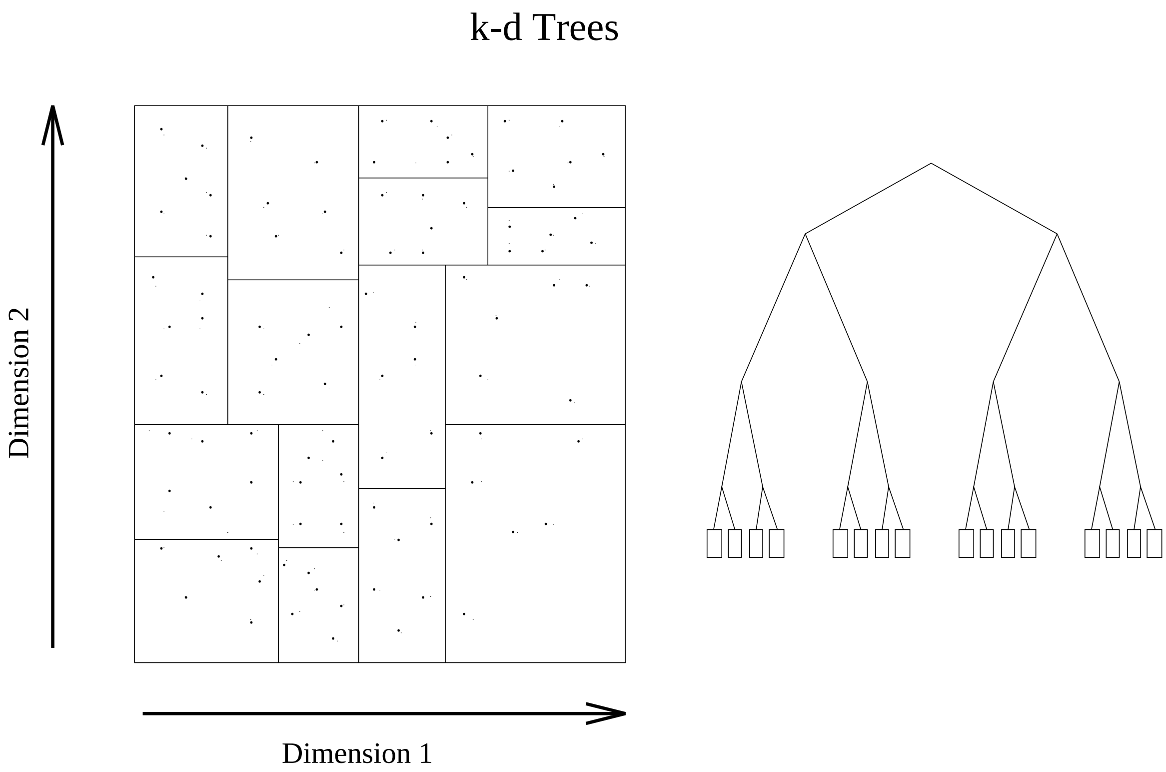


Figure 2: *k-d tree partitioning of a 2-dimensional data cluster*

cluster. The top level of the corresponds to a decision in the first dimension of the data, the second level to decisions in the second dimension, the third level refers back to the first dimension, and this continues until the data has been partioned into rectangular areas that each hold a small portion of the data. The ordering of the dimensions can be arbitrary, but typically they are alternated at each level. These rectangular areas are the terminal nodes of the tree, sometimes called buckets, and may contain any number of data points. This is shown pictorially by the figure to the left of the tree. The construction of the k-d tree is still order $N \log_2 N$, and recall of an item from the tree is still order $\log_2 N$. K-d trees become useful in memory based applications when they are used to retrieve nearest neighbors. When a query point is presented to the k-d tree, the tree structure is used to determine which hyper-rectangle, or bucket a query point belongs to. The distance from the query point to each point in the bucket is measured, and the results are placed in a priority queue which keeps track of the top n thus far retrieved. If there are fewer points to be retrieved than points in the bucket, the queue will be filled, but may not necessarily contain the closest n points, because the query point may lie near a boundary in the corresponding hyper-rectangle. The distance from the farthest point in the priority queue to the query point is measured, and a hypersphere is formed with the query point as center, and the distance measurement as radius. If the hyper-rectangle contains that hypersphere within its boundaries, the true top n have been found. If not, back searching is performed, and neighboring buckets on the tree are searched after backing up a level in the tree. If the hyper-rectangle defined by the level above the original bucket contains the hypersphere defined above, the true top n have been found. If not, the the algorithm must continue to search further and further up the tree. This is illustrated in figure 3. The 'x' in the figure indicates the

