Pablo Moscato

CeTAD
Universidad Nacional de La Plata
C.C. 75
1900, La Plata
ARGENTINA

moscato@cetad.edu.ar

Natalio Krasnogor

Centro de Técnicas Analógico - Digitales
Departamento de Electrotecnia
Universidad Nacional de La Plata
Calle 48 y 116
1900 La Plata
ARGENTINA

natk@cetad.edu.ar

ABSTRACT

Many problems currently addressed by using the so-called "Artificial Neural Network" systems can be viewed as "learning" problems in which the task is to perform a correct classification of a set of examples and induce a general rule from them. These problems often use the "learning-by-example paradigm" so they deal with databases in which information is coded as points in a multi-dimensional space.

For some learning heuristics there are no guarantees of performance, no computational complexity results and not even some average-case computational studies on its asymptotics, so a good behavior on a test done on a small test case would not infer that the approach is of any interest when the dimension of the database is large.

In this paper we examine the use of *Semidynamic Point Sets* and efficient data structures, that will be of particular interest when a polynomial-time learning algorithm is used with them. We make use of *K-d trees* which are binary search trees representing a set of points in K-dimensional space. They are very efficient and useful to perform certain types of searches like exact-match, partial-match and range queries as well as problems involving deletions, insertions and membership queries. A *convex hull* algorithm will provide us with a more detailed classification. We will also use a *BSP-tree* (*BSP* stands for Binary Space Partition) which is another useful data structure enabling us to correctly separate the resulting hyper-polygons. These techniques will be mainly discussed in the context of a recent algorithm for polynomial-time learning in neural networks but the performance improvement techniques disclosed would also find application in other classification algorithms.

In the so-called "learning-from-examples paradigm", a database of labeled examples, called S, is generally used to "train" an *adaptive* system, that is to fix the value of a set of parameters of that system. Following the notation of Ref. [3], the examples are pairs $(\vec{x}, t(\vec{x}))$ where $\vec{x}$ is an input vector and $t(\vec{x})$ is the value of an implicitly defined target function for the input $\vec{x}$. Thus the task is to inductively infer the function from a set of examples.

We must differentiate three cases, sometimes a "query" can be performed, that is we can consult an "oracle", meaning that we can retrieve the target value of a vector $\vec{x}_q$ which is not in S for all $\vec{x}_q$ in our domain of interest. On the other extreme, we may face situations in which there is no "oracle" at hand and we may only deal with the database we have. Between them there are situations in which the query can be performed but there is no guarantee of getting an answer.

If the system is an "Artificial Neural Network" (ANN) there are cases in which the learning problem (it would be better to say *"the 'loading' problem"*) is NP-Complete even for apparently "trivial" cases [8] [6] [9] [1]. The so-much advertised (so-often inefficiently implemented) Back-Propagation heuristic is suboptimal in many aspects and, being expensive in terms of CPU time, it is not expected to give good performances when learning large databases and to leave the "toy-examples domain" in which it has been mostly applied. Moreover, the time-complexity scaling of this kind of "learning-by-gradient-descent" schemes is poorly studied and it constitutes a serious impairment to any judicious researcher who would like to use it for any given application. In issues related with Computational Complexity, rigorousness is not just another formal exercise, it is a real practical issue as well.

On the other hand, the BP heuristic can be taken as an instance of these learning approaches which work comparatively well on few examples, and, regarding the advantages (both practical and theoretical [12]) of learning using queries, a valid subject of research is to see if we can reduce the computational complexity using particular types of data structures.

This paper is organized as follows: in section 2 we explain an existing polynomial-time algorithm for neural net learning and we introduce a new one, in sections 3 to 6 we present the data structures used in our algorithm and a discussion on their performance, in Section 7 we introduce an extension of the new algorithm, and in Section 8 some conclusions can be found.

2 A Neural-Net Algorithm for Polynomial-Time Learning

A good example of an algorithm which is proved to be polynomial-time in some cases is due to E.B. Baum [3]. In his algorithm (as described in Ref. [3]) it is implicitly assumed that the target function can be written as a feedforward net with n inputs completely connected to one hidden layer of k linear threshold units, and these are completely connected to the output linear threshold unit (or units).

