Neural Net Applications

Willard Miranker

Yale/DCS/TR1248

March 2003

Table of Contents

Using Genetic Algorithms to Achieve Better Performance with Feed Forward Networks and Backpropagration

Charles Coglianese
Yale University, Department of Computer Science
New Haven, CT 06520

Abstract

The power of training feed forward nets with backpropagation has been shown in many applications since its invention. However, there are still problems with this approach, including the local nature of the gradient computed and used in the weight-updating scheme. Genetic algorithms (GAs) have also been shown to be extremely useful for searching difficult spaces where little may be known except for an objective function to tell how good a potential solution is. As a result of the genetic operators and selection methods used in GAs, the search performed is global in the space of the objective function and thus does not suffer from the local minima that backpropagation training does. However, GAs tend to search many unnecessary spaces and thus can take a long time to converge due to the less directed nature of the search. The intuition, therefore, is to combine the local accuracy of backpropagation with the global ability of GAs to escape local minima to create neural nets that can produce more accurate answers, through less training. Thus, we use a modified GA with an extra step of training via backpropagation. Also novel to this study was the use of neurogenesis as a type of mutation, though this proved to have less of an impact than desired. The hypothesis turns out to be correct and is shown in the results of training nets on two different problems with two different stopping criteria.

Keywords – neural networks, genetics algorithms, backpropagation, neurogenesis

1 Introduction

1.1 Neural Networks and Training with Backpropagation

A neural network is a graph of connected nodes that interact through the weights between them to produce output useful in a particular domain. The difficulty in producing neural nets to solve different problems has to do with the difficulty of training the weights of the net to perform the desired task and also of designing the architecture of the net to have the right characteristics for the problem in question. One method for training the weights of a neural net is backpropagation, which performs a gradient descent to minimize the error between the output produced by the net and the desired output for the exemplar presented. One of the drawbacks of backprop is that it performs a local search and is thus susceptible to finding local minima that do not turn out to be the global minimum.

1.2 Genetic Algorithms and Their Application to Neural Networks

Another paradigm for searching function spaces for optimal solutions is genetic algorithms. Genetic algorithms are so-called because they are based on the principals of Darwinian evolution. The basic search algorithm is to create a population of random individuals (genotypes) and to evolve that population using genetic operators on certain members chosen due to their fitness as determined by an objective function. Once nice property of GAs is that the combination of genetic operators and selection methods allows the search performed to be global over the set of possible genotypes as opposed to the local weight updates calculated in backpropagation. Thus, the intuition behind this study is that by combining these two approaches in some way we can generate neural nets that find the global optimum of the search space in a shorter amount of time due to the population-based approach of GAs. A good introduction to the interactions between GAs and neural nets is provided in [3].

1.3 Testing Domains and Methodology

We tested this hypothesis on two different problems with two different stopping criteria. The first problem tested was the XOR problem. This problem was used to see how the algorithm would perform on a relatively simple task. The two different stopping criteria used here were two different tolerances for how good a net had to be to stop the search

and declare success. The difference between the two tests was how accurate the results produced by the net were in relation to the desired outputs from the problem. Thus, a less accurate stopping criteria would tell us how fast the algorithm was able to get decent results whereas a more accurate stopping criteria would show the performance of the algorithm in getting close to the actual global minimum as opposed to only finding a local minimum. The second problem tested was the Letter Classifier problem. For this problem I created a set of 8x8 bitmaps for each of the letters A through J. The task for the net to learn was to be able to fire one of ten outputs corresponding to the letter provided as input. I also took these trained nets and then tested their generalization capabilities on perturbed sets of the input to see how well the nets developed were able to generalize beyond the exemplars.

The algorithm presented here was tested against the exact same algorithm with the genetic operators removed. Thus, we used a population of nets and trained each with backprop until one of them met the required stopping criteria. Although this may seem unnecessary at first, it turned out to be necessary as a result of backprop's sensitivity to initial conditions. I will present a fuller explanation with experimental data later in the paper, but suffice it to say that a population's worth of nets was necessary to guarantee finding one that would reach the termination criteria.

2 Specifics of the algorithm

The basic format of the algorithm used in this study is provided in Figure 1. The idea is to combine the basic format of a general GA with some added training using backprop. Thus, the algorithm inserts a step of training the new population with backprop in between applying the genetic operators and calculating the fitness.

One of the hardest parts about designing a GA is figuring out what to use as a fitness function. Since the fitness function is what makes the algorithm go at a fundamental level, it is very important to pick a fitness function that correctly assess how good a particular member of the population is. For this study, I chose a very simple function. The fitness function takes the difference between the output produced for each exemplar and the desired output for that exemplar and sums them up, normalized over the number of exemplars. In other words, the fitness is just the error signal from the output

Start generate a random population of nets

Fitness evaluate the fitness of each net (sum of error values – smaller is better)

Selection preserve the top few fitness values using elitism, then run rank selection to select a set of nets for the next generation based on their fitnesses

Crossover with some probability, run double point crossover on a selected pair of parents by crossing over at a random level of each net

Mutation with some probability, mutate one weight in a selected net by a random factor at a random index value

Neurogenesis with some probability, replace one node of a selected net with a new node with all random weights

Backpropagation update the weights of the net using the backprop algorithm

Test if we have a net whose fitness is within some tolerance (or we've reached the max number of iterations), then stop, else go back to fitness stage

Figure 1: The algorithm used in this study.

layer of the neural net. Thus, in this representation we are trying to minimize fitness (error) as opposed to maximizing it, though it is somewhat irrelevant to the performance of the GA.

One key aspect of the algorithm here is the number of iterations through the backprop training algorithm we do during each iteration of the GA. The reason is that there is a balance between time and effectiveness with the training. If we use too many iterations, then the cost of computing the gradient for all the nets that don't get selected into the next round is too high and thus we can only run fewer iterations of the overall algorithm with the GA. However, if the number of iterations is too few, then the weights will not be sufficiently updated to cause a positive change in fitness and thus the backprop training will be ineffective.

Another key aspect of the algorithm has to do with the termination criteria used. In the tests for this algorithm, two different criteria were used of varying tolerance for error. The idea behind this was to test either how fast the algorithm could evolve nets of some acceptable fitness or to test if the algorithm could evolve nets of very high fitness. These two different stopping criteria produce very telling results about the effectiveness

of the algorithm. The next few sections discuss specific parts of the implementation in more detail.

2.1 Representations of Genotypes and Problems

One important choice in designing a genetic algorithm is the representation of the genotype. The genotype is the encoding of a potential solution to a problem. This genotype can then be evaluated for its fitness in the particular environment. In this case, the genotype is a specification of a neural net. More specifically, the genotype specifies how many layers there are in the net, how many nodes are in each layer, and what the weights are between each of the nodes. In the representation used, each node in one layer is connected to all nodes in the previous layer. In reality, many of these weights may be very close to zero and thus may be equivalent to there being no connection at all between two nodes. The representation is limited in the kinds of nets it can generate, however, in that nodes may only be connected to nodes in the following level.

Another interesting thing to note about the representations used here is that the characteristics of a particular problem domain determine how many input and output nodes there are in each net and thus the randomness is entirely in the characteristics of the hidden levels and the weights. This is a nice encapsulation of a problem since all we have to provide is the set of exemplars with the inputs and their desired outputs and we can use this algorithm to solve the problem. The reason is that the fitness, or objective, function is calculated simply by summing the distance from the desired output over all outputs. Thus, in this representation a low fitness value really means a better "fitness." This distinction is important to note when looking at the values produced by the algorithm, but makes no difference for the effectiveness of the algorithm versus using high fitness values as better "fitness."

2.2 Design Choices for the Genetic Algorithm

There are several other design choices that must be made in order to implement a GA that I have yet to discuss. Table 1 gives an idea of just how many customizable parameters there are in such an algorithm, let alone the actual design of the genetic algorithm.

Number of generations	200
Population size	100
Number of iterations over the exemplars of backprop training	10
Number of elite genotypes	10
Probability of mutation	0.25
Probability of a mutated node undergoing neurogenesis	0.25
Probability of crossover	0.1
Coefficient of the sigmoid function	1
Learning rate	4
Min number of hidden layers	1
Max number of hidden layers	3
Fitness tolerance for "good enough" answers	0.1
Fitness tolerance for "perfect" answers	0.01

Table 1: Parameters of the algorithm.

use a combined approach of using rank selection with elitism. After each the application of the genetic operators of crossover, mutation, neurogenesis (and in this case backpropagation training) the fitness of each net is computed and they are sorted according to their fitness. The rank of each net is just the position of the net in that sorted array, or, in other words, the rank a net is just the number of nets with better fitness than it. Rank selection means that each genotype has a probability of being selected in proportion to its rank, or:

$$\frac{N-i}{N*(N+1)/2}$$

where i is the rank of the genotype in question and N is the total number of nets in the population. Thus, nets with the lowest rank (highest fitness) have the best chance of being selected, while nets with lower fitness have a good chance of being selected due to the linear decrease in probability with decreasing rank. Elitism means that we take a fixed number, 10 in this case, of the best genotypes from each generation and copy them untouched into the next generation. These genotypes are still available to be selected by the rank selection method so that they may also be operated on by the various genetic operators. The combination between elitism and rank selection is a good one because elitism favors the current best genotypes whereas rank selection allows some of the lower ranked genotypes to be carried on into future generations. Thus, we keep around some

interesting genotypes that have low fitness while still keeping the current best from disappearing.

The other major part of the genetic algorithm is the genetic operators that we use. In this algorithm, we use the familiar operators of mutation and crossover and add a new operator modeled on another natural phenomenon, neurogenesis. For mutation, we choose a random node in the network and permute one of its weights by a random value. This value may be positive or negative and may make the weight either greater or smaller in magnitude. For crossover, we attempt to create new and different interesting architectures by crossing over two nets by choosing a random index into the vector of levels for each net and crossing over on either side of those two indices. I have used neurogenesis in this algorithm as a form of mutation as it is a fundamentally similar, though more drastic, operation. Neurogenesis is thus defined as taking the random node picked during mutation and replacing all of its weights with new random weights.

3 Results

The goal of this experiment was to test two basic hypotheses about combining GAs with backprop. The first is that doing this would allow us to produce nets within an acceptable tolerance faster. This is shown through tests with a higher error tolerance on the XOR problem and by looking at the convergence properties of each on the Letter Classifier problem. The second hypothesis is that using GAs together with backprop would allow us to produce nets of better fitness. This is shown through tests of the XOR problem with lower error tolerance and through tests of the Letter Classifier problem.

3.1 Getting Good Enough Results Faster

The results of trying to get reasonably good answers quickly are decidedly in favor of using GAs with backprop as opposed to just using backprop on its own. Table 2 shows the average results over 100 trials of the algorithm that uses the GA versus the algorithm that uses just backprop on the XOR problem when looking for fitness values under .1. The important thing to notice is that the number of iterations through the algorithm is much less for the algorithm using GAs as opposed to backprop alone. Thus, we are able to come to an acceptable solution in a much shorter period of time than if we had only

	GA and Backprop		Backprop alone	
	Iterations	Fitness	Iterations	Fitness
Average	8.48	0.072	23.61	0.097
Standard Deviation	20.38	0.019	3.95	0.002
Variance	415.54	3.79E-4	15.62	5.02E-6
Min	3	0.023	17	0.089
Max	200	0.140	50	0.100

Table 2: Good enough results for XOR (tolerance of .1 for fitness). The method that uses both GAs and backprop outperforms backprop alone, both in terms of the umber of iterations (time) to converge as well as in terms of fitness achieved. Notice the huge variance (i.e. unreliability) for the method using the GA, a result of the unpredictability added by the GA.

used backprop. Moreover, the average value is deceiving as evidenced by the high standard deviation. In fact, most solutions were found in 3 to 5 iterations with the GA versus 21 to 25 iterations for backprop alone (Figure 2 and Figure 3.) Another important thing to notice is that the average fitness produced is better when using GAs. Thus, in addition to finding solutions in fewer iterations, the algorithm also found nets with better fitness. Something else to notice about these results is the huge variance for the method that uses GAs. This is a result of the randomness added by the genetic operators and is worth noting as a downside to using GAs in that there are no guarantees about performance except for the average case.

It is important to make a note here about the method of comparing these algorithms. Even in the case where we were testing backpropagation alone, we used a population of nets. This is contrary to the usual use of backpropagation, however, without designing the architecture there was no other way to guarantee that we would generate a net that would find a solution at all. For example, we tested backpropagation training on an individual randomly generated net for 200 different trials and in only 20% of the cases did the net actually converge in under 50,000 iterations. Thus, the comparison here is between training a population of, say, 100 nets with either the GA and backprop or with just backprop. Each iteration, we sort the nets according to their fitness and stop if the best meets the stopping criteria (has good enough fitness.) Thus, although, the algorithm has more overhead than just using regular backprop on one net, we need to use it anyway since backprop with one net doesn't necessarily find a solution at all. This is thus the price of having a general algorithm that can take arbitrary problem descriptions.