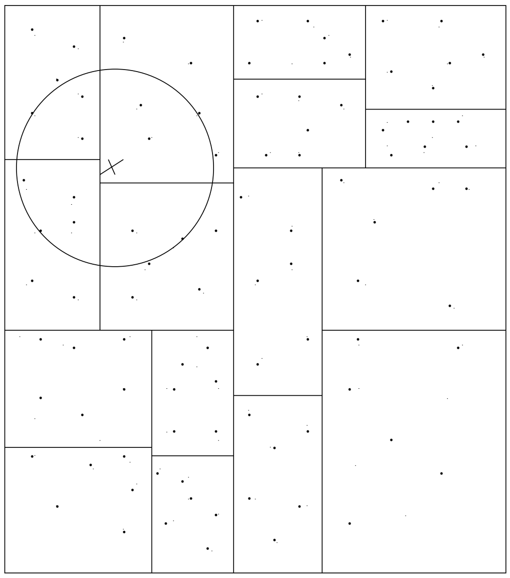


Figure 3: *backsearching to find the nearest neighbors*

location of the query point. The circle includes the ten nearest points to the query point. Initially, the six points in the rectangle that contains the query point were in the priority queue, but since the query point lies near the corner of the rectangle, the true top ten points lie in four different buckets. The retrieval time of the top n points depends on the amount of backsearching required, and the number of data points examined, which are both related to the construction of the tree. To this end, Friedman *et al* [3] suggest finding the dimension with maximal spread at each level. This requires an order N computation for each dimension at each step. For k dimensions, it takes order $kN \log_2 N$ time to construct the tree. This ensures more spatial locality, as backsearching is performed up the tree. In general, this method creates a balanced tree. In addition, the number of data points in each bucket affects the amount of backsearching. Friedman *et al* [3] note that if only one point is in each bucket, a tremendous amount of backsearching is required which slows down the algorithm. When the buckets are too large, the algorithm is forced to examine more data points than typically necessary. Usually, choosing between ten and twenty points per bucket provides an acceptable balance. Also, the distribution of the data affects the amount of backsearching required. Balanced trees work well for uniformly distributed data, but are not as efficient for other distributions. Omohundro [4] proposes a series of tree construction algorithms that are suitable for different data distributions. Additionally, he proposes the use of decision trees as a general efficient approach to learning by example [5]. K-d trees have a small construction time, which could be thought of as a "learning" phase. The retrieval of the top n examples to an input query can be affected quickly for pattern classification, or for local interpolation. This method has been used with success in time series prediction [2], and for nearest neighbor searches in vector quantization encodings [6].