Each of the hidden units corresponds to a hyperplane through the input space R^n which will be called *separating hyperplanes*. As an illustrative example, let's say that

Let $\vec{x}_+$ be a positive example and $\vec{x}_-$ a negative one. Let $\vec{x} = (\vec{x}_+ + \vec{x}_-)/2$ and query whether $\vec{x}$ is positive or negative. If positive, query the point halfway between $\vec{x}$ and $\vec{x}_-$ and so on. Repeating this process b times gives a point $\vec{y}_0$ on a separating hyperplane with b bits of precision.

Let $\vec{q} = \vec{y}_0 + \vec{p}_{rand}$ for $\vec{p}_{rand}$ a small random vector and query whether $\vec{q}$ is positive or negative. If (say) negative, query a point near $\vec{q}$ in the direction of $\vec{x}_+$. A few queries again suffice to give a point $\vec{p}$ on the separating hyperplane. We may establish rapidly that $\vec{y}_0$ and $\vec{p}$ lie on the same separating hyperplane, since if they do so do other points on the line which connects them. By repeating this process for several perturbations $\vec{p}_{rand}$, we may rapidly find k points on the same hyperplane, so the hyperplane is determined in this way.

Given a database S with $S_+ \cup S_-$ we have an associated collection C of pairs of examples such that $C = \{(s_+, s_-)\}$ $s_+ \in S_+$ and $s_- \in S_-$. Then a *"reasonable search procedure"* (we are quoting Baum in Ref. [3]) is to randomly call pairs from C without replacement and for each pair find a separating plane until (a) either we have found k distinct separating planes, in which case we know we are done or (b) we have exhausted all pairs in C. This seems to leave some space to a more efficient implementation since it is clear that we would not like to test all possible pairs.

Proceeding in this way, we search for the separating hyperplanes. If we find all k separating hyperplanes, we know the representation at a hidden layer in the target net of any input vector. Given a labelled database, we may discover a consistent set of weights at the output layer using Karmarkar's or the perceptron algorithms which will find a linear threshold function separating the hidden layer representations of the positive and negative examples. Another disadvantage of this method is that there are no guarantees of finding all the k separating hyperplanes [3].

We examine the use of semidynamic point sets and data structures in the context of a new MetaAlgorithm for polynomial time learning. The pseudocode of our MetaAlgorithm is as follows:

```
[0] BEGIN
[1] make a first classification of the examples using a K-d tree.
[2] for all the buckets generated with the K-d tree
    make a second classification of them using a BSP tree.
[3] with the planes found with the BSP tree use Karmarkar's
    algorithm to find an internal representation of them.
[4] create a neural network using the resulting planes
[5] DONE.
```

the extension of this procedure using convex hulls will be discussed later on.

3 Semidynamic Point Sets

It is then intuitive that the use of special data structures would give some benefit in connection with these methods for neural network training. By *Semidynamic Point Sets* we understand that an original set of N points (the database of examples, viewed as

operations, some points can be deleted from the set or undeleted but no new points can be inserted.

3.1 K-d Trees

A *K-d tree* is a binary search tree which represents a set of points in K-dimensional space and is useful to perform a certain type of searches. They include exact-match, partial-match and range queries. In particular they support two which are of interest for this discussion

- **Nearest Neighbor query**: Report the *nearest neighbor point* to a given query point, under a specified metric.
- **Fixed-radius near neighbor query**: Report *all points within a fixed radius distance* of the given query point, under a specified metric.