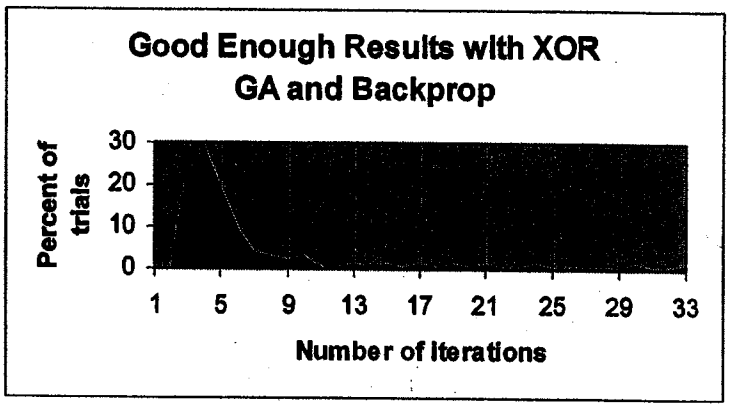


Figure 2: With both GA and Backprop, most trials converged in just a few iterations. The average number of trials needed is thus a bit deceiving since there are a handful of outliers that skew the result. These outliers are not to be overlooked, however, since they are evidence of the randomness introduced by using a GA.

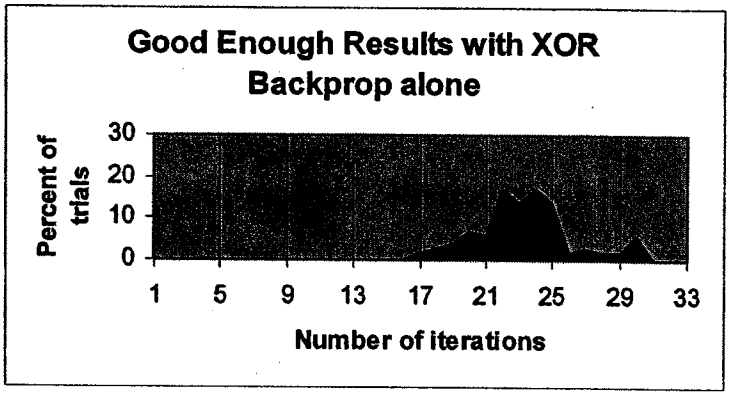


Figure 3: With just Backprop, most trials took much longer than with using the GA and the average is a good measure of performance given the small standard deviation.

Another interesting set of data to look at is a plot of how fast the two algorithms converged to their final answers on the Letter Classifier problem. In this case, the GA converged to within 1% of its final solution in only 20 iterations, whereas with backprop alone it took 200 iterations (Figure 4.) This shows that the GA is much more powerful in achieving good results fast than the algorithm that uses backprop alone.

3.2 Getting Better Results

The basic ideas for how to test to see which algorithm could get better results was to test on a simple problem with a low error tolerance and on a harder problem. The results again were conclusive in favor of the algorithm that used GAs in addition to backprop.

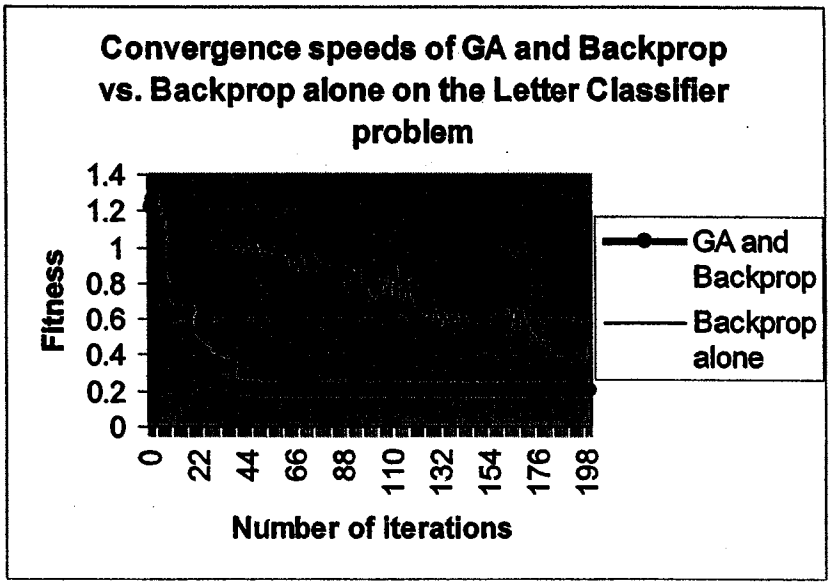


Figure 4: The convergence speeds of the two different algorithms on a difficult problem like the letter classifier problem. It is clear from the graph that the GA allows the algorithm not only to evolve nets of better fitness, but also to do so in many fewer iterations.

The first test was on the XOR problem with a tolerance of .01 for fitness. The results in Table 3 show that, while the algorithm using both GA and backprop is able to produce nets at this very low error tolerance in relatively few iterations, the backprop algorithm is unable to achieve this good of a fitness value at all in 200 iterations (except for on one occasion where it reached this fitness value on the 200th iteration.) Thus, the GA here helps us to find better solutions than backprop alone is even capable of finding, while at the same time doing it in relatively few iterations. This can be at least partly explained due to the fact that backprop is susceptible to getting stuck in local minima whereas this is not true for GAs as much due to the randomness of the genetic operators. The next test was on the Letter Classifier problem and again displays the power of the GA in finding better solutions. In this case, the algorithm using GAs and backprop was able to produce nets that were on average of significantly better fitness than those produced by backprop alone (Table 4.) This again shows the advantage of using GAs to help design neural nets, even on more complex problems.

3.3 Unexpected Results

One interesting result that I did not predict from the beginning was that GAs would be

	GA and Backprop		Backprop alone	
	Iterations	Fitness	Iterations	Fitness
Average	28.02	0.0046	200	0.0160
Standard Deviation	11.28	0.0005	0	0.0027
Variance	127.14	2.53E-07	0	7.38E-06
Min	11	0.0020	200	0.0074
Max	66	0.0050	200	0.0265

Table 3: Perfect results for XOR (tolerance of .01 for fitness). The method that uses both GA and Backprop outperforms Backprop alone by a huge amount when our goal is to produce nets with better accuracy. The results here show that using the GA allowed the nets to be evolved to the desired accuracy in a small number of iterations as compared to just using Backprop, which could only evolve one net out of 100 that could produce answers to the desired accuracy.

	GA and Backprop		Backprop alone	
	Iterations	Fitness	Iterations	Fitness
Average	200	0.34	200	0.47
Standard Deviation	0	0.12	0	0.11
Variance	0	0.015	0	0.012
Min	200	0.20	200	0.26
Max	200	0.50	200	0.66

Table 4: Results for Letter Classifier (tolerance of .1 for fitness). The method that uses both GA and Backprop outperforms Backprop alone with significantly better average fitness when our goal is to produce nets with better accuracy. The results here show that using the GA allowed the nets to be evolved to the desired accuracy in a small number of iterations as compared to just using Backprop, which could only evolve one net out of 100 that could produce answers to the desired accuracy.

helpful not only in allowing the algorithm to perform a global versus local search, but that they would also be helpful in fine-tuning the results produced by backprop. This is of course due to the ability of the mutation operator to "tweak" the weights towards better and better fitnesses. Thus, the fact that the GA helped in both aspects is one explanation for why it helped the overall algorithm so much. In the same vein, some testing showed that using neurogenesis as a form of mutation did not help the overall algorithm at all. This can be explained by the fact that the result of neurogenesis could be achieved by several mutations and thus it would not really be necessary or terribly helpful in achieving better nets.

Another interesting result came from testing nets produced by both algorithms on perturbed inputs from the letter classifier problem. The fitness of these nets on the inputs that had 10 or 20% of their bits flipped minus the original fitness was used as a measure of how well the nets generalize to noisy inputs. After several tests, it turned out that the nets produced by backprop alone had slightly better generalization ability when measured in this way. However, the fitness of nets produced by the GA and backprop on these

noisy inputs was still significantly better than the fitness of nets produced by backprop alone. Thus, the nets produced by GAs and backprop are still better, even with this as a metric.

The most interesting and unexpected result was that the crossover operator was not helpful in producing better nets faster. As it turns out this is a well-documented problem in the literature. In [4], the authors conclude that this may be the result of small population size. This doesn't seem to be the case here and it may be more as [1] suggests which is that crossover only works when there are basic building blocks to cross over. The location or even presence of these blocks with neural nets is unclear and thus finding a crossover operator that helps is extremely difficult. Recently, some have suggested the use of more complex schemes for determining which parts of two nets to cross over and it seems apparent from this study that this is worthwhile research. Another direction suggested in [2] is using a variety of mutation operators instead of using crossover at all. These operators change both connection weights and architectures and thus are an interesting choice as opposed to using crossover.

4 Conclusions

The best conclusion to be drawn from this is that using genetic algorithms to add a randomized, population-based approach to evolving and training neural nets is very beneficial. When combined with backpropagation, we get a high quality algorithm for evolving neural nets. This algorithm was shown here to evolve neural nets that produce better answers in a shorter period of time than by using just the backprop algorithm alone. The other main conclusion to take away from this work is that there are a huge number of design choices and parameters to deal with and so it can be difficult and time consuming to turn these ideas into a working implementation.

References

[1] X. Yao. 1999. *Evolving artificial neural networks*, Proceedings of the IEEE vol. 87, no. 9, p.1423–1447.

[2] X. Yao and Y. Liu. 1996. *A new evolutionary system for evolving artificial neural networks*, IEEE Transactions on Neural Networks, vol. 8, no. 3, p. 694–713, 1996.

[3] S. W. Wilson. 1990. *Perceptron redux: emergence of structure*, Physica D, vol. 42 p. 249–256.

[4] W. M. Spears and V. Anand. 1991. *A study of crossover operators in genetic programming*, Proceedings of the Sixth International Symposium on Methodologies for Intelligent Systems, p. 409–418.

[5] S. Haykin. 1999. *Neural Networks: A comprehensive foundation*, Prentice-Hall.

Pattern Searching and Labeling Using Self Organized Maps

Dan Andrei Iancu
Department of Computer Science
Yale University
New Haven, CT 06520

Abstract

We design a program that is capable of searching for certain small patterns embedded in a very large search space. This could be done with a classical search algorithm, but with enormous cost of computation. Neural Networks are very good at searching huge-sized spaces. Out of the many paradigms available, we have chosen to implement an auxiliary version of the Kohonen algorithm for self-organized maps. This choice has lead to an efficiency of about 70% in searching and labeling.

Key words: *Self Organized Map, Kohonen Algorithm, pattern, search space*

1. INTRODUCTION

Searching for small entities embedded in a huge space has applications in fields such as biology, genetics, and even astronomy. Decoding the DNA sequence of the human genome involves a continuous search for recurring patterns; a true breakthrough in this field would favorably impact diagnosis of Down's Syndrome, kidney disease, prostate and colorectal cancer, leukemia, hypertension, diabetes and atherosclerosis...

Such a search can help us understand how the human brain works. Discovering "maps" of the human brain (i.e. the way information is represented and where different stimuli are mapped in the brain) can inform the fields of Cognitive Science and Artificial Intelligence.

The task is to seek a proper way of implementing a search algorithm with Neural Networks, so as to avoid a computationally expensive classical search. We have adapted the powerful Kohonen Algorithm for self-organized maps to this particular search problem.

2. SELF ORGANIZED MAPS

The principle behind Self Organized Maps (SOM) is *competitive learning*; the output neurons of the network compete among themselves in order to fire, with the result that only *one* output neuron is on at a time.

In a SOM, the neurons are placed as the nodes of a *lattice* that is usually one- or two-dimensional. The neurons become selectively tuned to various input patterns (stimuli) or classes of input patterns in the course of a competitive learning process. The locations of the neurons so tuned become structured with respect to each other in such a way that a meaningful coordinate system for different input *features* is created over the lattice. [Haykin 1999]

Therefore, a SOM is characterized by the formation of a *topographic map* of the input patterns, in which *special locations of the neurons in the lattice are indicative of intrinsic statistical features contained in the input patterns*.

There are several possible models of SOM. The Kohonen model emerged as the most adequate and adaptable one for our applications. [Haykin 1999]

2.1 The Kohonen Model

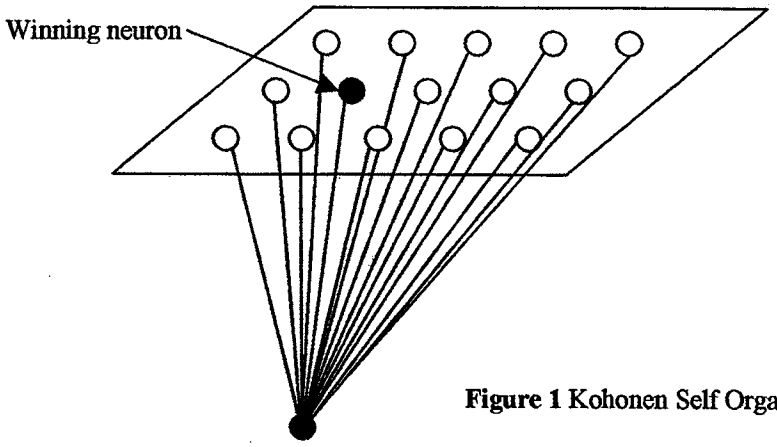


Figure 1 Kohonen Self Organized feature map

As can be seen in Figure 1, the output neurons are arranged in a two-dimensional lattice. This kind of topology specifies a set of neighbors for each neuron, with which it will later compete.