3.3 k-d Trees vs. Brute Force Searching

Decision trees begin to lose their advantage as the dimensionality of the data they sort increases. This is because the number of neighboring hyper-rectangles increases as 2^d where d is the number of dimensions. Thus, more data points much be searched to find the nearest n. k-d trees and brute force searching were implemented on an HP-755 workstation using C code. Each algorithm was used to retrieve the top 10 and 100 matches from databases of dimensionality varying from 2 to 16. Example database sizes of 10,000 and 100,000 examples were used. Retrieval time per query was used as a performance metric. The results are shown in figure 4 The graph shows the retrieval time for the top 10 points out of a database of 10,000 examples versus the dimensionality of the examples. Both linear and semi-log plots are shown. The dashed lines represent the k-d tree performance. The time required for retrieval grows exponentially with dimension, as more and more examples are searched. The curve levels off at higher dimensions, because almost all the examples in the database must be searched, so the time required saturates. Brute force search timing is represented by the solid line, and shows almost no increase with increased dimension. The two methods have roughly the same performance at nine input dimensions, but above that, the brute force search is more efficient, because it does not require complex backsearching procedures. Thus, let us compare the performance of the two methods by indicating the dimension at which the speed of the two methods is equal. We call this the *performance equivalency point*. This point is dependent on the size of the database primarily, and the algorithmic implementation of the two methods secondarily. For a given database size, brute force will be more efficient for dimensionality higher than the equivalency point, and k-d trees will be more efficient for lower dimensionality. The log plot shows the improvement offered by k-d trees at low dimensionality. For two-dimensional examples, the k-d tree method is roughly 500 times faster than a brute force approach. However, the brute force method is ten times as fast as the k-d tree for sixteen-dimensional examples. The dimensionality at which k-d tree and brute force search methods have equal performance changes as the number of examples in the database, and the top n points to be retrieved changes. As the number of examples in the database rises, the dimensionality at which the performance of the two methods is equivalent rises. Figure 5 shows the retrieval of the top 10 points out of 100,000 examples, and the performance equivalency point, is roughly 13 input dimensions.

The performance equivalency point decreases as the top n required increases, but always increases as the number of examples in the database increases. This behavior is summarized in table 1. The number of dimensions for which the two methods

Number of Matches	Intersection for 10^2 examples	Intersection for 10^3 examples	Intersection for 10^4 examples	Intersection for 10^5 examples
1	2	6	11	15.5
10	brute force always faster	4.5	9	13
100		brute force always faster	6	10

Table 1: *Dimensionality at which k-d trees and brute force searches have equivalent performance for various database sizes and retrieval sizes.*

are equivalent increases with data base size, and decreases as the number of matches required increases. An approximate formula is

$$D \propto \log_{10} \frac{\text{examples}^{c_1}}{\text{matches}^{c_2}} \tag{1}$$

or

$$D \propto c_1 \log_{10} \text{examples} - c_2 \log_{10} \text{matches} \tag{2}$$

Where D is the dimensionality at which the performance of the two methods is equivalent, and c_1 and c_2 are constants determined by the implementation. For the results obtained on the HP-755, $c_1 \approx 4$ and $c_2 \approx 2.5$. Thus, performing a brute force search of memory is a desirable task for problems utilizing example databases with a high input dimensionality.

4 Accelerating Brute Force Searching with Custom VLSI

A brute force search engine can be implemented with custom parallel VLSI. There are essentially two major components of the brute force search; measuring the distance between a query point and the examples in the database by some sort of metric, (L_2 Norm for example), and sorting through the distances to retrieve the top n. While sorting can be performed in parallel, it requires massive communication. A sample architecture is shown in figure 6. The examples are divided among local memories which are each connected to a distance calculation unit, which will calculate distance as a weighted $L1$ or $L2$ norm. The results from each unit are fed into a corresponding linear sorting array. Each linear array contains a fixed number of storage locations, m. Thus, when the processors have finished streaming through memory, the top m points for each local memory will be stored in the linear arrays. The true global top m points can be retrieved from all these units by a binary tree reduction scheme. Thus, m must be larger than the desired value of top n to be retrieved. This solution is extremely hardware intensive, but all the processes can be pipelined in an efficient manner. The graphs of brute force query time shown in figures 4 and 5 show that while query time increases linearly with dimension, the slope is very shallow, much lower than an expected value of 1. In fact, timing results indicated that an average of 150 cycles was required to process a single example. Thus, not only can a speed-up be achieved by simple hardware parallelism, but processor efficiency introduces an additional speed-up.

