On Error Probability of Search in High-Dimensional Binary Space with Scalar Neural Network Tree

Vladimir Kryzhanovsky[1], Magomed Malsagov[1], Juan Antonio Clares Tomas[2] and Irina Zhelavskaya[3]
[1]*Scientific Research Institute for System Analysis, Russian Academy of Sciences, Moscow, Russia*
[2]*Institute of secondary education: IES SANJE, Alcantarilla, Murcia, Spain*
[3]*Skolkovo Institute of Science and Technology, Moscow, Russia*

Keywords: Nearest Neighbor Search, Perceptron, Search Tree, High-Dimensional Space, Error Probability.

Abstract: The paper investigates SNN-tree algorithm that extends the binary search tree algorithm so that it can deal with distorted input vectors. Unlike the SNN-tree algorithm, popular methods (LSH, k-d tree, BBF-tree, spill-tree) stop working as the dimensionality of the space grows ($N > 1000$). The proposed algorithm works much faster than exhaustive search (26 times faster at N=10000). However, some errors may occur during the search. In this paper we managed to obtain an estimate of the upper bound on the error probability for SNN-tree algorithm. In case when the dimensionality of input vectors is $N \geq 500$ bits, the probability of error is so small ($P<10^{-15}$) that it can be neglected according to this estimate and experimental results. In fact, we can consider the proposed SNN-tree algorithm to be exact for high dimensionality ($N \geq 500$).

1 INTRODUCTION

The paper considers the problem of nearest-neighbor search in a high-dimensional ($N > 1000$) configuration space. The components of reference vectors take either +1 or -1 equiprobably, so the vectors are the same distance apart from each other and distributed evenly. We measure the distance between two points with the Hamming distance. In this case popular algorithms become either unreliable or computationally infeasible.

In (Kryzhanovsky, 2013) we investigated the following algorithms: k-dimensional trees (k-d trees) (Friedman, 1977), spill-trees (Ting, 2004), LSH (Locality-sensitive Hashing) (Indyk, 1998). We have found that k-d trees for $N > 100$ requires one or two orders of magnitude more computations than exhaustive search (BBF-trees (best bin first) (Beis, 1997) were used). As dimensionality N grows, the error probability of the LSH algorithm approximates one. In the event when the working point coincides with a reference, the spill-tree algorithm works faster than the exhaustive search (by an order of magnitude), but slower than the binary tree by approximately five orders of magnitude. The paper examines the case when the distance between query point and reference one is greater than $0.1N$. In these conditions the spill-tree algorithm is slower than the exhaustive search and thus its use makes no sense.

In (Kryzhanovsky, 2013) we offered a tree-like algorithm with perceptrons at tree nodes. Going down the tree is accompanied with the narrowing of the search area. The tree-walk continues until the stop criterion is satisfied. The algorithm works faster than the exhaustive search even when the dimensionality increases (for example, at $N = 2048$ it is 12 times faster).

In this paper we estimated the upper bound on the error probability of the algorithm. The error probability drops exponentially as the dimensionality of the problem N grows. For example, at $N \geq 500$ the error probability cannot be measured, i.e. the proposed algorithm can be considered exact in this range. Thus, the exact algorithm that excels exhaustive search in speed was obtained.

2 PROBLEM STATEMENT

The algorithm we offer tackles the following problem. Let there be M binary N-dimensional patterns:

$$\mathbf{X}_\mu \in R^N,\ x_{\mu i} = \{\pm 1\},\ \mu \in [1; M]. \qquad (1)$$

Kryzhanovsky V., Malsagov M., Clares Tomas J. and Zhelavskaya I..
On Error Probability of Search in High-Dimensional Binary Space with Scalar Neural Network Tree.
DOI: 10.5220/0005152003000305
In *Proceedings of the International Conference on Neural Computation Theory and Applications* (NCTA-2014), pages 300-305
ISBN: 978-989-758-054-3
Copyright © 2014 SCITEPRESS (Science and Technology Publications, Lda.)