Each neuron in the lattice is fully connected to all the source nodes in the input layer (in the figure, only one node from the input layer is represented).

3. THE KOHONEN ALGORITHM

The *Kohonen algorithm* is specified by the following five steps:

1) Initialization

Choose random values for the weight vectors $w_j(0)$. The only restriction is that $w_j(0)$ be different for $j = 1,2,...,l$, where l is the number of neurons in the lattice.

2) Sampling

Present the network with a sample vector x from the input space. The dimension of x is equal to m (which is also the dimension of each of the weight vectors for the neurons in the lattice).

3) Similarity matching

Find the best-matching (winning) neuron $i(x)$ at time step n by using the minimum-distance Euclidean criterion (see figure 2):

$$i(x) = \arg\min_j \|x(n) - w_j\|, \quad j = 1,2,...,l$$

4) Updating

Adjust the synaptic weight vectors of all the neurons according to the formula:

$$w_j(n+1) = w_j(n) + \eta(n) h_{j,i(x)}(n)\left[x(n) - w_j(n)\right]$$

where:

$\eta(n) = \eta_0 e^{\frac{-n}{\tau_2}}$ is the *learning-rate parameter*

$h_{j,i(x)}(n) = e^{\frac{-d_{j,i}^2}{2\sigma^2(n)}}$, $n = 0,1,2,...,$ is the *neighborhood function* centered around the winning neuron $i(x)$

$\sigma(n) = \sigma_0 e^{\frac{-n}{\tau_1}}$, $n = 0,1,2,...,$ is the *width of the topological neighborhood function h*; τ_1, τ_2 are *time constants*

5) Continuation

Repeat starting with step 2 until no noticeable changes in the feature map are observed

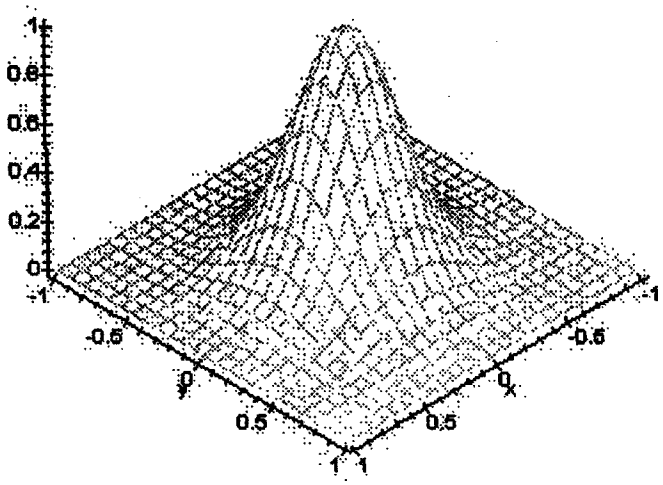


Figure 2 Plot of neuronal response, showing the winning neuron and its "neighborhood"

The Kohonen algorithm delivers a *feature map*, associating each input pattern with the weight vector of the firing neuron corresponding to that pattern (see Figure 3).

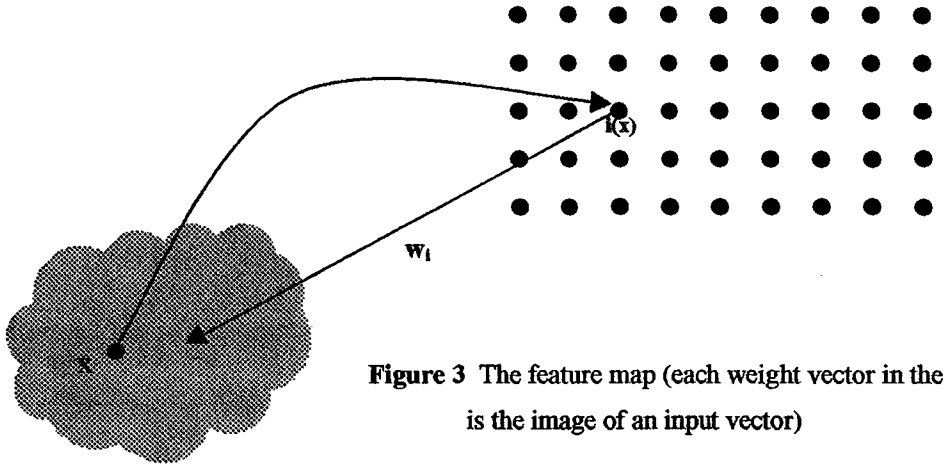


Figure 3 The feature map (each weight vector in the lattice is the image of an input vector)

a) ***The feature map represented by the set of synaptic weight vectors $\{w_j\}$ provides a good approximation of the input space***

b) ***It is topologically ordered (i.e. the spatial location of a neuron in the lattice corresponds to a particular domain or feature of input patterns)***

c) ***Feature selection – the SOM is able to select a set of best features for approximating the underlying distribution of the inputs***

We can sum up the algorithm as follows:

Represent the input space by the synaptic weight vectors w_j, in such a way that the map provides a faithful representation of the features that characterize the input vectors x in terms of certain criteria.

4. THE TASK

However, this algorithm does not match our search task a priori. We shall first formulate the task and describe the shortcomings and how they can be overcome in order to be able to implement the Kohonen algorithm.

The idea behind the program is to be able to search for small patterns embedded in bigger entities. The inputs are – however – of different sizes. First, we must specify the patterns that are to be searched – and those patterns are taken to be small matrices of 0's and 1's. Afterwards, we have to specify a search space, which is also a matrix of 0's and 1's, but of greater dimension. To make things clear, we give a simple example:

Suppose we want to look for a pattern like a *square* in the search space in Figure 4:

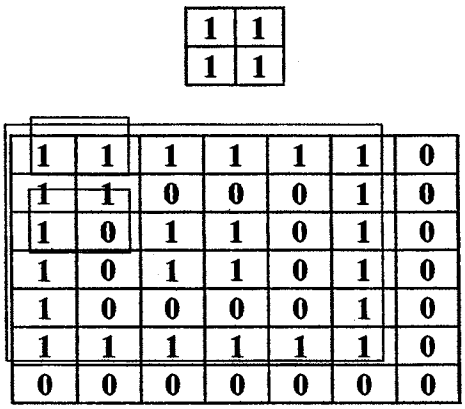


Figure 4 The *square* pattern and the *search space*

The pattern can obviously be found in the upper left corner and in the middle. But if we look more carefully, we notice that these are not the only squares in the space – there is a bigger square (of dimension 6). Therefore, we must match patterns the size of our given one, but also what we call *zoomed* versions of the pattern.

This is the main obstacle in matching this problem to the Kohonen algorithm: the inputs for the latter have to be of the same size (equal to the length of each weight vector in the lattice).

Therefore, the main idea in the program is the next one:

1) *Using a form of* ***heuristic****, augment the input pattern space by scaling all small input patterns, and train the network with the augmented set (Kohonen algorithm)*

2) *Using the trained net, input a new search space (big pattern), and find a positive match to one of the small patterns*

To find out where the pattern was found, we must examine the weight vector of the winning neuron – it contains the most relevant information about the findings. To perform repetitive searching on the same space, we can eliminate the pattern just-found, and repeat the whole procedure from the beginning. This may not be the best choice. (see **Section 6**).

5. THE EXPERIMENT

We train the net to recognize three distinct, basic shapes – square, triangle, and rhombus. The three shapes have been provided in three distinct files (see figure 9 in ***Appendix B***). The heuristic part of the program transformed a particular pattern into a file that was an adequate input for the training of the Kohonen SOM.[1]

There are four experiments. The first three involve a ***10x10*** search space, with a ***10x10*** lattice. The fourth experiment involves a ***20x20*** search space, with a ***10x10*** lattice.

Experiment # 1

The network has been trained to recognize all three patterns – ***squares, triangles*** and ***rhombuses***[2]. The parameters used in training are displayed in Table 1.

Parameters held constant: Search Space = 10x10; Lattice = 10x10; Lowest weight = 0.0; Highest weight 1.0; Learning rate (η_0) = 0.3; Initial neighborhood size (σ_0) = 4; Shrink rate ($1/\tau$) = 0.2 Minimum to_win = 0.0; Maximum_for_others = 1.0												
Number of training epochs	Types of ***tests performed*** and number of results ***right (R)*** or ***wrong (W)***											
	Pure ***square***			Pure ***triangle***			Pure ***rhombus***			Custom		
	R	W	%R	R	W	%R	R	W	%R	R	W	%R
500	195	90	68.4	145	140	50.8	13	107	10.8	8	17	32
750	157	128	55.0	182	103	63.8	35	85	29.1	4	21	16
1000	121	124	49.3	235	50	82.4	3	117	2.5	9	16	36
1500	162	123	56.8	205	80	71.9	40	80	33.3	10	15	40
2500	116	169	40.7	243	42	85.2	13	107	10.8	8	17	32
5000	139	146	48.7	175	110	61.4	61	59	50.8	11	14	44
7500	112	173	39.2	200	85	70.1	68	52	56.6	10	15	40
10000	111	174	38.9	203	82	71.2	66	54	55	11	14	44
20000	112	173	39.2	203	82	71.2	66	54	55	11	14	44
30000	112	173	39.2	203	82	71.2	66	54	55	11	14	44

Table 1 – Results from ***Experiment 1***

[1] For more information, refer to ***Appendix A*** – *Technical Specification*

[2] Refer to the final pages in ***Appendix B*** for the contents of the pattern files and of the files used in the custom tests

Interpretation:

We employed a 10x10 search space, and a 10x10 lattice of neurons. Each neuron in the lattice has a weight vector the size of the input space, that is 10x10 = 100.

We trained the network to recognize all ***squares, triangles*** and ***rhombuses***. After doing that, we have run certain tests, and the results are displayed in Table 1.

The network was requested to detect the patterns in several types of search spaces:

- files that contained ***only squares*** (column entitled "pure square")
- files that contained ***only triangles*** (column entitled "pure triangle")
- files that contained ***only rhombuses*** (column entitled "pure rhombus")
- custom files (contain none / one / two / three of the patterns)

We counted the correct/wrong answers in matching for different numbers of training epochs (column 1 in ***Table 2***). The number of correct answers is stored in the columns labeled ***R***, whereas the number of wrong guesses is in the ***W*** columns. The ***%R*** columns store the percent of guesses. The results delivered by the net are plotted in Figure 5.

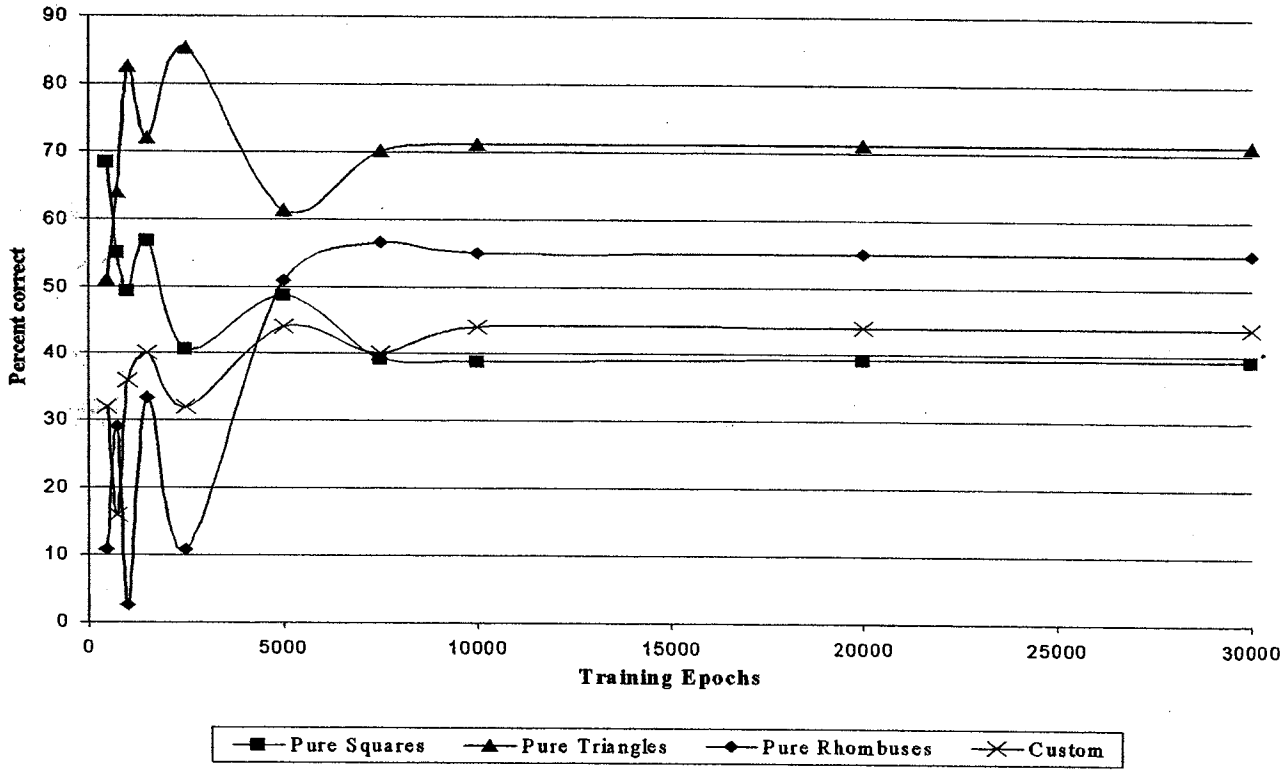


Figure 5 – Plotted results from ***Experiment 1***

As can be seen from Figure 5, the network "liked" triangles – the efficiency was quite high, reaching 85% at about 2500 epochs, and saturating at about 70%. However, the saturation efficiency of the network for the other cases was approximately 50%.