There are two types of nodes in a K-d tree, the *internal* and the *external* nodes. The internal nodes partition the space by a *cut plane* defined by a value in one of the K dimensions. The external nodes or *buckets* store the points in the resulting hyperrectangles of the partition. A node can be represented with the following C structure

```
typedef struct kdnode
{ int     bucket; /* boolean flag says whether a bucket or no */
  int     empty; /* boolean flag says whether empty or no */
  int     cutdim; /* indicates dimension of cut plane */
  long    cutval; /* indicates value at which cut happens */
  struct  kdnode  *loson; /* pointer to kdnode with values < cutval */
  struct  kdnode  *hison; /* pointer to kdnode with values > cutval */
  struct  kdnode  *father;/* pointer up the tree */
  KdItem  **permSeg;      /* If bucket, pointed into perm  */
  int     len;
  long    bnds[DIMENSIONS][2];
  int     lopt;           /* low part of permSeg  */
. int     hipt;           /* high part of permSeg */    } KdNode;
```

The variable `bucket` is one if the node is a bucket and zero if it is an internal node. In an internal node, `cutdim` is a value in that dimension and `loson` and `hison` are pointers to its two subtrees (containing, respectively, points not greater than or less than `cutval` in that dimension. A field called `empty` is added to each node in the tree. It is set to 1 if all points in the subtree have been deleted and is 0 otherwise. The search procedures take it into account and they are modified so as to return immediately when they visit an empty node. As the tree is being built, two pointers into `perm` represent a subset of the points. The tree is built by

```
for (i=0; i<n; i++) perm[i] = i; root = build(0, n-1);
```

The recursive function `build` is

```
if (u-l+1 <= cutoff) { p->bucket = 1; p->lopt = l; p->hipt = u;}
else { p->bucket = 0; p->cutdim = findmaxspread(l,u); m=(l+u)/2;
select(l, u, m, p->cutdim); p->cutval = px(m, p->cutdim);
p->loson = build(l,m); p->hison = build(m+1,u);  }
return p;}
```

The function `findmaxspread` returns the dimension with largest difference between minimum and maximum among the points in `perm`. The function `px` accesses the j-th coordinate of point `perm`. The function `select` permutes `perm`$[l..u]$ such that `perm`[m] contains a point that is not greater in the `p->cutdim`-th dimension than any point to its left, and is similarly not less than the points to its right. It's useful to remark that the recursive partitions will take place as long as $u - l + 1 \leq cutoff$. This means that when an internal node has a number of points which is less or equal than the $cutoff$, it becomes a bucket stopping the recursive call. Function `build` runs in $O(KNlogN)$ time.

4 Nearest-Neighbor Queries

The `nn` function (see below) computes the nearest neighbor to a point. The external (global) variables are used by the recursive procedure `rnn` which is in charge of doing the work. At a bucket node, `rnn` performs a sequential nearest neighbor search. At an internal node, the search first proceeds down the closer son, and then searches the farther son only if necessary. The function `x(i,j)` accesses the j-th dimension of point i. These functions are correct for any metric in which the difference between point coordinates in any single dimension does not exceed the metric distance (the Minkowski L_1, L_2, $\ldots L_\infty$ metrics all display this property).

```
int nn( int aPoint){
nntarget = aPoint; nndist = MAX_INT; rnn(root); return (nnptnum);}

void rnn(KdNode * p){
    if (p->bucket) {
       for (i=p->lopt;i<=p->hipt;i++)  {
          dist2=dist2(perm[i],nntarget);
          if (dist2<nndist2)  { nndist2=dist2; nnptnum=perm[i]; } } }
    else { val = p->cutval; thisx=x(nntarget,p->cutval);
           if (thisx<val) { rnn(p->loson);
                            if(thisx+nndist2>val)  rnn(p->hison); }
           else { rnn(p->hison); if (thisx-nndist2<val) rnn(p->loson);}}}
```

5 Deletions

To delete a node from the tree we first delete it from its bucket and then (as needed) turn on empty bits on the path to the root. To delete a point from a bucket we swap that point in the `perm` vector with the current `hipt` and decrement the latter. This

contains a `father` pointer, and the array element `bucketptr[i]` points to the bucket that contains node `i`. The deletion code is

```
void delete(int pointnum)
{ p = bucketptr(pointnum); j = p->lopt; while (perm[j] != pointnum) j++;
  permswap(j, (p->hipt)--);
  if (p->lopt > p->hipt) {
  p->empty = 1;
  while ((p = p->father) && p->loson->empty
                        && p->hison->empty )  p->empty = 1;}    }
```

Any single deletion requires at most $O(logN)$ time. Analogously, the `undelete` function performs a sequential search within the bucket, increments `hipt` and performs a `permswap` and then finally moves up the tree, turning off as many `empty` flags as necessary.