A binary vector $\mathbf{X}$ is an input of the system. It is necessary to find any reference vector $\mathbf{X}_\mu$ belonging to a predefined vicinity of input vector $\mathbf{X}$. In mathematical terms the condition looks like:

$$\left|\mathbf{X}\mathbf{X}_\mu\right| \ge \left(1-2b_{\max}\right)N, \tag{2}$$

where $b_{\max} \in \left[0;0.5\right)$ is a predefined constant that determines the size of the vicinity.

We will show below that the algorithm solves a more complex problem from a statistical point of view: it can find the closest pattern to an input vector. The Hamming distance is used to determine the closeness of vectors.

In this paper we consider the case when reference vectors are bipolar vectors generated randomly. Generated independently of one another, the components of the reference vectors take +1 or -1 with equal probability (density coding).

3 THE POINT OF THE ALGORITHM

The idea of the algorithm is that the search area becomes consecutively smaller as we descend the tree. In the beginning the whole set of patterns is divided into two nonoverlapping subsets. A subset that may contain an input vector is picked using the procedure described below. The subset is divided into another two nonoverlapping subsets, and a subset that may contain the input vector is chosen again. The procedure continues until each subset consists of a single pattern. Then the input vector is associated with one of the remaining patterns using the same procedure.

The division of the space into subsets and the search for a set containing a particular vector can be quickly done using a simple perceptron with a "winner takes all" decision rule. Each set is controlled by a perceptron trained on the patterns of corresponding subset. Each output of the root perceptron points to a tree node of the next level. The perceptron of the descendant node is trained on a subset of patterns corresponding to one output of the root perceptron. The descent down a particular branch of the tree brings us to a pattern that can be regarded as a solution. At each stage of the descent we pick a branch that corresponds to the perceptron output with the highest signal. It is important to note that the same vector $\mathbf{X}$ is passed to each node rather than the result of work of the preceding-node perceptron.

4 THE PROCESS OF LEARNING

Each node of the tree is trained independently on its own subset of reference points. A root perceptron of the tree is trained on all M patterns. Each descendent of a root node is trained on $M/2$ patterns. The nodes of the *i*-th layer are trained on $M/2^{i-1}$ patterns, $i=1,2,\ldots k; k=\log_2 M$ is the number of layers in the tree.

All nodes have the same structure – a single-layer perceptron (Kryzhanovsky, 2010) that has N input bipolar neurons and two output neurons each of which takes one of the three values $y_i \in \{-1,0,+1\}$, $i=1,2$.

Let us consider the operation of one node using a root element as an example (all nodes are identical to each other). The Hebb rule is used to train the perceptron:

$$\hat{\mathbf{W}} = \sum_{\mu=1}^{M} \mathbf{Y}_\mu \mathbf{X}_\mu^T, \tag{3}$$

where $\hat{\mathbf{W}}$ is a $2\times N$-matrix of synaptic coefficients, and $\mathbf{Y}_\mu$ is a two-dimensional vector that defines the required response of the perceptron to the μ-th reference vector $\mathbf{X}_\mu$. $\mathbf{Y}_\mu$ may take one of the following combinations of values: (-1,0), (+1,0), (0,-1), and (0,+1). If the first component of $\mathbf{Y}_\mu$ is nonzero, the reference vector $\mathbf{X}_\mu$ is assigned to the left branch. Otherwise, it is assigned to the right branch. Since the patterns are generated randomly (and therefore distributed evenly), the way they are divided into subsets is not important. The set of patterns is always divided into two equal portions corresponding to the left and right branches of the tree during the training so that the four possible values of $\mathbf{Y}_\mu$ should be distributed evenly among all patterns, i.e. $\sum_{\mu=1}^{M} \mathbf{Y}_\mu = 0$.