An importnat aspect of this design is that it provides a high bandwidth to memory. Most workstations currently face a memory access bottleneck. Many processors can fetch instructions at a rate of 100 Mhz or more, whereas DRAM main memory accesses may be no faster than 15 Mhz. To combat this problem, fast local cache memories have been developed, but these are typically optimized for locality of reference, and frequency of access; i.e. they store frequently used variables and their neighbors. However, the brute force search process necessitates a sweep through the entire memory, and a cache is not helpful in this case. A custom VLSI implementation can increase the memory bandwidth by attaching fast local memories to each processing unit. By pipelining the computational processes, new data from the memory can be loaded on each clock cycle. A standard microprocessor will typically be much more inefficient, and may average several clock cycles per memory fetch. Thus, by using several custom VLSI brute force processors, a speed-up on the order of 10^2 to 10^3 over a workstation can be achieved.

5 Conclusion

Neural networks and memory based computation are used widely for learning by example. They each have computational bottlenecks arising from learning algorithms, or memory searches. We have seen how both approaches might be accelerated using a high-speed top-n search engine. For examples of low dimensionality, k-d tree algorithms are a good high-speed search engine, but as the dimensionality of the examples increases linearly, k-d tree performance decreases exponentially. Thus, for high-dimensional examples, a parallel brute force search becomes an attractive solution. We have shown a sample hardware layout of a parallel brute force search engine, and note that the key to its design is a high bandwidth between the memory and the processors. The system shown, once implemented, will offer a speed-up of 2 to 3 orders of magnitude over a conventional workstation.

References

[1] J.L. Bentley, "Multidimensional binary search trees used for associative searching." *Communications of the ACM* 18(9), pp. 509-517, 1975.

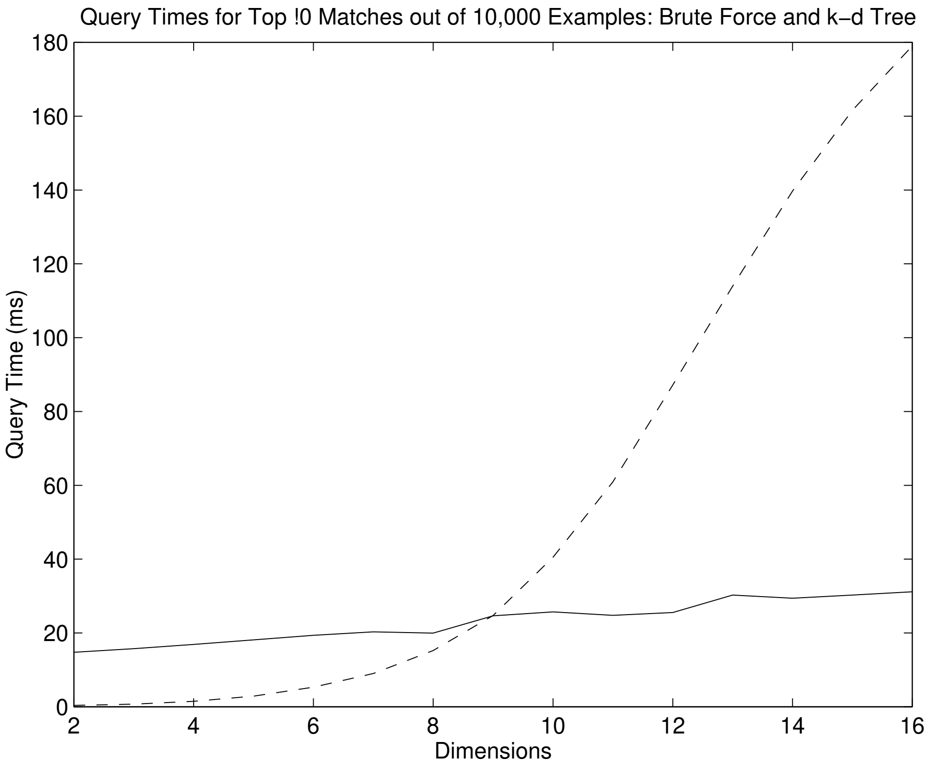
[2] J.D. Farmer and J.J. Sidorowich, "Predicting Chaotic Time Series." *Physical Review Letters*, 59(8):845-8, 1987.