Experiment # 2

Parameters held constant: Search Space = 10x10; Lattice = 10x10; Lowest weight = 0.0; Highest weight 1.0; Learning rate (η_0) = 0.2; Initial neighborhood size (σ_0) = 5; Shrink rate ($1/\tau$) = 0.2 Minimum to_win = 0.0; Maximum_for_others = 1.0												
Number of training epochs	Types of ***tests performed*** and number of results ***right (R)*** or ***wrong (W)***											
	Pure ***square***			Pure ***triangle***			Pure ***rhombus***			Custom		
	R	W	%R	R	W	%R	R	W	%R	R	W	%R
500	179	106	62.8	154	131	54.0	29	91	24.1	7	18	28
750	136	149	47.7	225	60	78.9	14	106	11.6	10	15	40
1000	155	130	54.3	222	63	77.8	14	106	11.6	7	18	28
1500	166	119	58.2	184	101	64.5	42	78	35	9	16	36
2500	131	154	45.9	217	68	76.1	26	94	21.6	10	15	40
5000	128	157	44.9	202	83	70.8	47	73	39.1	11	14	44
7500	116	169	40.7	209	76	73.3	54	66	45	11	14	44
10000	127	158	44.5	205	80	71.9	56	64	46.6	11	14	44
20000	134	151	47.0	199	86	69.8	57	63	47.5	12	13	48
30000	134	151	47.0	199	86	69.8	57	63	47.5	12	13	48

Table 2 – Results from ***Experiment 2***

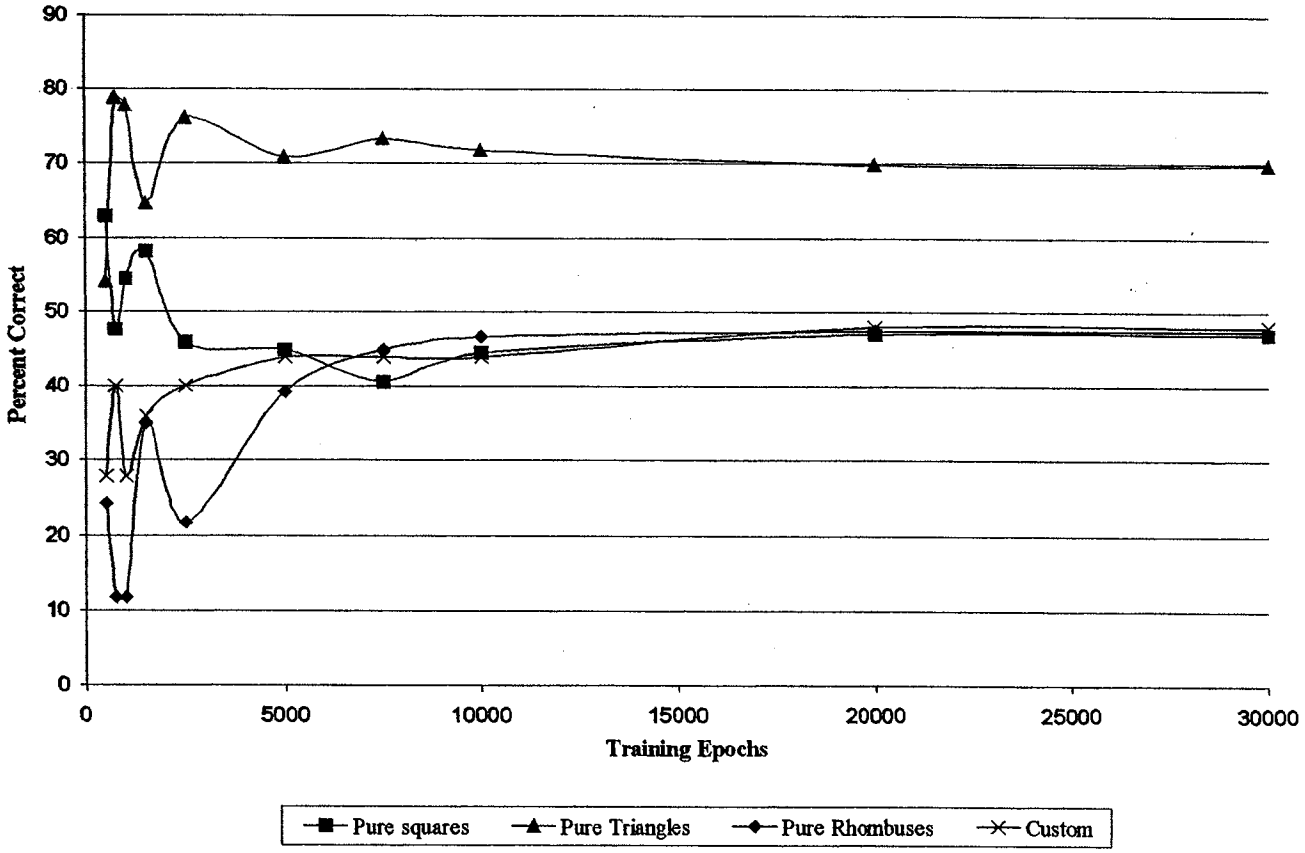


Figure 6 – Plotted results from ***Experiment 2***

Experiment # 3

Parameters held constant: Search Space = 10x10; Lattice = 10x10; Lowest weight = 0.0; Highest weight 1.0; Learning rate (η_0) = 0.2; Initial neighborhood size (σ_0) = 5; Shrink rate ($1/\tau$) = 0.1 Minimum to_win = 0.0; Maximum_for_others = 1.0												
Number of training epochs	Types of ***tests performed*** and number of results ***right (R)*** or ***wrong (W)***											
	Pure ***square***			Pure ***triangle***			Pure ***rhombus***			Custom		
	R	W	%R	R	W	%R	R	W	%R	R	W	%R
500	150	135	52.6	135	135	67.0	24	96	20	9	16	36
750	144	141	50.5	141	141	70.8	50	70	41.6	7	18	28
1000	106	79	57.2	79	79	82.4	21	99	17.5	7	18	28
1500	163	122	57.1	122	122	56.1	44	76	36.6	7	18	28
2500	119	166	41.7	166	166	81.0	23	97	19.1	8	17	32
5000	151	134	52.9	134	134	65.2	59	61	49.1	9	16	36
7500	120	165	42.1	165	165	72.6	60	60	50	7	18	28
10000	125	160	43.8	160	160	70.1	62	58	51.6	7	18	28
20000	129	156	45.2	156	156	68.7	60	60	50	8	17	32
30000	129	156	45.2	156	156	68.7	60	60	50	8	17	32

Table 3 – Results from ***Experiment 3***

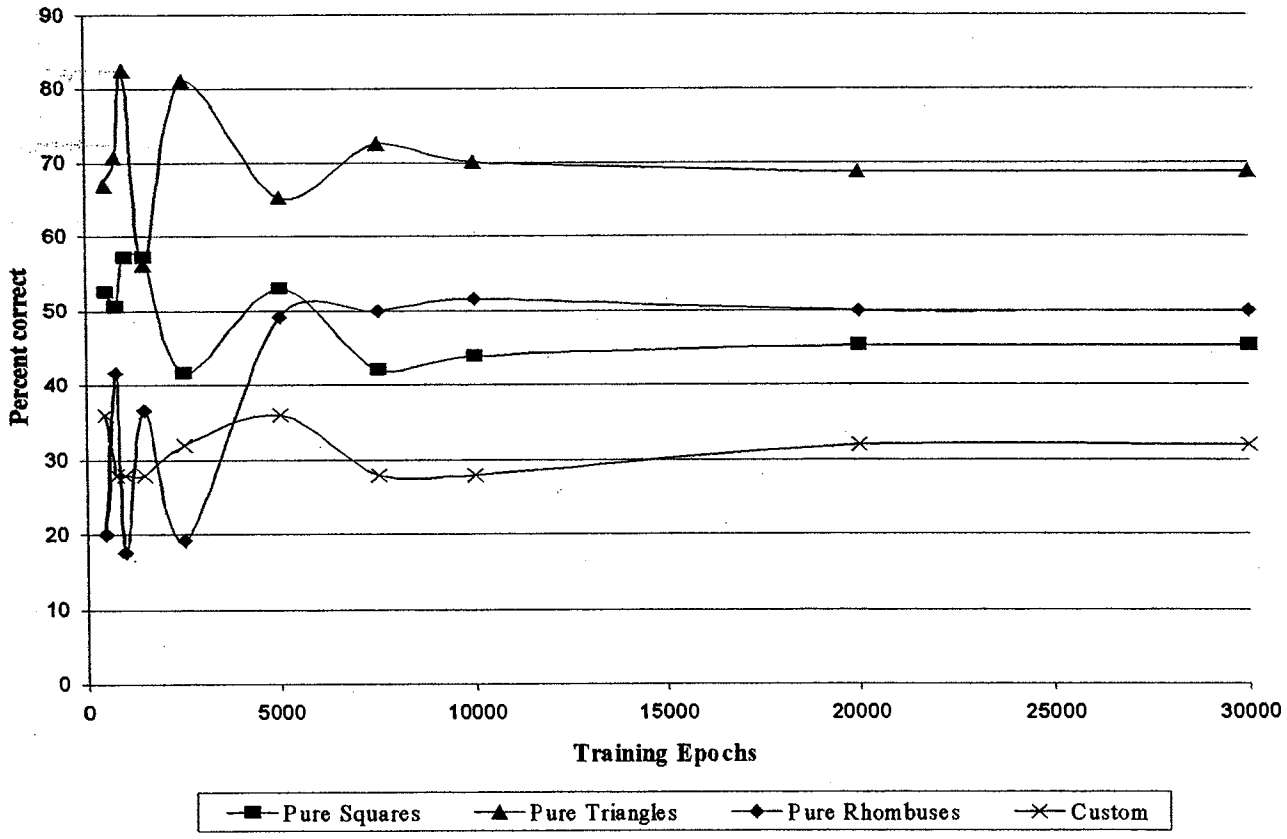


Figure 7 – Plotted results from ***Experiment 3***

For experiment 3, we changed the learning rate to 0.1. The small initial learning rate has had a negative influence on the performance of the SOM. The average efficiency of all cases was 47.14%.

Experiment # 4

The final experiment involved a 20x20 search space.

Parameters held constant: Search Space = 20x20; Lattice = 10x10; Lowest weight = 0.0; Highest weight 1.0; Learning rate (η_0) = 0.3; Initial neighborhood size (σ_0) = 4; Shrink rate ($1/\tau$) = 0.1 Minimum to_win = 0.0; Maximum_for_others = 1.0												
Number of training epochs	Types of ***tests performed*** and number of results ***right (R)*** or ***wrong (W)***											
	Pure ***square***			Pure ***triangle***			Pure ***rhombus***			Custom		
	R	W	%R	R	W	%R	R	W	%R	R	W	%R
500	728	1742	29.4	1895	575	76.7	113	1027	9.9	8	17	32
1000	917	1553	37.1	1862	608	75.3	***8***	***1132***	***0.7***	6	19	24
5000	2184	286	88.4	381	2089	15.4	223	917	19.5	7	18	28
10000	1973	497	79.8	925	1545	37.4	36	1104	3.1	7	18	28
20000	1148	1322	46.4	1597	873	64.6	97	1043	8.5	6	19	24
30000	1763	707	71.3	974	1496	39.4	359	781	31.4	4	21	16
40000	1298	1172	52.5	1713	757	69.3	53	1087	4.6	5	20	20
50000	1368	1102	55.3	1435	1035	58.0	593	547	52.0	8	17	32
60000	725	1745	29.3	2174	296	88.0	319	821	27.9	8	17	32
70000	1065	1405	43.1	1973	497	79.8	399	741	35	7	18	28
80000	753	1717	30.4	2176	294	88.0	431	709	37.8	6	19	24
90000	918	1552	37.1	2020	450	81.7	317	823	27.8	6	19	24
100000	1093	1377	44.2	1890	580	76.5	310	830	27.1	8	17	32
110000	1209	1261	48.9	1720	750	69.6	323	817	28.3	7	18	28
120000	1093	1377	44.2	1890	580	76.5	310	830	27.1	8	17	32
130000	1209	1261	48.9	1720	750	69.6	323	817	28.3	7	18	28
140000	1222	1248	49.4	1716	754	69.4	327	813	28.6	7	18	28
150000	1139	1331	46.1	1857	613	75.1	276	864	24.2	7	18	28
160000	1137	1333	46.0	1857	613	75.1	277	863	24.2	7	18	28
180000	1187	1283	48.0	1746	724	70.6	340	800	29.8	7	18	28
200000	1182	1288	47.8	1804	666	73.0	281	859	24.6	7	18	28
220000	1183	1287	47.8	1802	668	72.9	281	859	24.6	7	18	28
240000	1183	1287	47.8	1804	666	73.0	282	858	24.7	7	18	28
260000	1183	1287	47.8	1804	666	73.0	282	858	24.7	7	18	28

Table 4 – Results from ***Experiment 4***

This test turned out to be the most interesting and most computationally expensive. In the end, the network's average proficiency was of about 42.17% - the lowest of all the tests. Below, we have included a plot of the data and some explanations for the very low efficiency.