The perceptron works in the following way. The signal on output neurons is first calculated:

$$\mathbf{h} = \hat{\mathbf{W}}\mathbf{X}. \tag{4}$$

Then the "winner takes all" criterion is used: a component of vector $\mathbf{h}$ with the largest absolute value is determined. If it is the first component, the reference vector should be sought for in the left branch, otherwise in the right branch.

The number of operations needed to train the whole tree is

$$\Theta = 2MN\log_2 M. \tag{5}$$

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

5 THE SEARCH ALGORITHM

Before we start describing the search algorithm, we should introduce a few notions concerning the algorithm.

Pool of losers. When vector **X** is presented to a perceptron, it produces certain signals at the outputs. An output that gives the largest signal is regarded as a winner, the others as losers. The pool of losers keeps the value of the output-loser and the location of the corresponding node.

Pool of responses. After the algorithm comes to a solution (tree leaf), the number of a pattern associated with the leaf and the value of the output signal of a perceptron corresponding to the solution are stored in the pool of responses. So each pattern has its leaf in the tree.

Search stopping criterion. If the algorithm comes to a tree leaf and the signal amplitude becomes greater than a threshold value, the search stops. It means that condition (2) holds.

Location of a node is a unique identifier of the node.

Descending the tree is going down from one node to another until the leaf is reached. The branching algorithm is as follows:

1. The input neurons of a perceptron associated with a current tree node are initiated by input vector $\mathbf{X}$. Output signals of the perceptron h_L and h_R are calculated.
2. The output with the highest signal and the descendent node related to this output (descendent-winner) are determined. The signal value of the loser output and location of the corresponding descendent-node are stored in the pool of losers.
3. If a tree leaf is reached, go to step 5, otherwise to step 4.
4. Steps 1 to 4 are repeated for the descendent-winner.
5. The result is put in the pool of responses. At this point the branching algorithm stops.

Now we can formulate our algorithm. Process of descending different tree branches is repeated until the stopping criterion is met. The stages of the algorithm are:

1. We descend the tree from the root node to a leaf. The pool of losers and pool of responses are filled in during the process.
2. We check the stopping criterion (2) for the leaf, i.e. we check if the scalar product of vector $\mathbf{X}$ and the pattern related to the leaf is greater than a predefined threshold. If the criterion is met, we go to step 4, otherwise to step 3.
3. If the criterion fails, we pick a node with the highest signal amplitude from the pool of losers and repeat steps 1 to 3 starting the descend from this node now.
4. We pick a pattern with the highest signal value in the pool of responses, and regard it as a solution.

6 EXAMPLE OF THE ALGORITHM OPERATION

Let us exemplify the operation of the algorithm. Figure 1 shows a step-by-step illustration of the algorithm for a tree built for eight patterns $(M=8)$. Step 1: the tree root (node 0) receives input vector $\mathbf{X}$. The root perceptron generates signals h_L and h_R at its outputs. Let $|h_L|>|h_R|$, then h_R and the location of the descendant-node connected to the right output (node 2) are placed in the pool of losers. Step 2: vector $\mathbf{X}$ is fed to the node-winner (node 1). A winning node is determined again and the loser is put in the pool (e.g. h_{LL} and node 3). Step 3: after reaching the leaves, we put patterns ($\mathbf{X}_3$ and $\mathbf{X}_4$) associated with the leaves and signal values $h_{LRL}=\mathbf{XX}_3$ and $h_{LRR}=\mathbf{XX}_4$ in the pool of responses. Then we check if the patterns meet criterion (2). In our case the criterion is not met, so the algorithm continues its work. Step 4: if none of the patterns satisfies the solution, we pick the highest-signal node from the pool of losers (for example, node 2