[3] J.H. Friedman, J.L. Bentley, R.A. Finkel, "An Algorithm for Finding Best Matches in Logarithmic Expected Time." *ACM Transactions on Mathematical Software,* 3(3):209-226, 1977.

[4] S.M. Omohundro, "Five Balltree Construction Algorithms." International Computer Science Institute Technical Report tr-89-063, 1989.

[5] S.M. Omohundro, "Efficient Algorithms with Neural Network Behavior." *Complex Systems* 1:273-347, 1987.

[6] V. Ramasubramanian and K. Paliwal, "Fast K-Dimensional Tree Algorithms for Nearest neighbor Search with Application to Vector Quantization Encoding." *IEEE Transactions on Signal Processing*, 40(3):518-31, March 1992.


Query Times for Top !0 Matches out of 10,000 Examples: Brute Force and k−d Tree
Query Time (ms)
Dimensions
0
20
40
60
80
100
120
140
160
180
2
4
6
8
10
12
14
16

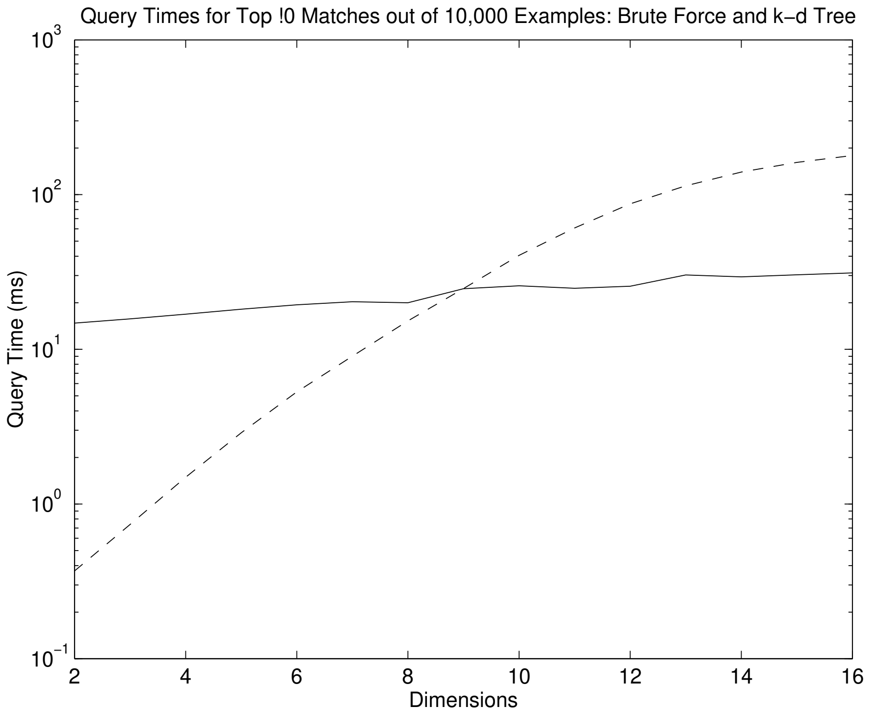
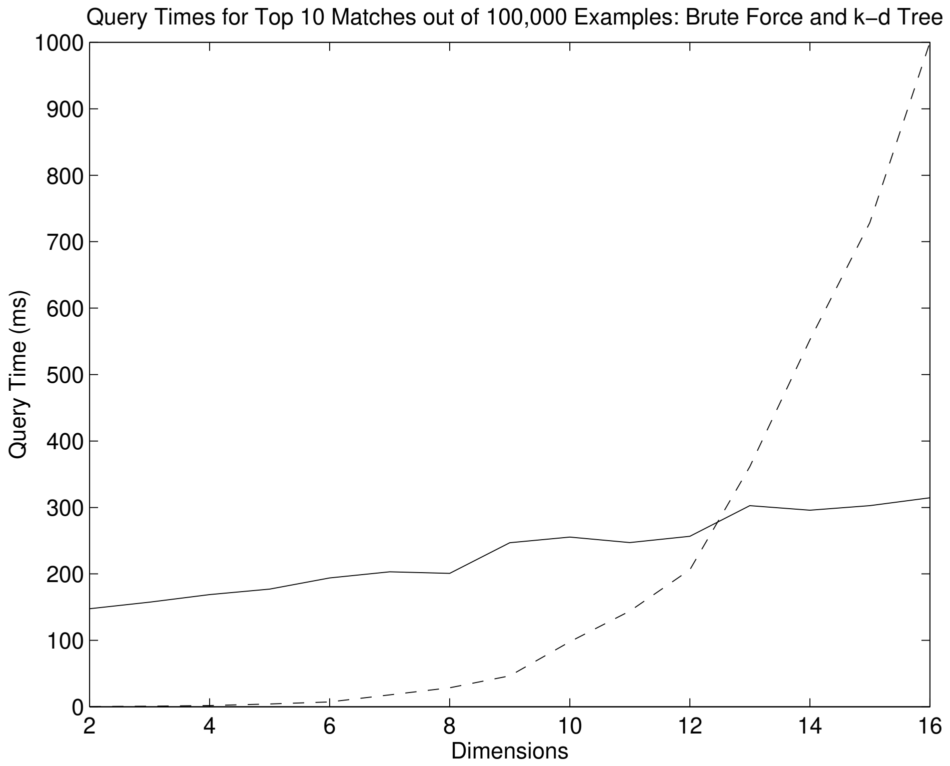