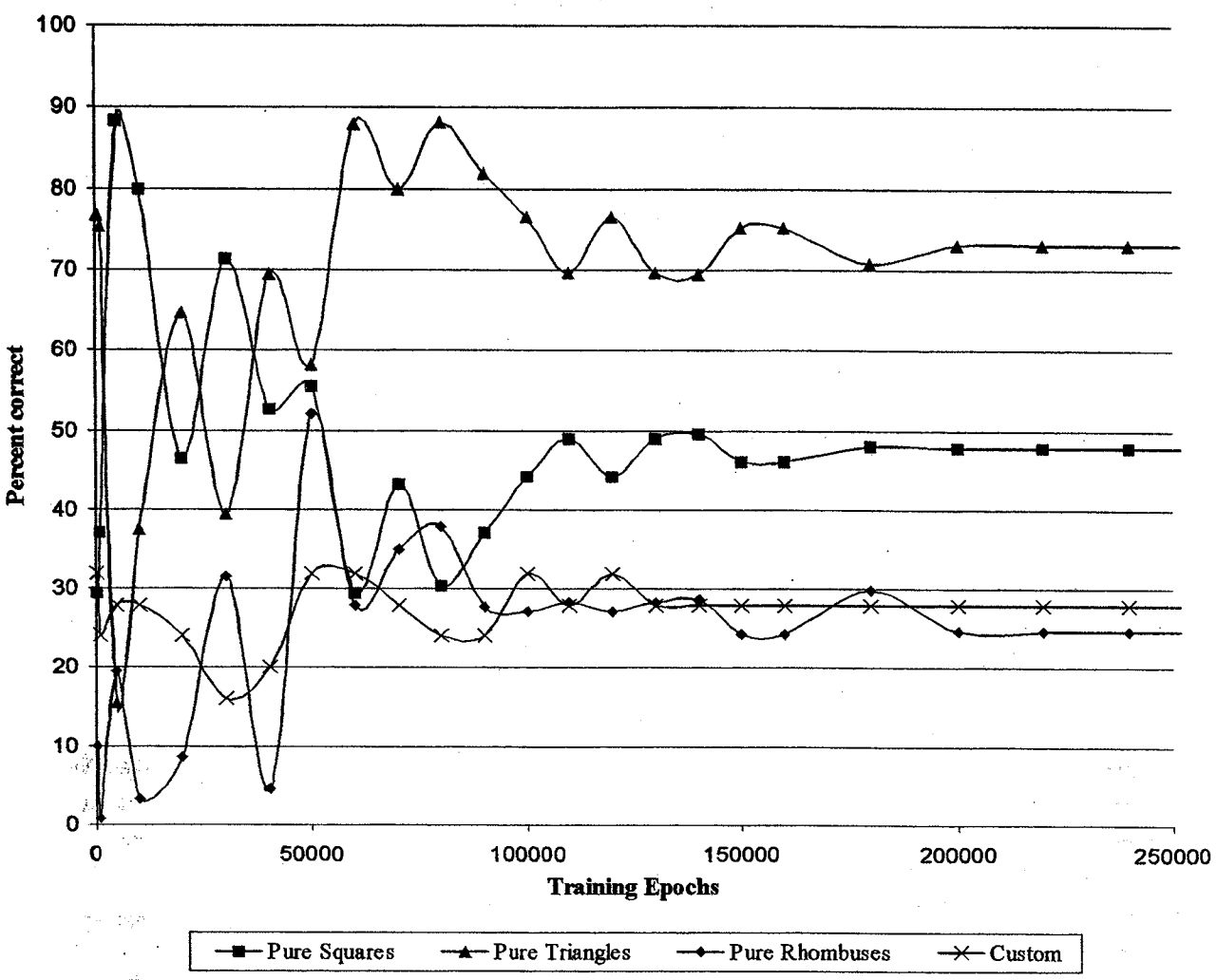


Figure 8 – Plotted results from ***Experiment 4***

First of all, the search space was much larger, and hence the possibility of error was also larger.

However – if we look at the data – we notice that the differences in performance for the three shapes were radically different. In all other tests, the differences were of about 10 to 20% (with small deviations), with the network somewhat better at triangles than at rhombuses or squares. The early stages of ***Experiment 4*** provided the biggest difference so far – when trained for 1000 cycles, the network managed to pass only ***8*** tests for rhombuses out of the ***1140*** total tests! That meant an efficiency of ***0.701%***! In fact, throughout all tests for rhombuses, the efficiency ***never*** surpassed 32%! There seems to be something else here that the network is unable to learn. We deter this question to future study, using this SOM approach.

It is now clear why despite the 65-85% efficiency in triangles, the average turned out to be only 40%! The problem was with rhombuses! An explanation for this deviation is provided in **Section 6**.

6. CONCLUSIONS

General conclusions:

- The Kohonen algorithm is a very powerful tool for implementing a Self Organized Map. It is remarkable how such an algorithm works – maps organize on their own, without exogenous guidance. An interesting aspect of the Kohonen Algorithm is that it can be considered a generalization of Hebb's rule – a winning neuron also affects its neighborhood, meaning that the neurons close to it have their synaptic strengths enhanced
- The network tends to reduce differences between efficiency in matching different patterns. If a pattern was found very *easily* in the beginning, more training leads to *lower* efficiencies. Similarly, a hard-to-find pattern gets easier to find, as the training proceeds. A **key conclusion** is that *if we are satisfied with the efficiency for a certain pattern in the early stages, we should stop the training, because the matching efficiency tends to drop.*
- More training meant more computational cost. But once the training was complete, the SOM was able to identify patterns **very quickly**.
- Changing the parameters for the net did not seem to have a significant influence on the final outcome. The overall differences in efficiency did not vary by more than 10%. However, it has become clearer that ***starting with too small a learning rate is not favorable*** for the final outcome (compare experiments 1 and 2).
- The network has behaved better when it had to recognize patterns in "pure" environments (meaning only one type of patterns in the environment).

Specific conclusions:

- The lattice handles some shapes more easily than others. In all tests, there has been a clear preference for the ***triangle***, which can be explained in many ways. As we have already noted, an SOM has the ability to retain certain "features" or "characteristics" of the input space. So to get some answers, we should also examine the topology of our patterns. One of the things that a ***square*** and a ***rhombus*** share in common (but our ***triangle lacks***) is point ***symmetry***. Hence, it is possible that the net has a propensity for lack of symmetry, disorder. However, this is only a speculation, based on a rather limited test case. It merits further consideration, perhaps by extending the use of the program to detect other different patterns (some symmetrical, some not).
- Perhaps another explanation involves the choice of the initial patterns. The square and the triangle were both 2 by 2 matrices (as patterns).[3] Therefore, in all the tests for squares, the network could have found a small triangle (in the upper right corner of the square!). This is why in many cases *good squares could have been confused with triangles*![4]
- Although this might go against common belief, some test cases have demonstrated that *"more teaching does not necessarily lead to better learning"*. Especially when learning to distinguish ***squares***, the network initially exhibits a high efficiency (in some tests, even 90%!). However – almost paradoxically – as the training proceeds, the efficiency decreases (in some cases quite drastically. It could be that a forgetting factor is needed, to correct this problem.

[3] Refer to the last page of ***Appendix B*** – *Data* to see the initial patterns

[4] This is yet another indicator of just how good the SOM is (detecting squares *despite* this confusion)

7. FUTURE WORK

The first and most important change is the replacement of the initial patterns. As pointed out in **Section 6**, any valid square (at any point in the experiment) could have been interpreted as a valid small triangle (the upper right corner). This can be easily accommodated, by placing as initial patterns 3 by 3 matrices (instead of 2 by 2) and simply ignoring all the 2 by 2 patterns.

If this is not satisfactory, we can simply train the net to recognize the patterns separately, and then search the same space twice (obviously, a more costly one also).

Appendix B contains a list of all the different parameters that can be individually set in the experiments. We have narrowed our study to variations with the number of epochs, and influences of certain parameters. We can imagine experiments in which we can study the variation related to the learning rate, or to the size of the initial neighborhood (with much larger test cases than the ones used here).

In addition to modifying the given parameters, we could insert a forgetting factor; by doing so, we might obtain better final results in the cases in which training decreases efficiency (as with triangles).

There is another change we could make – in the script file (***script***), which counts the erroneous identifications by the network. In some cases, the network does not find any match, and is simply "undecided". According to the current script, this is taken to be a mistake as significant as detecting the wrong pattern. However, we could adjust the "label-counting" into 3 categories: wrong decisions, right decisions and undecided (as do, in fact, most polls). We could also redesign the heuristics part of the algorithm, which might lead to better overall results.

References:

[Haykin 1999] - *Neural Networks, A Comprehensive Foundation.* Simon Haykin, 2nd edition, Prentice Hall, 1999.

Building Neural Networks, David M. Skapura, Addison-Wesley, 1996

http://www.helsinki.fi/~niskanen/tk/koho.html – Professor Teuvo Kohonen's Home Page

Cancer Classification and Genetic Markers Identification from DNA Microarray Expression Data

Taijiao Jiang[1] and Yin Liu[2]

[1]Department of Computer Science

[2]Biological and Biomedical Sciences Program

Yale University

New Haven, CT 06520

Abstract

Identification of genetic markers is important in increasing the accuracy of tumor classification that separates a population of patients into two or more diagnostic groups. Here we report on the successful application of feature selection method to a classification problem in biomedical science involving colon tumor DNA microarray data with only 62 data points in a 2000 dimensional space. Our approach is a distinct two-step feature selection approach. This approach incorporated the random forest method as the first step to reduce the high dimensional space of 2000 genes to less than 100 genes, and a genetic algorithm with different fitness functions such as decision tree and support vector machine to find a core subset of genes as the second step. Based on the above dataset, 3 to 5 features obtained in this combined method were good enough to achieve high colon tumor classification accuracy by using a multilayer feed forward neural net as the classifier. This small feature subset is thus called core feature subset, which might be regarded as the basic set of genetic markers effective for discrimination between colon cancer and normal tissue.

Keywords – genetic algorithms, tumor classification, feature selection, neural network

1. INTRODUCTION

Among the most powerful and versatile tools for functional genomic studies are high-density DNA microarrays. One of the most important applications of such a microarray is tumor classification [1], which distinguishes morphologically similar human cancers by the differences

between their expression profiles. The main advantage of microarray data is that it allows biologists to simultaneously monitor the expression of thousands of genes. However, a large number of genes increases the dimensionality, computational complexity and cost of data analysis, and introduces some undesired noise [2]. In the clinical setting of testing or implementing a set of prognostic markers, it would not be feasible to accurately measure and standardize measurement of an entire set in large numbers of patients [3]. So, the goal of this project is to identify an effective subset of markers from a large pool of potential markers. The selected subset will provide an optimal separation of a population of patients into two or more classes. Feature selection is a process of selecting an optimal subset of features from a possibly enormous set of potentially useful features, for use in classifiers. Using a smaller number of features may in fact improve accuracy in some contexts. A small feature set should generalize better beyond the whole data set, and may reflect the actual dominance of some key genes in cancer development, which may be potential drug targets.

Based on a genetic algorithm (GA) guided search (genetic search), we investigated and optimized several parameters (especially the fitness function, also called the evaluation function) of the GA for finding a core subset of genes (attributes or features) from DNA microarray data. We employ different types of evaluation functions. These functions include not only the popular evaluation function arising in the probabilistic approach[4], or the correlation approach (based on information gain, information gain ratio or chi squared test)[5], but also more sophisticated, advanced machine learning schemes, such as a neural network, a decision tree and a support vector machine[6-7]. Finally, the cancer classification performance of the selected features is assessed by a multilayer neural network as a classifier. We found that a set as small as 4 or 5 selected genes can achieve a higher cancer classification accuracy than the original 2000 genes, and therefore can be regarded as genetic markers of cancer tissues.

2. PROCEDURE

2.1. Data Set and Software Package

The data with expression profiles of 2,000 genes in 22 normal and 40 colon tumor cell lines is retrieved from the website www.sph.uth.tmc.edu/hgc.