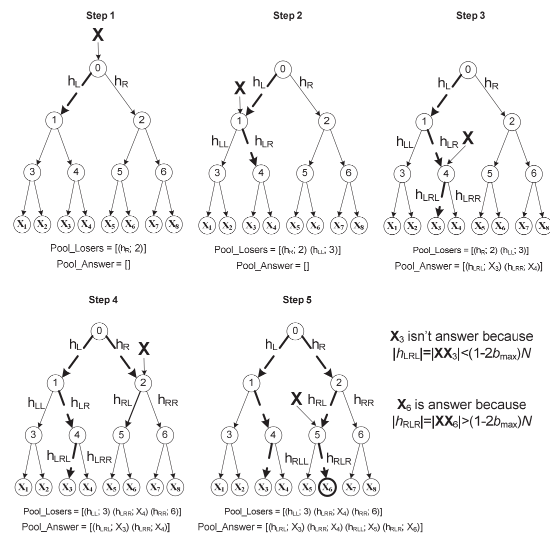

Figure 1: An example of the algorithm operation.

with signal h_R). Step 5: now the descent starts from this node (node 2) and continues until we reach the leaves while the pool of losers taking new elements. At that pair $(h_R;2)$ is moved away from the pool of losers. Here $|h_{RLL}|>|h_{RLR}|$ and $(1-2b_{\max})N<|h_{RLR}|$, i.e. criterion (2) is true for pattern $\mathbf{X}_6$. The pattern becomes the winner and the algorithm stops. If the criterion never works during the operation of the algorithm, the pattern from the pool of responses with the highest signal value is regarded as winner.

7 ESTIMATION OF THE ERROR PROBABILITY

It is hard to obtain a precise estimate of the error probability for the proposed algorithm as for now. However, it is possible to get its upper bound.

SNN-tree algorithm can fail in case when there is more than one pattern that satisfies criterion (2) in the set. Formally, the probability of this event can be written as:

$$P^* = 1 - \Pr\left[\bigcap_{m=1}^{M-1} |\mathbf{XX}_m| < (1-2b_{\max})N\right]. \qquad (6)$$

Presence of such patterns does not always lead to the algorithm failure. Therefore, probability (6) can be used as an estimation of the upper bound on the proposed algorithm failure.

Equation (6) can be calculated exactly by formula:

$$P^* = 1 - \left\{1 - \sum_{k=0}^{b_{\max}N} \frac{C_N^k}{2^{N-1}}\right\}^{M-1}. \qquad (7)$$

However, it is not possible to use formula (7) at large values of N (N>200). For high dimensions, it is better to use approximation:

$$P^* < \frac{2M}{\sqrt{2\pi\tilde{N}}}\exp\left(-\frac{\tilde{N}}{2}\right), \qquad \tilde{N} = N(1-2b_{\max})^2. \qquad (8)$$

Equation (8) shows that the error probability exponentially decreases as the problem dimensionality N grows. For example, at N=500 and b_{max}=0.3 power of the exponent is -40, which explains the fact that experimental error probability for high dimensions could not be measured in work (Kryzhanovsky, 2014). In fact, SNN-tree can be considered exact for high dimensional problems.

Figure 2 shows dependence of the error probability on dimensionality N at $b_{\max}$=0.3 and M=N. As

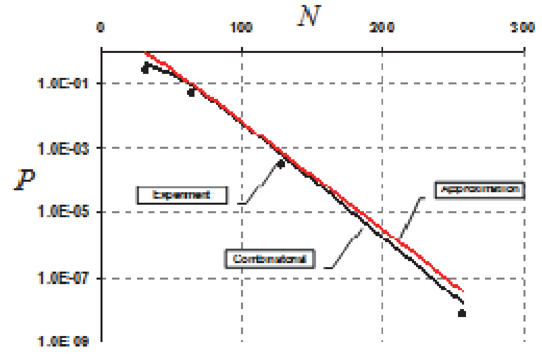

Figure 2: The algorithm error probability.

expected, the error probability of the algorithm (markers) is smaller than probabilities calculated using (7) and (8) (solid lines). Therefore, expressions (7) and (8) can be used for algorithm reliability estimation. Moreover, it can be seen that expression (8) is a sufficient approximation of (7).

8 ESTIMATION OF THE COMPUTATIONAL COMPLEXITY

Estimation of the proposed algorithm computational complexity is a quite sophisticated problem that was not solved yet. In this section, results of computational modeling are presented.