Query Times for Top !0 Matches out of 10,000 Examples: Brute Force and k−d Tree
Query Time (ms)
Dimensions
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
2
4
6
8
10
12
14
16

Figure 4:


Query Times for Top 10 Matches out of 100,000 Examples: Brute Force and k−d Tree
Query Time (ms)
Dimensions
0
100
200
300
400
500
600
700
800
900
1000
2
4
6
8
10
12
14
16

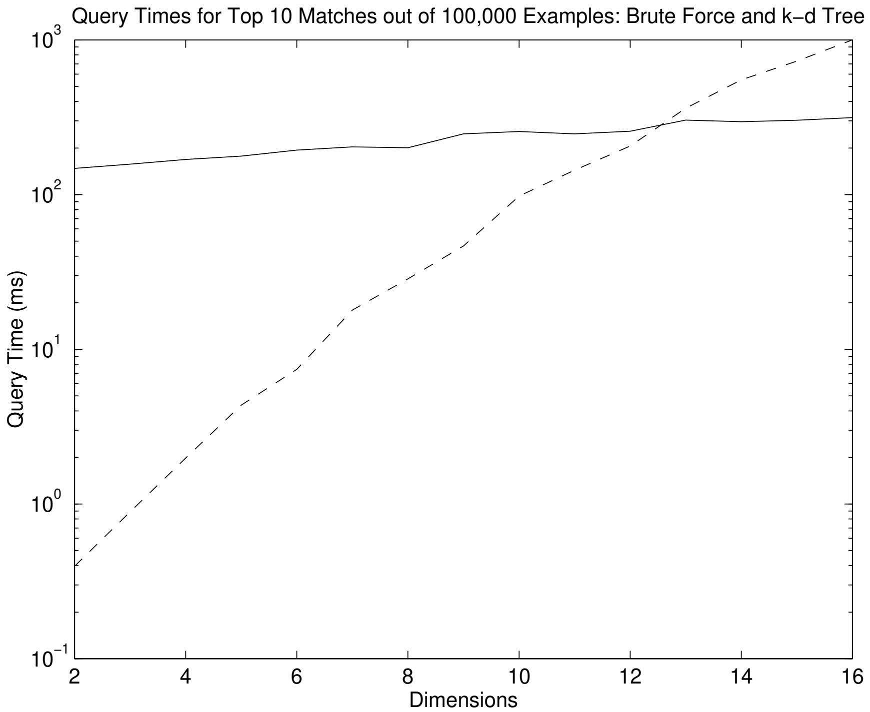

Query Times for Top 10 Matches out of 100,000 Examples: Brute Force and k−d Tree
Query Time (ms)
Dimensions
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
2
4
6
8
10
12
14
16