The software we are exploring is the weka machine learning package:

www.cs.waikato.ac.nz/weka

2.2. Microarray Data Representation

Suppose we have a DNA microarray dataset with n instances ($n = n_N + n_T$, where n_N is the number of normal samples and n_T is the number of tumor samples. Each sample (also called instance) consists of m genes (features) with a certain expression value (or activity). Thus for tissue sample *i*, we have the vector $Y_i = [Y_{i1}, Y_{i2}, \ldots\ldots, Y_{im}]^T$. The Y_i's for normal (N) and tumor (T) samples constitute the following data matrix,

$Y_N = [Y_{N1}, Y_{N2}, \ldots\ldots, Y_{Nn_N}]^T (m \times n_N)$

$Y_T = [Y_{T1}, Y_{T2}, \ldots\ldots, Y_{Tn_T}]^T (m \times n_T)$

The total dataset is the matrix $Y = [Y_N\ Y_T]^T$.

Then our goal is to select a small subset of k features (genes) from m features ($k << m$) achieving high tumor classification accuracy.

2.3. Two-Steps Feature Selection

We propose to apply a genetic algorithm that uses crossover and mutation to find the subset of features. However, it is not very helpful to use a genetic algorithm from ab initio, because the initial dataset contains a large number of features. We propose a two-step feature selection method. The algorithm operates as follows:

TwoStepsFS(dataset data, FSRmethod fsr, evaluator eval)

1. FeatureSpaceReduction(data, fsr)
2. GeneticSearchForRefinement (data from 1, eval)
3. PerformanceEvaluation.

Principal Component Analysis (PCA) is probably the most popular method for feature space reduction. However, because PCA is an unsupervised learning technique, some researchers found that PCA does not take into account the class labels of the training set from microarray data, it is not reliable and does not generalize well [8]. Therefore, we employ the random forests as feature space reduction method. The algorithm is:

Random_forest(dataset data, evaluator decisionTreeAlgorithm)

Repeat n times:

Resampling the total instances from the data set with replacement

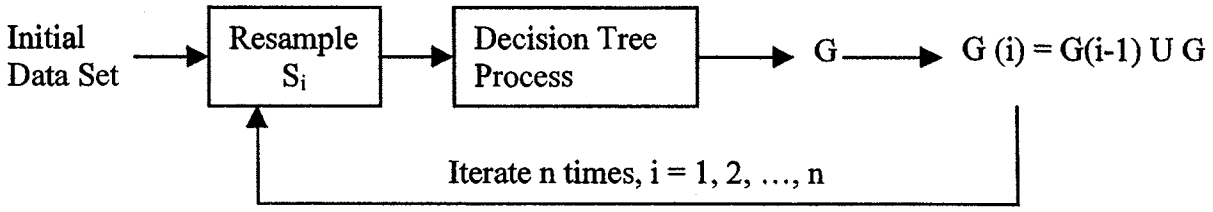
reorder the columns(features) randomly

Build a decision tree

Check the classification accuracy of the decision tree

Create new data set with only the features collected from above trees

The schema of this method is shown as follows:



Here G(0) = Ø.

After performing the function FeatureSpaceDeduction, we apply the genetic algorithm to further refine the selected features. The simple genetic algorithm is as follows:

```
initialize population;
evaluate population;
while TerminationCriteriaNotSatisfied  {
        select parents for reproduction;
        perform recombination and mutation;
        evaluate population;
}
```

Genetic algorithms have been used in many problem domains. The detailed steps in our approach are elaborated in Appendix D. There are many parameters to be explored for the particular problem domain at hand. We are interested in using the parameters in feature selection. The parameters include:

1. fitness function (evaluation function: using probabilistic approach such as Consistency Attribute Subset Evaluation(CSE)[4], or Correlation-based Feature Subset Selection such as CFS[5], or popular supervised machine schemes such as decision tree(Can be traced back to the information based evaluation), naïve Bayes, SVM and neural networks[6-7].
2. number of generations
3. probability of crossover and mutation

2.4. Evaluation of the selected features performance for classification

We created a feed forward multilayer neural net that used back-propagation learning algorithm [6] for performing classification based on the selected features. A commonly used statistical approach, 10-fold cross-validation [7] is used to evaluate the classification accuracy.

3. EXPERIMENT

3.1. Feature space reduction

After performing the random forest algorithm for feature space reduction, 20 out of 2000 features were returned (see Table 1). These attributes were collected from about twenty trees.

Gene No.	11	201	249	251	377	491	493	513	576	682
Gene ID	T72863	Z24727	M63391	U37012	Z50753	H4411	R87126	M22382	D14812	T51849
Gene No.	765	780	783	964	980	1042	1153	1423	1473	1671
Gene ID	M76378	H40095	R01755	T86473	U06698	R36977	R84411	J02854	R54097	M26383

Table 1. Features returned from the first step of feature reduction using the gene expression profile of 2,000 genes in 22 normal and 40 colon tumor cell lines. The data set is retrieved from the website www.sph.uth.tmc.edu/hgc.

3.2. Using different evaluation functions in the genetic algorithm

We tried different evaluation functions. A neural net (multilayer peceptron) is too computational intensive to be chosen as the evaluation function. For the small dataset with only 20 features (genes), genetic search employing 5000 generations with a neural net as the evaluation function had no indication to terminate after 24 hours. Even a small number of generations require hours to complete. Therefore, it is not feasible or practical to use this sophisticated computation model as the evaluation function. However, we shall use it as performance evaluator to assess the selected features, and also apply it to classify the tumor tissues based on the selected features. As shown in figure 1, for the CFS and CSE as the evaluation functions, the feature selection process employing 5000 generations terminates within 1 minute, while the process with a decision tree or SVM as evaluation function takes hours to complete. Interestingly, the classification accuracy doesn't increase with the number of generations for all the evaluation

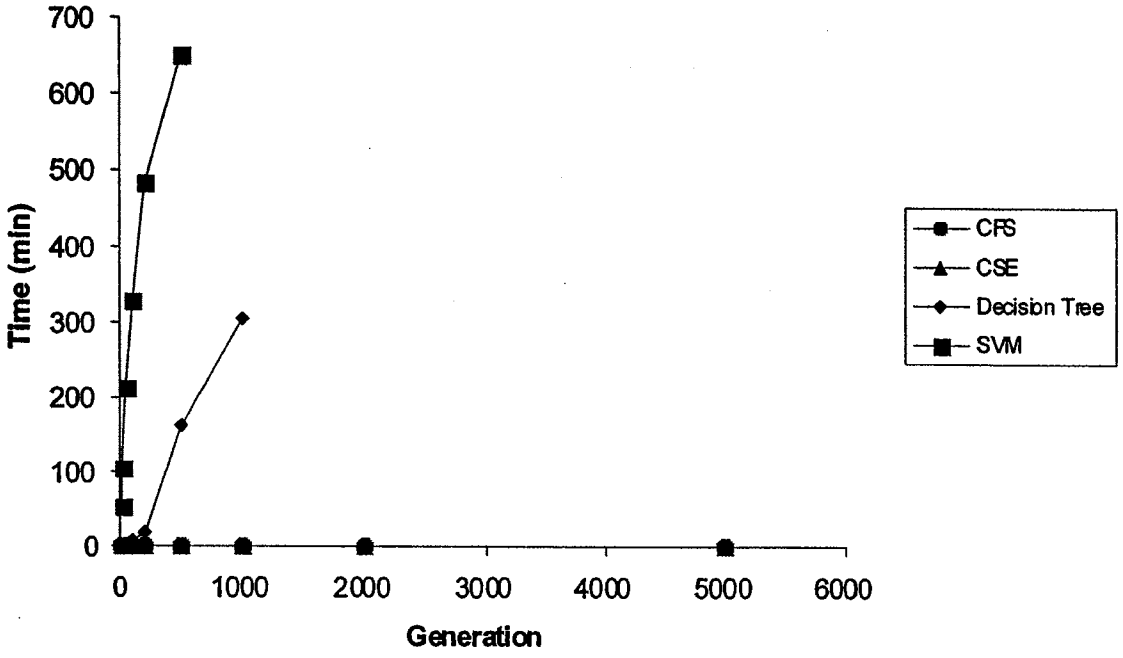


Fig.1. Time complexity with using different evaluation functions in genetic search

functions; instead, we found that all the evaluation functions converge very quickly and achieve the highest classification accuracy when the generation number is 20. More importantly, we found the decision tree achieves an accuracy of 87.1% with 4 selected genes (682, 765, 964, 1671) and SVM achieves a highest accuracy of 88.7% with 5 selected genes (377, 576, 682, 765, 1423). Therefore, we can conclude that the GA performs very well since it selects a small subset of genes and achieves high classification accuracy successfully. (Appendix A)

3.3. Different evaluation functions with different crossover and mutation rates

We fix the number of generations at 20. The probability of crossover ranges from 0.1 to 0.9 with a mutation probability of 0.06. (Fig.2 and Appendix B) In this experiment, high classification accuracy can be achieved when the crossover rate is as low as 0.1 with CFS selected as the evaluation function, and if CSE or decision tree is selected as the evaluation function, the highest accuracy is obtained when the crossover rate is 0.2 although the accuracy fluctuate when the crossover rate is higher than 0.3. Therefore, we conclude that in our approach the genetic algorithm tends to perform better at low rather than at high crossover rate.

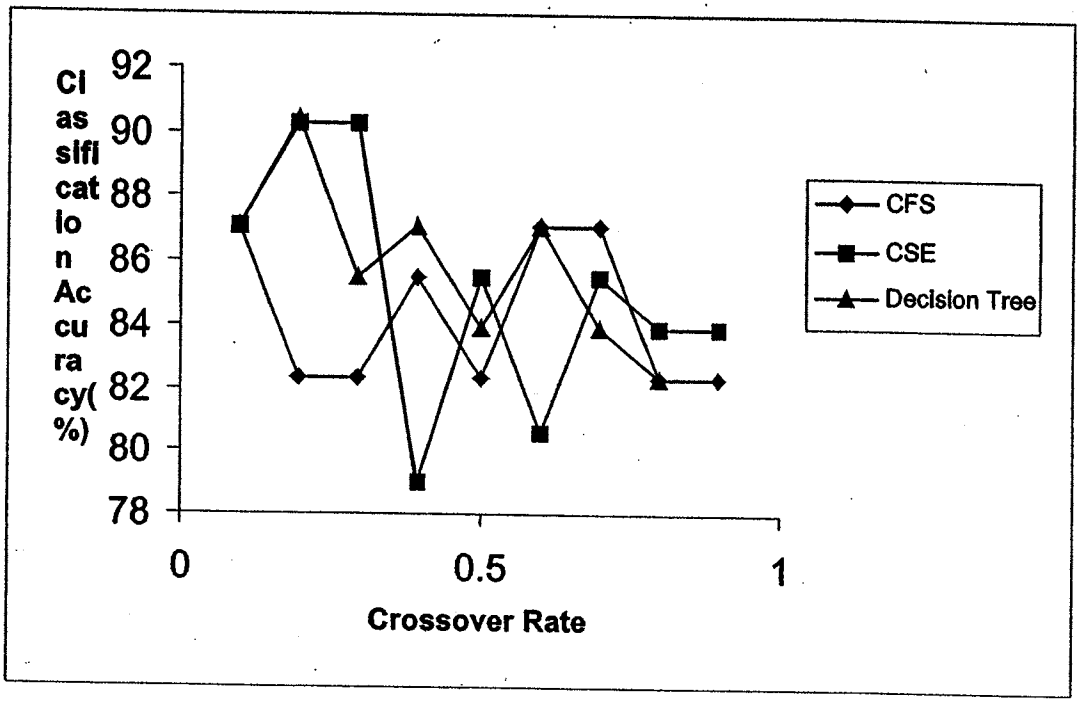


Fig.2. Effect of crossover rate on classification accuracy

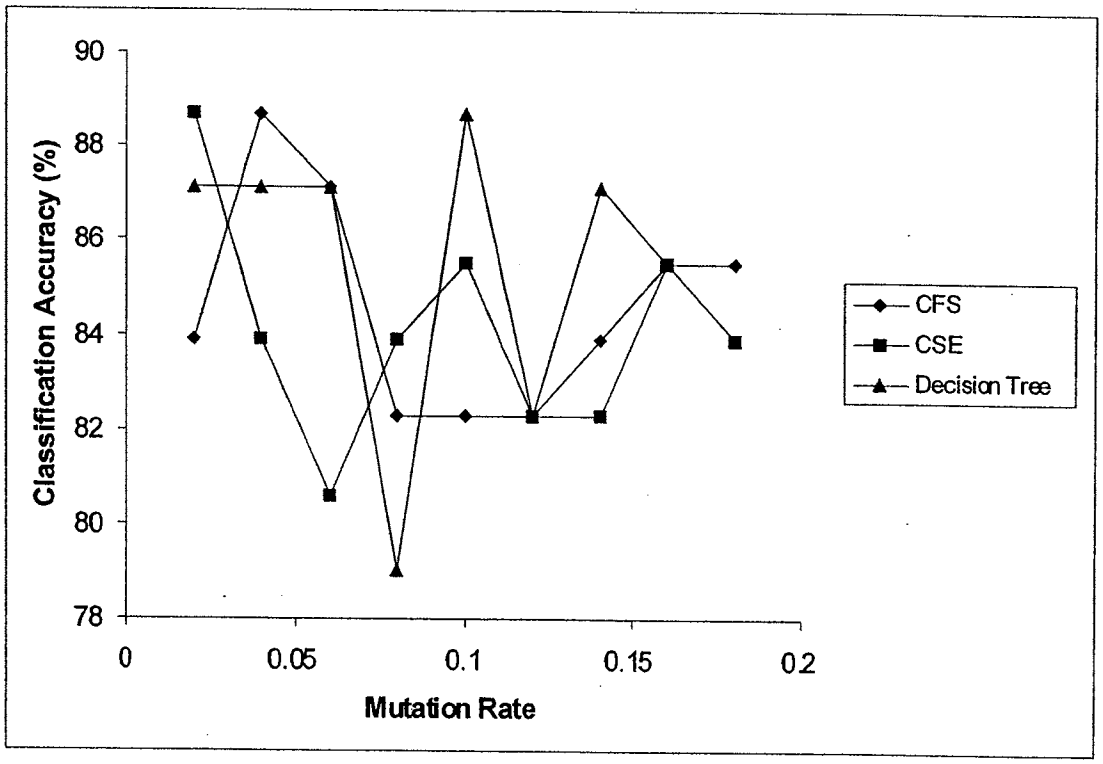


Fig. 3. Effect of mutation rate on classification accuracy

We fix the number of generations at 20. The mutation rate ranges from 0.02 to 0.18 with a crossover probability of 0.6. (Fig.3 and Appendix C). In this experiment, we can clearly observe that genetic algorithm performs best when the mutation rate is less than 0.05 if decision tree is selected as evaluation function. If CFS or CSE is selected as the evaluation function, although we couldn't discern a clear correlation between the classification accuracy and the probability of mutation, the highest accuracy is achieved at a mutation rate of 0.04 and 0.02, respectively.