6 Performance Improvements

There exist several ways in which K-d tree performance can be improved, in Ref. [11] we can find a discussion on those methods. Let us discuss some improvements which must be taken under consideration:

6.1 Distance Caching:

We suppose we are given a certain point set (which can be a subset of an excising `point set`). In many applications it is very expensive to compute the distance function between to points of the set. There are $N(N-1)$ interpoint distances, so it is also very impractical to store them all. Let M be the smallest power of 2 greater than or equal to N; note that $M < 2N$. The cached distance function uses M integers and M floating point numbers:

```
int cachesig[m]; float cacheval[m]; float  dist(int i, int j);
{ if (i>j) swap(&i, &j); ind = i ^ j;
  if (cachesig[ind] != i) {cachesig[ind]=i; cacheval[ind]=computedist(i,j);}
  return cacheval[ind]; }
```

The values i and j are first normalized so that $i \leq j$, and then the value of i is XOR-ed with j to produce a *hash index ind*. The integer `cachesig[ind]` is a signature of the *(i,j)* pair represented in *cacheval[ind]* (because *ind*=*i* XOR *j*, we can recover j by *ind* XOR i). The cache algorithm therefore checks the signature entry, $cachesig[ind]$ and, if necessary, updates $cachesig[ind]$ and recomputes the value entry $cacheval[ind]$. It finally returns the correct value. This cache is very effective if the algorithm we are using already performs many local searches in its use of the distance function.

For some algorithms using the Euclidean metric, sometimes it is not necessary to compute the true distance (which is computationally expensive since it requires a square root calculation). It may be enough to compute the square of the distance. The code fragment shows how the square is sufficient at internal nodes:

```
diff = x(nntarget, dim) - p->cutval;
if (diff < 0) { rnn(p->loson);
if (nndist2 >= diff*diff) rnn(p->hison);} else {  rnn(p->hison);
if (nndist2 >= diff*diff) rnn(p->loson);}
```

where the variable `nndist2` represents the square of `nndist`.

6.3 Partition planes orientation

As described before, the partitioning planes are set orthogonal to the coordinate axis and can be located at the median of the distances of the point set. However, we have chosen to locate them at a coordinate value which is halfway between the first point to the left and the first to the right of the median found. Due to the rotational invariance of the L_2 metric distance, an interesting alternative in this case is to let the cutting planes have arbitrary orientations. To achieve that purpose, we must find the principal eigenvector of the covariance matrix of the point set. The two alternatives mentioned above have their own advantages and pitfalls. The perpendicular cutting planes are easily computed, but they are not so accurate when they are intended to partition a point set which has a non-uniform distribution. On the other hand, the principal eigenvector of the covariance matrix of the point set can give us a "fine tuning" of the cutting planes, but we must note that the cost of computing it is $O(k^3)$ where k is the number of dimensions. This said, we believe that for large k it becomes an unpractical approach. It is also remarkable that the cost of computing the distance from a point to a plane, using the second option, is more expensive than with the first choice. Regarding these situations, we conclude that the final election is application dependent. This means that, if we are developing an algorithm in which the time required to build the tree is irrelevant but we want a good time performance for the search, then K-d trees which use principal eigenvectors are recommended, because they will deliver a faster search. On the contrary, we may be interested in a fast building algorithm for the K-d tree at the expense of efficiency in the searching process.

Regarding the polynomial-time learning algorithm, we must choose alternatives which reduce the building time, since the K-d tree is intended to make a first classification of the examples. The K-d tree can be viewed as a first level classification. With that purpose in mind, we will use orthogonal cutting planes and the square of the true distance. Furthermore we will modify the `build` function in order not to continue partitioning an internal node if all points belong to the same class.