It was shown in work (Kryzhanovsky, 2014) that the problem in hand could be solved using only these two algorithms: exhaustive search and SNN-tree. Conducted research shows that the proposed algorithm works faster than exhaustive search, however it errors may occur. According to the results from the previous sections, the error probability at dimensionality N≥500 is so small that it can be neglected. Therefore, even a small speed advantage of SNN-tree over exhaustive search makes it preferable.

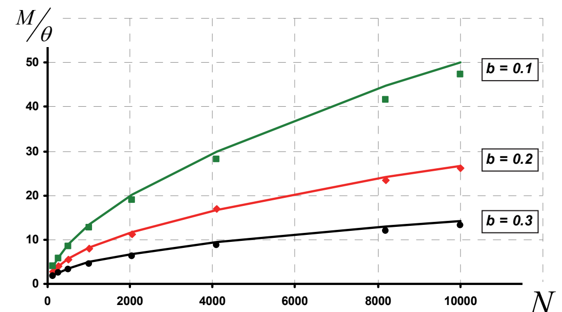

Figure 3: The speed advantage of SNN-tree over exhaustive search (markers - experiment; solid lines – estimation).

Experiments show (Fig. 3) that as dimensionality N grows the speed advantage of SNN-tree over exhaustive search increases. For example, at N=M=2 000 and b=0.2 SNN-tree is faster than exhaustive search in 12 times, and at N=M=10 000 acceleration reaches 26 times. Note b_{max} is a fixed value, but b is a number of distorted components in the input vector. Using experimental result, we estimated the average number of scalar product operations needed for SNN-tree search:

$$\theta = M \exp\left\{1.3 + b - \left(0.44 - 0.4b\right)\log_2 N\right\}. \qquad (9)$$

Solid lines in figure 3 were built using equation (9). This equation allows estimating average speed advantage of SNN-tree over exhaustive search. Using (9), it is possible to predict advantage of SNN-tree at large values of parameters (table 1).

Table 1: The speed advantage of SNN-tree over exhaustive search for M=N=10^5 and b_{max}=0.3 using equation (9).

b	M/θ
0.1	189
0.2	88
0.3	41

9 CONCLUSIONS

The paper considers the problem of nearest-neighbor search in a high-dimensional configuration space. The use of most popular methods (k-d tree, spill-tree, BBF-tree, LSH) proved to be inefficient in this case. We offered a tree-like algorithm that solves the given problem (SNN-tree).

In this work, theoretical estimate of the upper bound on the error probability of SNN-tree algorithm was obtained. This estimate shows that the error probability decreases as the dimensionality of the problem grows. Since even at N>500 the error is less than 10^{-15}, it does not seem possible to measure it experimentally. Therefore, it is safe to say that SNN-tree is an exact algorithm. Research investigations of the computational complexity of the algorithm shows that the speed advantage of SNN-tree algorithm over exhaustive search increases as the dimensionality N grows.

So, we can conclude that SNN-tree algorithm represents an efficient alternative to exhaustive search.

ACKNOWLEDGEMENTS

The research is supported by the Russian Foundation for Basic Research (grant 12-07-00295a).

REFERENCES

Friedman, J.H., Bentley, J.L. and Finkel, R.A., 1977.*An algorithm for finding best matches in logarithmic expected time.* ACM Transactions on Mathematical Software. vol. 3. pp. 209–226.

Ting Liu, Andrew W. Moore, Alexander Gray and Ke Yang., 2004. *An Investigation of Practical Approximate Nearest Neighbor Algorithms.* Proceeding of Conference. Neural Information Processing Systems.

Indyk, P. and Motwani, R., 1998. *Approximate nearest neighbors: Towards removing the curse of dimensionality.* In Proc. 30th STOC. pp. 604–613.

Beis, J.S. and Lowe, D.G., 1997. *Shape Indexing Using Approximate Nearest-Neighbor Search in High-Dimensional Spaces.* Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition. pp. 1000-1006.