Figure 5:

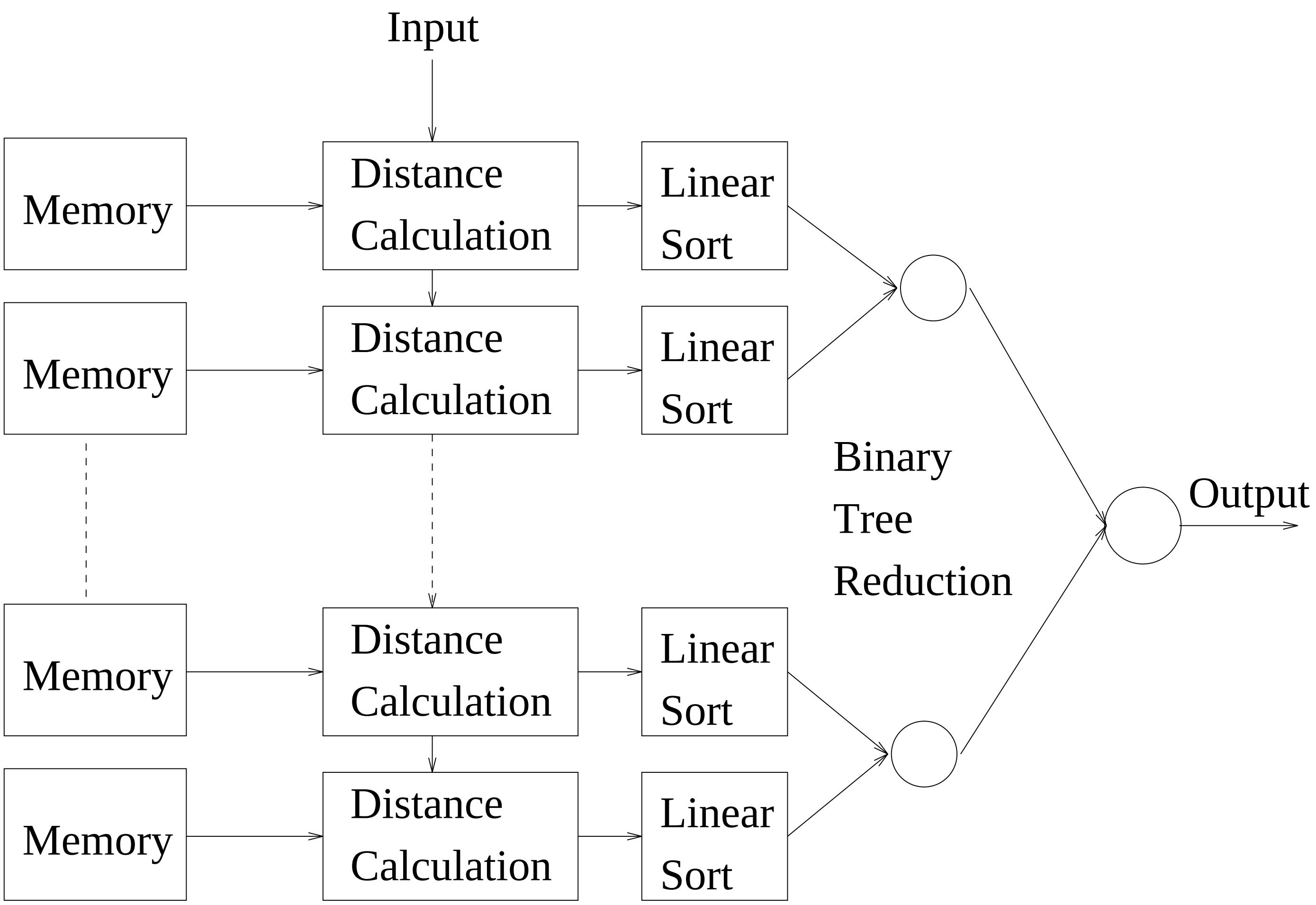


Figure 6: Parallel hardware implementation of a brute force search