4. CONCLUSION

Genetic algorithms are heuristic search algorithms that use a population of possible problem solution methods and a performance criterion to evaluate those solution methods, to search for a global optimum solution method. The advantage of GA is it can achieve parallelism and avoid local optimization. Here, the genetic algorithm is applied with several different evaluation functions to perform gene selection. Applying this approach on expression data from 2000 genes in 22 normal and 40 colon cancer samples, we found that 4 or 5 genes are enough to classify colon tissue samples with decision tree or SVM selected as the evaluation function, and we can achieve 89% classification accuracy. With the increase of the number of genes in the set, the classification accuracy does not increase. It is likely that the information obtained in a large number of genes can be captured by a small subset without significant loss of information. Among the selected genes, gene 1671(M26383) that encodes protein IL-8 was found constitutively over expressed by some human tumor lines [9]. Therefore, the results are appealing and may have profound implication for clinical applications to maximize the therapeutic efficacy and minimize toxicity.

REFERENCES

1. Berns, A. (2000) Cancer: gene expression in diagnosis. *Nature*, 403, 491-492.
2. Li, W. *et al.*(2002) Tclass: tumor classification system based on gene expression profile. *Bioinformatics*, 18, 325-326.
3. Zhang, H. *et al.* (2001) Recursive partitioning for tumor classification with gene expression microarray data. *Proc. Natl. Acad. Sci. USA,* 98, 6730-6735.
4. Liu, H., and Setiono, R., (1996). A probabilistic approach to feature selection - A filter solution. In 13th International Conference on Machine Learning (ICML'96), July 1996, pp. 319-327. Bari, Italy.
5. Hall, M. A. (1998). Correlation-based Feature Subset Selection for Machine Learning. Thesis submitted in partial fulfilment of the requirements of the degree of Doctor of Philosophy at the University of Waikato.
6. Haykin, S. (1999) *Neural Networks: A comprehensive Foundation, 2nd ed.* New Jersey: Prentice Hall, Inc.
7. Mitchel, T. M. (1997). Machine Learning, The McGraw-Hill Companies, Inc.
8. Model F, Adorjan P, Olek A and Piepenbrock C. (2001) Feature selection for DNA methylation based cancer classification, *Bioinformatics*, 17, S157-S164
9. Kowalski J, Denhardt DT. (1989) Regulation of the mRNA for monocyte-derived neutrophil-activating peptide in differentiating HL60 promyelocytes. *Mol Cell Biol* 9(5):1946-57

Acknowledgement: Our work on this project was supervised by Professor Willard Miranker and Dr. David Tuck.

Appendix A. Time complexity and classification accuracy with different evaluation functions

Evolution Function		Cfs	Consistency SubsetEval	Decision Tree	SVM
10 Generations	Time	1 sec	1 sec	1 min	54 min
	Features Selected	10 genes (249, 377, 493, 576, 682, 765, 780, 1042, 1153, 1671)	4 genes (201, 576, 682, 1671)	4 genes (201, 682, 1423, 1671)	6 genes (377, 491, 576, 682, 765, 1423)
	Classification Accuracy	85.5%	80.7%	80.5%	87.1%
20 Generations	Time	1 sec	1 sec	2 min	105 min
	Features Selected	12 genes (249, 377, 493, 576, 682, 765, 780, 1042, 1153, 1423, 1473, 1671)	5 genes (201, 249, 513, 576, 682)	4 genes (682, 765, 964, 1671)	5 genes (377, 576, 682, 765, 1423)
	Classification Accuracy	87.1%	80.6%	87.1%	88.7%
50 Generations	Time	1 sec	2 sec	4 min	214 min
	Features Selected	10 genes (249, 377, 576, 682, 765, 780, 1042, 1153, 1423, 1671)	5 genes (201, 249, 513, 576, 682)	3 genes (682, 765, 1671)	5 genes (377, 576, 682, 765, 1423)
	Classification Accuracy	87.1%	80.6%	85.5%	88.7%
100 Generations	Time	1 sec	2 sec	9 min	331 min

	Features Selected	10 genes (249, 377, 576, 682, 765, 780, 1042, 1153, 1423, 1671)	5 genes (201, 249, 513, 576, 682)	3 genes (682, 765, 1671)	5 genes (377, 576, 682, 765, 1423)
	Classification Accuracy	87.1%	80.6%	85.5%	88.7%
200 Generations	Time	1 sec	4 sec	19 min	483 min
	Features Selected	8 genes (377, 576, 682, 765, 1042, 1153, 1423, 1671)	4 genes (201, 576, 682, 1671)	3 genes (682, 765, 1671)	5 genes (377, 576, 682, 765, 1423)
	Classification Accuracy	82.3%	79.0%	85.5%	88.7%
500 Generations	Time	2 sec	7 sec	163 min	652 min
	Features Selected	8 genes (377, 576, 682, 765, 1042, 1153, 1423, 1671)	4 genes (201, 576, 682, 1671)	3 genes (682, 765, 1671)	5 genes (377, 576, 682, 765, 1423)
	Classification Accuracy	82.3%	79.0%	85.5%	88.7%
1000 Generations	Time	3 sec	7 sec	306 min	
	Features Selected	8 genes (377, 576, 682, 765, 1042, 1153, 1423, 1671)	4 genes (201, 576, 682, 1671)	3 genes (682, 765, 1671)	
	Classification Accuracy	82.3%	79.0%	85.5%	
2000 Generations	Time	4 sec	23 sec		

	Features Selected	8 genes (377, 576, 682, 765, 1042, 1153, 1423, 1671)	4 genes (201, 576, 682, 1671)		
	Classification Accuracy	82.3%	79.0%		
5000 Generations	Time	8 sec	49 sec		
	Features Selected	8 genes (377, 576, 682, 765, 1042, 1153, 1423, 1671)	4 genes (201, 576, 682, 1671)		
	Classification Accuracy	82.3%	79.0%		

Appendix B. The effect of crossover rate on selected features and classification accuracy (20 generations, mutation rate = 0.06)

Evolution Function		Cfs	Consistency SubsetEval	Decision Tree
0.1	Features Selected	10 genes (249, 377, 576, 682, 765, 780, 1042, 1153, 1423, 1671)	4 genes (576, 682, 765, 1671)	4 genes (682, 765, 1042, 1671)
	Classification Accuracy	87.1%	87.1%	87.1%
0.2	Features Selected	8 genes (377, 576, 682, 765, 1042, 1153, 1423, 1671)	5 genes (513, 576, 682, 765, 1423)	5 genes (249, 251, 377, 576, 765)
	Classification Accuracy	82.3%	90.3%	90.4%

0.3	Features Selected	8 genes (377, 576, 682, 765, 1042, 1153, 1423, 1671)	5 genes (513, 576, 682, 765, 1423)	7 genes (201, 491, 493, 682, 980, 1153, 1671)
	Classification Accuracy	82.3%	90.3%	85.5%
0.4	Features Selected	10 genes (249, 377, 513, 576, 682, 765, 1042, 1153, 1423, 1671)	4 genes (201, 576, 682, 1671)	6 genes (251, 491, 576, 682, 765, 1153)
	Classification Accuracy	85.5%	79.0%	87.1%
0.5	Features Selected	8 genes (377, 576, 682, 765, 1042, 1153, 1423, 1671)	5 genes (201, 249, 576, 682, 1671)	8 genes (251, 491, 493, 682, 780, 980, 1157, 1671)
	Classification Accuracy	82.3%	85.5%	83.9%
0.6	Features Selected	10 genes (249, 377, 576, 682, 765, 780, 1042, 1153, 1423, 1671)	5 genes (201, 249, 513, 576, 682)	4 genes (682, 765, 964, 1671)
	Classification Accuracy	87.1%	80.6%	87.1%
0.7	Features Selected	11 genes (249, 377, 576, 682, 765, 780, 1042, 1153, 1423, 1473, 1671)	5 genes (576, 682, 765, 780, 1423)	6 genes (201, 491, 493, 682, 980, 1671)
	Classification Accuracy	87.1%	85.5%	83.9%
0.8	Features Selected	8 genes (377, 576, 682, 765, 1042, 1153, 1423, 1671)	5 genes (201, 576, 682, 765, 780)	7 genes (201, 491, 493, 513, 682, 980, 1671)
	Classification Accuracy	82.3%	83.9%	82.3%

0.9	Features Selected	8 genes (377, 576, 682, 765, 1042, 1153, 1423, 1671)	5 genes (249, 493, 576, 682, 1153)	7 genes (201, 491, 493, 682, 780, 980, 1671)
	Classification Accuracy	82.3%	83.9%	85.5%

Appendix C. The effect of mutation rate on selected features and classification accuracy (20 generations, crossover rate = 0.6)

Evolution Function		Cfs	Consistency SubsetEval	Decision Tree
0.02	Features Selected	10 genes (249, 377, 576, 682, 765, 964, 1042, 1153, 1423, 1671)	5 genes (201, 377, 576, 682, 765)	5 genes (249, 377, 576, 76 780)
	Classification Accuracy	83.9%	88.7%	87.1%
0.04	Features Selected	11 genes (249, 377, 493, 576, 682, 765, 1042, 1153, 1423, 1473, 1671)	6 genes (201, 251, 493, 576, 682, 765)	6 genes (249, 377, 513, 57 765, 783)
	Classification Accuracy	88.7%	83.9%	87.1%
0.06	Features Selected	12 genes (249, 377, 493, 576, 682, 765, 780, 1042, 1153, 1423, 1473, 1671)	5 genes (201, 249, 513, 576, 682)	4 genes (682, 765, 964, 1671)
	Classification Accuracy	87.1%	80.6%	87.1%

0.08	Features Selected	14 genes (201, 249, 377, 493, 513, 576, 682, 765, 780, 1042, 1153, 1423, 1473, 1671)	5 genes (249, 513, 576, 682, 765)	6 genes (201, 251,493, 682 980, 1671)
	Classification Accuracy	82.3%	83.9%	79.0%
0.10	Features Selected	10 genes (249, 377, 513, 576, 682, 765, 1042, 1153, 1423, 1671)	5 genes (201, 513, 576, 682, 765)	6 genes (249, 377, 491, 57(765, 783)
	Classification Accuracy	82.3%	85.5%	88.7%
0.12	Features Selected	9 genes (249, 377, 576, 682, 765, 1042, 1153, 1423, 1671)	6 genes (201, 249, 576, 682, 780, 980)	9 genes (201, 251, 377, 491 493, 682, 964, 980 1671)
	Classification Accuracy	83.9%	82.3%	82.3%
0.14	Features Selected	12 genes (201, 249, 377, 493, 576, 682, 765, 780, 1042, 1153, 1423, 1671)	5 genes (201, 576, 682, 765, 980)	5 genes (249, 682, 1042, 1423, 1671)
	Classification Accuracy	83.9%	82.3%	87.1%
0.16	Features Selected	10 genes (249, 377, 493, 576, 682, 765, 1042, 1153, 1423, 1671)	5 genes (201, 576, 682, 765, 964)	5 genes (251, 513, 682, 76 1671)
	Classification Accuracy	85.5%	85.5%	85.5%

0.18	Features Selected	12 genes (201, 249, 377, 576, 682, 765, 780, 1042, 1153, 1423, 1473, 1671)	7 genes (201, 251, 576, 682, 765, 783, 1153)	7 genes (251, 491, 513, 68 765, 980, 1671)
	Classification Accuracy	85.5%	83.9%	83.9%

Appendix D.