Kryzhanovsky B., Kryzhanovskiy V., Litinskii. L., 2010. *Machine Learning in Vector Models of Neural Networks.* // Advances in Machine Learning II. Dedicated to the memory of Professor Ryszard S. Michalski. Koronacki, J., Ras, Z.W., Wierzchon, S.T. (et al.) (Eds.), Series "Studies in Computational Intelligence". Springer. SCI 263, pp. 427–443.

Kryzhanovsky V., Malsagov M., Tomas J.A.C., 2013. *Hierarchical Classifier: Based on Neural Networks Searching Tree with Iterative Traversal and Stop Criterion.* Optical Memory and Neural Networks (Information Optics). vol. 22. No. 4. pp. 217–223.

Kryzhanovsky V., Malsagov M., Zelavskaya I., Tomas J.A.C., 2014. *High-Dimensional Binary Pattern Classification by Scalar Neural Network Tree.* Proceedings of International Conference on Artificial Neural Networks. (in print).

APPENDIX A

It is necessary to calculate the following probability:

$$P^* = 1 - \Pr\left[\bigcap_{m=1}^{M-1} \left|\mathbf{XX}_m\right| < (1-2b_{\max})N\right]. \qquad (A1)$$

Let scalar products $\mathbf{XX}_m$ and $\mathbf{XX}_\mu$ be independent random quantities, $m \neq \mu$.

$$P^* = 1 - \prod_{m=1}^{M-1} \Pr\left[\left|\mathbf{XX}_m\right| < (1-2b_{\max})N\right]. \qquad (A2)$$

Now, it is necessary to calculate the probability that the product of each pattern by input vector is smaller than the threshold.

Scalar product $\mathbf{XX}_m$ is a discrete quantity, which values lie in $[-N;N]$. Let k be the number of components with the opposite sign in vectors $\mathbf{X}$ and $\mathbf{X}_m$. Then its probability function is:

$$\Pr\left[\mathbf{XX}_m = (N-2k)\right] = \frac{C_N^k}{2^N}. \quad \text{(A3)}$$

Random variable $\mathbf{XX}_m$ is symmetrically distributed with zero mean, so

$$\Pr\left[\left|\mathbf{XX}_m\right| < (1-2b_{\max})N\right] = 1 - 2\sum_{k=0}^{b_{\max}N} \frac{C_N^k}{2^N}. \quad \text{(A4)}$$

From A2 and A4 we can conclude that

$$P^* = 1 - \left\{1 - 2\sum_{k=0}^{b_{\max}N} \frac{C_N^k}{2^N}\right\}^{M-1}. \quad \text{(A5)}$$

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

APPENDIX B

Scalar product

$$\xi = \mathbf{XX}_m = \sum_{i=1}^{N} x_i x_{mi}. \quad \text{(B1)}$$

consists of a large number of random quantities. Therefore, at big dimensions (N>100) its distribution can be approximated by Gaussian law with the following probability moments:

$$\overline{\xi} = 0 \quad \text{и} \quad \sigma^2\left(\xi\right) = N. \quad \text{(B2)}$$

Therefore, probability (A1) can be described by integral expression:

$$P^* \sim 1 - \left\{1 - \frac{2}{\sqrt{2\pi N}} \int_{-\infty}^{-(1-2b_{\max})N} e^{-\frac{\xi^2}{2N}} d\xi\right\}^{M-1}. \quad \text{(B3)}$$

Using the following approximation

$$\int_x^{\infty} e^{-t^2} dt \approx \frac{e^{-x^2}}{2x},\ x \gg 1, \quad \text{(B4)}$$

obtain the final estimation of probability (A1.1):

$$P^* < \frac{2M}{\sqrt{2\pi\tilde{N}}} \exp\left(-\frac{\tilde{N}}{2}\right),\quad \tilde{N} = N(1-2b_{\max})^2. \quad \text{(B5)}$$