In what follows `sameclass` indicates a flag which tells if the points inside `permSeg` are all of the same class. On nonrecursive calls must be set to 0. `CutPlanes`: A hyperectangle including all the points in the tree (or subtree for recursive calls). On a nonrecursive call it must be set to the extremes of the (x, y) values found in all points. The first thing build does is to `malloc` a `kdnode`. Next it copies the `Cutplanes` it

arranges for the bucket pointers to point to the bucket and bottoms out of the recursion. Otherwise it calls `findmaxspread` which will find the dimension in which the elements of the `perm` are most spread out and will check if the points are all of the same class. If not, it splits the `perm` in half around `perm[m]` such that all cities `perm[i]` where $i < m$ have values in the chosen dimension which are greater than those where $i > m$. (Actually it just sorts them). Then it calls itself recursively: once on those elements of its section of the vector with index above m and once on those with index below m. It passes into the calls `CutPlanes` modified by the cut that has been made in the chosen dimension at element m. Finally it sets the pointers from its children to it and also arranges for the bucket pointers to point to the bucket and bottoms out of the recursion.

The resulting `build` function is

```
KdNode *build ( KdItem **permSeg, int len, int sameclass,
                long CutPlanes[DIMENSIONS][2] )

{ KdNode *p; int m,i; p = AllocNode(); CopyPlanes (CutPlanes, p->bnds);
  p->cutdim = findmaxspread(permSeg, len,&sameclass);
  if ((len <= KDCUTOFF) || sameclass) {
     p->bucket = 1; p->permSeg = permSeg; p->len = len;
     /* Bucket pointer allows access at the bottom of the tree */
     for (i=0; i<len;i++) {
        permSeg[i]->status=UNDELETED; permSeg[i]->bucket = p;
        CopyPlanes(CutPlanes,p->bnds); Buckets++;  } }
  else { p->bucket = 0; m = len/ 2; select(permSeg,len, p->cutdim);
  p->cutval = 0.5*(x(permSeg[m-1], p->cutdim)+x(permSeg[m], p->cutdim));
  p->minitem=*permSeg; p->maxitem=*(permSeg+len-1);
 CutPlanes [p->cutdim][HIBOUND] = p->cutval;
 p->loson = rbuild(permSeg, m,sameclass,CutPlanes);
 p->loson->father = p; CopyPlanes (p->bnds, CutPlanes);
 CutPlanes [p->cutdim][LOBOUND] = p->cutval;
 p->hison = rbuild(permSeg+m,len-m,sameclass,CutPlanes);
 p->hison->father = p; CopyPlanes (p->bnds, CutPlanes); }
  return p; }
```

After the K-d tree was finished, we have clustered the set of points regarding their classes and positions. Each bucket of the kdtree has all points belonging to the same class (up to the extreme case of only one point in the bucket). In Figs. 1 and 2 we can see how two different sets with many classes have been partitioned by the K-d tree (the cardinality of the sets and classes is small to simplify the drawings). Each point belongs to one class. In the drawing we can see how the K-d tree is adaptive in the sense that it finds more cutting planes in the areas where the classes approach each other. The K-d tree is capable of finding a good preliminary clustering.

The next step is to choose a set of planes which are sufficient to separate the classes since it is redundant to take all the K-d tree cut-planes. We will use a Binary Space Partitioning tree, (in what follows BSP-tree) as a second level classification. A BSP-tree node consist of a root polygon (in our case we have used the hyperrectangles

the root polygon and the other to those that are behind. We make a list of polygons (defined by the buckets' bounds) and from this list we generate a BSP-tree. Explicitly the code of the BSP-tree building function is:

```
BSP_tree BSP_makeTree(BSP_Lista l)
{ BSP_tree   aBSP_tree;   polygon   *root;       BSP_Lista *backList;
  BSP_Lista  *frontList;  polygon   *apolygon;   polygon   *backPart;
  polygon    *frontPart;
  if ( empty(l) || sameClass(l) ) { return(NULL); }
  else {allocate_polygon(&root);
  *root = *get_item(l,0); delete_item(l,0);
  emptyList(backList); emptyList(frontList); begin_iter(l);
  while (!end_iter(l))  {
             apolygon=get_iter(l);
             if (inFrontof(apolygon,root)) add_item(frontList,apolygon);
             else if (inBackof(apolygon,root) add_item(backList,apolygon);
             else { allocate_polygon(&frontPart);
                    allocate_polygon(&backPart);
             splitPoly(apolygon,root,&frontPart,&backPart);
                    add_item(frontList,frontPart);
                    add_item(backList,backPart);  }
     next_iter(l);    }
  return(combineTree(BSP_makeTree(*frontList),root,BSP_makeTree(*backList)));
}}
```

The reduction of the number of separating planes has an important effect in developing a final network architecture with (expected) better generalization properties. However, it must be said that the BSP-tree actually does not give the minimun number of planes which are needed, because the number of separating planes depends on the sequence of polygons we have chosen to be roots. It also depends on the number of partitions made during the construction. It is possible to improve the number of cutting planes using a simple heuristic. The idea is to sample the polygons and to select those which reduce the number of splittings. Only a small number of them are needed (typically 5 or 6). This strategy would give us a good BSP-tree.

7 Refined classification with a convex hull

The convex hull of a point set in Euclidean K-dimensional space is defined to be the smallest convex set containing that set of points (see Refs. [4] [7] and references therein).

The problem of finding the convex hull of a point set has captured the attention of many researchers who have already designed an abounding set of algorithms to compute it in polynomial time. Bentley (in Ref. [4]) described a linear algorithm for an *approximate* convex hull of a point set, and in [7] we can find a description of two parallelizable algorithms to find the exact convex hull in $O(NlogN)$ in the average case. In this paper, we will use an algorithm which does not pretend to be the current

The convex hull algorithm is separated into two main functions, `convex_hull` and `rconvex_hull`. The first one initializes the variables and finds the q most scattered points (note that q could only have values in the range [1,..,4]). With the q points already found it builds a convex polygon (a square or a triangle in the cases $q = 4$ or $q = 3$, a line or a point otherwise). For each edge of the figure it finds the points to the right of it (supposing we walk the polygon counterclockwise), thus it searches for the set of points which are outside of it. At the same time, when checking for the relative position of points to the edges, it memorizes which is the outer point of each side of the polygon. The `rconvex_hull` comes to scene when the original point set is divided into inter and outer points, the algorithm continues recursively dividing each edge into two, taking the original nodes and the outer point of each side, so long points remains to the left of some new side.

The Convex Hull Algorithm

```
List  convex_hull(List  point_set)
{  List extremes; List l1; List l2; List l3; List l4;
   Point  * p1; Point  * p2; Point  * p3; Point  * p4;
   extremes=find_extremes(point_set);
   p1=get_point(extremes,0);  /* minimum x point */
   p2=get_point(extremes,1);  /* minimum y point */
   p3=get_point(extremes,2);  /* maximum x point */
   p4=get_point(extremes,3);  /* maximum y point */
   l1=find_right_points(p1,p2,point_set);l2=find_right_points(p2,p3,point_set);
   l3=find_right_points(p3,p4,point_set);l4=find_right_points(p4,p1,point_set);
   extremes=rconvex_hull(p1,p2,l1,extremes); /* recursive calls with each */
   extremes=rconvex_hull(p2,p3,l2,extremes); /* side of the polygon */
   extremes=rconvex_hull(p3,p4,l3,extremes);
   extremes=rconvex_hull(p4,p1,l4,extremes);
   return(extremes); /* the convex hull list of points */}

List  rconvex_hull(Point * d1,Point * d2, List  l,List  extremes)
{
 Point  d,e; List l1; List l2;
 if (get_size(l)) { d=e=* find_farthest_point(d1,d2,l);
  extremes=add_point(extremes,&d); /* add the farthest point found to */
                                   /* the list of convex hull points  */
  l1=find_right_points(d1,&d,l); l2=find_right_points(&e,d2,l);
  extremes=rconvex_hull(d1,&d,l1,extremes);
  extremes=rconvex_hull(&e,d2,l2,extremes); }  return(extremes);}
```

Now that we are familiar with convex hulls we can explain their role. The K-d tree was used as a first level classification because it divides the space with orthogonal cutting planes trying to cluster, in a rough way, the classes it finds regarding the scattering of the points. After that, we make a second level classification using a BSP-tree to take from the K-d tree just a sufficient number of planes to correctly separate the classes. Once the BSP-tree has been built, we already have a correct

classes. In that way we have found a neural network which is capable of classifying the original multidimensional point set in polynomial time, since this *metaheuristic* is composed of at most polynomial time algorithms.

Once we got a net architecture and internal values, we can go ahead with the learning task using the convex hull, that is, we will take each generated kdtree's bucket. For the point inside each one we will compute the convex hull and delete the points which do not belong to it. In that way we have improved the first approximation made by the K-d tree and the BSP-tree taking, instead of the square hyperrectangles, the convex hull of each one. Then we can use every point in a convex hull as part of a reduced training set to run the ordinary backpropagation heuristic if we want more accuracy. It must be said that in general the number of points belonging to convex hulls generated by this method is (on average) very small in comparison with the cardinality of the original set. Furthermore, Kd-trees are capable of answer which are the points of one class which are closer to points of other classes using ballsearch and neighbor query. That kind of information can be used to ask to develop some kind of "oracle" regarding points which are halfway between different classes in order to adjust the separating planes.

8 Conclusions

In this paper we have discussed the use of Kd-trees and BSP-trees to do some preclassification of N-dimensional points in polynomial time. The use of these datastructures allows also some efficient implementations of queries such as the nearest neighbor to a point. With the bounds given by K-d trees and BSP trees we are able to find a good set of separating planes which can be used as inputs to Karmarkar's algorithm in order to find a representation of the planes using the "hidden" neurons (as in Ref[3]).

Acknowledgements

We feel particularly indebted to E. Amaldi, D.S. Johnson and J.A. Tierno for sending of re/preprints, which are of a value beyond estimation in these latitudes. We also want to express our gratitude to M.G. Norman for useful discussions during his stay in Argentina. P.M is supported by Comisión de Investigaciones Científicas de la Provincia de Buenos Aires, Argentina.

[1] E. Amaldi, On the complexity of training perceptrons, in: Proceedings of the 1991 International Conference on Artificial Neural Networks (ICANN '91), T. Kohonen, K. Mäkisara, O. Simula and J. Kangas, eds., (North-Holland, Amsterdam, 1991) pp. 55-60.

[2] E.B. Baum, What can Back Propagation and k-nearest neighbor learn with feasible sized sets of examples ?, in Neural Networks, EURASIP Workshop 1990 Proc., pp. 2-25.

[3] Neural net algorithms that learn in polynomial time from examples and queries, E.B. Baum, IEEE Transactions on Neural Networks, **2**, No. 1, Jan. 1991.

[4] J.L. Bentley, Approximation Algorithms for Convex Hulls Communications of the ACM No.1, Vol. 25, Jan 1982

[5] J.L. Bentley, K-d trees for semidynamics point sets, Sixth Annual Symposium on Computational Geometry, Berkeley CA, June 1990, pp. 187-197.

[6] A. Blum and R.L. Rivest, Training a 3-node neural networks is NP-Complete, in *Advances in Neural Information Processing Systems*, **1**, D.S. Touretsky, Ed. San Mateo, CA, Morgan Kaufmann, 1988, pp. 494-501.

[7] Z. Hussain A fast approximation to a convex hull Pattern Recognition Letters 8 (1988) 289-294

[8] J.S. Judd, Complexity of Connectionist Learning with Various Node Functions, Technical Report 87-60, University of Massachusetts, Amherst, MA, (1987).

[9] J.S. Judd, Neural network design and the complexity of learning, (MIT Press, Cambridge MA, 1990).

[10] P. Moscato, An Introduction to Population Approaches for Optimization and Hierarchical Objective Functions: The role of Tabu Search, Annals of Operations Research - Special Issue on Tabu Search, to appear (1993).

[11] R.F. Sproull, Refinements to nearest neighbor searching in K-dimensional trees Algorithmica **6** (1991) 579-589.

[12] L.G. Valiant, A theory of the learnable, Communications of the ACM, **27** (1984) 1134-1142.