The detail steps in our approach.

1. Define a genetic representation of the problem
2. Create an initial population $P=\{g_1, \ldots, g_N\}$
3. evaluate the fitness, $F(X_i)$ for each of the individuals in the population with an evaluation function such as a decision tree
4. Compute the average fitness for the population, F_{avg}
5. Assign each individual the normalized fitness $F(g_i)/ F_{avg}$.
6. Assign each individual g_i a probability p_i proportional to its normalized fitness. Using this distribution, select N vectors from P to construct a subset S.
7. Pair all of the vectors in S at random forming N/2 pairs as parents.
8. Apply crossover with probability p_{cross} to each pair in S
9. Apply mutation with probability $p_{mutation}$ to some pairs in S
10. Check termination conditions. Terminate if solution achieved.
11. Otherwise, goto Step 3.

Using Neural Networks to Implement Selective Search

Tomislav Nad
Department of Computer Science
Yale University
New Haven, CT 06520

Abstract

Since the beginning of AI research, most game-playing computer programs have been using full-width search to probe all possible moves up to some maximum depth. In doing so, they also search a huge number of unpromising moves. Humans use a more sophisticated search strategy. They use selective-search to probe only a small number of promising moves, while pruning all uninteresting moves. Although humans cannot come even close to the number of moves searched by the computer, this selective-search strategy allows us to search to greater depths than most computers. We attempt to use an artificial neural network to implement selective-search for the game of 3-dimensional Tic-Tac-Toe. Our results make a strong case that in the future of computer game playing, selective-search strategies will surpass performance of full-width search.

Keywords - neural networks, temporal difference learning, TD(lamba), selective search, tic-tac-toe, beam search

1. INTRODUCTION

In his original paper on computer chess, Claude Shannon described two possible strategies for searching for the best move – full-width search and selective search (quoted in Tesauro, 1995). Most computer programs use a full-width approach to game playing, in which all possibilities up to a fixed depth are searched – this is also called Shannon-A strategy. In selective search – also called Shannon-B strategy – most of the possibilities are pruned and only promising ones are considered. This allows searching to much greater depths. Shannon thought that this approach is more efficacious because it mimics human behavior. However, it is extremely hard to decide which nodes in a search tree to prune and which to search - an apparently bad move might actually be a brilliant sacrifice that leads to a win later in the game.

Key to selective-search is designing a function that specifies what to search and what to prune (we will call this the *selector* function). We address this problem by using a neural network that will learn the *selector* function.

Notice that since we only want to search moves that will lead to a win for us, deciding which moves to search is same as projecting which moves will lead to a win. Therefore, an *evaluation* function, which evaluates how good some move is, can also be used as the *selector* function. Much work has been done on training neural networks as evaluation functions (Scott, 2002). We will use Temporal Difference learning to train our neural network.

We use three-dimensional 4x4x4 Tic-Tac-Toe (TTT) to test our method of searching. TTT is a good candidate for this, because it is straightforward to program an opponent that uses a classic full-width search, and because 4x4x4 TTT is complicated enough (average branching factor of 32) to engender interesting results.

2. NETWORK

2.1 Temporal Difference Learning

When using reinforcement learning, the learning agent observes an input state, produces an output signal, and then receives a reinforcement feedback signal from the environment. The goal of learning is to generate actions that will lead to a maximal reward. However, when playing games, the reward is delayed and given at the end of a long sequence of actions. Thus the agent that uses reinforcement learning to learn how to play games must solve a "temporal credit assignment" problem (Tesauro, 1995). In other words, it must determine how to partition the reward among each of the moves made by a player. Methods for solving the temporal credit assignment problem are known as "Temporal Difference" learning methods, and the basic idea behind these methods is that learning is based on the difference between temporally successive predictions (chance of winning). The goal of Temporal Difference learning is to make the learner's prediction at the current time step match his prediction at the next time step. In the neural network domain, prediction at each time step is represented by the output of the network at that

time step. Therefore, our goal is to train the neural network so that its current output would match its output at the next time step.

TD(*lambda*) is a temporal difference algorithm for training a multilayer neural network (MNN). The formula for weight change in a MNN using TD(*lambda*) is:

$$\Delta W_t = W_{t+1} - W_t = \eta(Y_{t+1} - Y_t)\sum_{k=1}^{t} \lambda^{t-k} \nabla_w Y_k$$

(Tesauro, 1995) where η is a small constant (commonly known as the "learning rate"), W is a vector of synaptic weights, Y_t is the output of the network at time step t, and $\nabla_w Y_k$ is gradient of the network output with respect to the weights. Output of the network Y_t is a measure of probability that the current board position will lead to a win. The quantity λ is a discount factor controlling how an error detected at the given time step feeds back to the previous estimates. When $\lambda = 0$, no feedback occurs, and when $\lambda = 1$, errors feed back without discounting, arbitrarily far in time (but not beyond the start of the current game) (Tesauro, 1995).

At end of each game, the same equation is used except that (Y_{t+1} - Y_t) is replaced with (O-Y_t), where O is expected output of the network. In case of win, O=0.9, in case of loss, O=0.1, and in case of draw, O=0.5.

As can be seen from the equation, TD(lambda) calculates the measure of error at each time step (namely, Y_{t+1} - Y_t), and this quantity drives the learning (the changing of the synaptic weights).

It is known that changing the learning rate during training can speed up the training, so we modified the TD(lambda) equation by including a "momentum term" (Haykin, 1999, p. 170):

$$\Delta W_t = \alpha \Delta W_{t-1} + \eta(Y_{t+1} - Y_t)\sum_{k=1}^{t} \lambda^{t-k} \nabla_w Y_k$$

where α is the momentum constant, and ΔW_{t-1} is weight change at the previous time step. When ΔW has the same algebraic sign on consecutive iterations, momentum causes the acceleration of descent (Tesauro, 1995, p. 170), and when ΔW has opposite algebraic signs on consecutive iterations, momentum causes a "stabilizing effect" (Tesauro, 1995, p. 171).

2.2 Network Design

We experimented with several network designs, using Java to implement our networks. Our first network design had 192 input nodes. 64 inputs for representing friendly pieces on the board, 64 inputs for opponent pieces, and 64 inputs for empty spaces on the board. The choice to represent the TTT board with 192 binary inputs rather than with 64 level inputs was based on the suggestion in (Skapura, 1996, p. 73)*. We used three different networks, with 50, 150, or 300 neurons in a hidden layer. There was one output neuron. The output of the network indicated an assessment of the current position for the player. Unfortunately, even after 1,000,000 training epochs, our networks failed to reach a satisfactory level of play. We believe this was due to the high complexity of 4x4x4 Tic-Tac-Toe, and that this could be solved with an addition of a second hidden layer or with larger hidden layers. However, this new network would require even more time to converge, so we decided to use a hybrid approach.

Our second network design had 200 inputs, 40 neurons in the hidden layer, and one output neuron. In this network, the 192 input neurons were the same as in the first network, but we added 8 hard-coded features. The first 4 features represented the number of lines in the TTT board filled with 1, 2, 3, and 4 friendly pieces, and the other 4 features represented number of lines filled with 1, 2, 3, and 4 enemy pieces. Those 8 features come from the heuristic that a human player focuses on these lines as he decides his move. We experimented with different sizes for the hidden layer, and decided on 40 hidden neurons. There was one output neuron, as in the first design.

2.3 Measuring performance

We used the win/loss ratio to measure the performance of each search method and network. We feel that the number of wins is by far the most important measure of performance, so we avoided more complicated measures such as the number of moves to reach the end of the game.

---

* Skapura claims that a significant benefit of this representation is that it insists that different input vectors are orthogonal to each other. He further claims that this is a benefit for neural networks that must discriminate between similar patterns because orthogonal vectors are easily detected by the network.

2.4 Training

We trained the neural network using TD(*lambda*) learning with a momentum term, and with the following parameters: α=0.5, η=0.015, λ=0.7. Parameters α and η were chosen arbitrarily, and λ was set high enough to facilitate sufficient feed-back of error.

We programmed an AI opponent that used classical depth-first search and a hard-coded *evaluation* function. During training, the network played against this AI opponent. The strength of this opponent was gradually increased during the training.

At the beginning of each training game, up to 5 moves were randomly placed on the board in order to avoid the same starting position all the time. The latter circumstance would cause both the neural network and its hard-coded AI opponent to always select the same sequence of moves. This randomization led to 8,303,632 possible starting positions. To prevent the NN from getting trapped in the local minima, we added a 10% chance that the NN will select a random move at each time step. The purpose of these random moves was to give the network a chance to explore a sequence of moves that it would otherwise ignore, and in particular, to give it a chance to jump out of a local minimum's basin of attraction.

Training lasted for 900,000 epochs (games), upon which the network's weights converged. At the end of the training, we measured the network's performance by letting it play 1,000 games against the hard-coded AI opponent. The strength of the hard-coded AI opponent was set to the maximum. The network's win/loss ratio against this opponent was 1.17 demonstrating that our network learned to play better than the hard-coded AI we created.

3. EXPERIMENT

3.1 Hypothesis

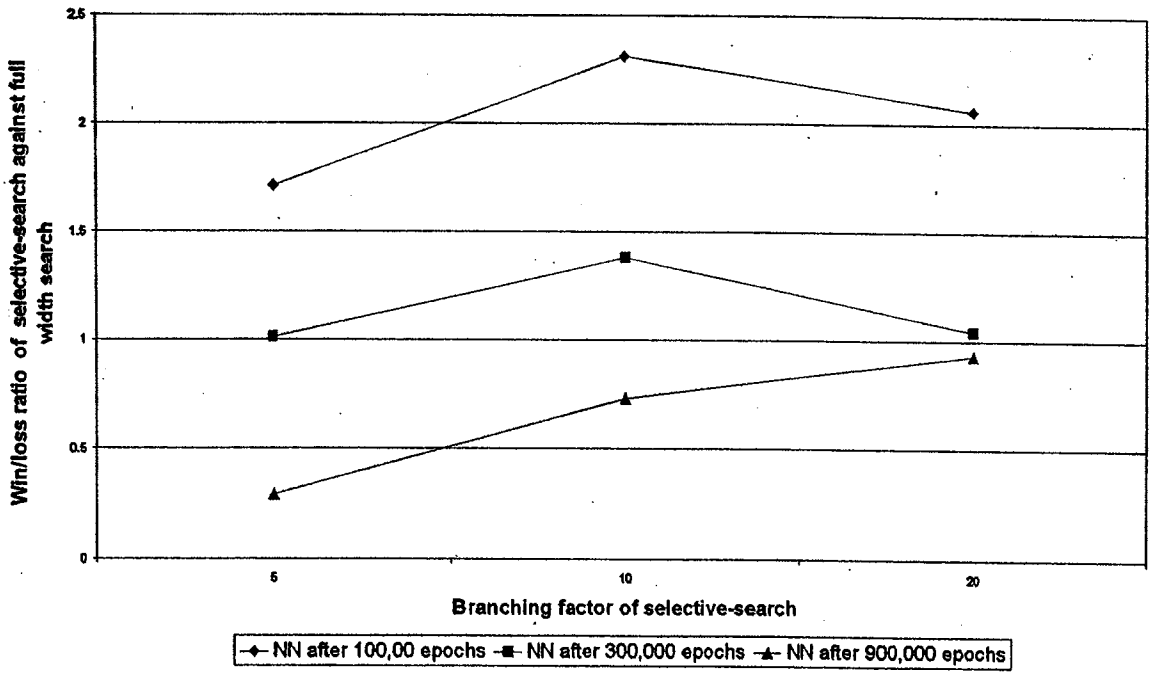
Our hypothesis is that given a good *evaluation (selector)* function, it is possible to reach a better level of play using selective-search than using a full-width search. A good *evaluation* function is crucial, because otherwise we will prune moves that lead to a win.

3.2 First experiment

In the first experiment we implemented selective-search by setting an artificial maximum branching factor (number of states each state can be expanded to) at each node of the search tree. If the number of a node's children was greater than artificially set maximum branching factor, all children were evaluated using the *evaluation* function, and children with the lowest value were pruned. This lowered the branching factor of the entire search tree, and allowed us to search to a greater depth.

We ran measurements with the maximum branching factor set to 5, 10, and 20. We wanted to see how the performance of *evaluation* function effects overall results, so we ran measurements on our neural network in three different stages of training – after 100,000 epochs, after 300,000 epochs, and after 900,000 epochs. At each test run, two identical neural networks played against each other; however, one NN used selective-search and other used full-width search. Each player had a maximum of 10 seconds to choose a move. The length of time constraint was chosen arbitrarily. During this time both players searched until they run out of time, and because of lower branching factor of selective-search, it was able to search deeper than full-width search.

Figure 1: Performance of selective search against full-width search



As can be seen from Figure 1, a branching factor of 10 was the best choice. Surprisingly, as the network progressed in training, the utility of using a selective-search instead of a full-width search shrunk. We believe this is the case because the *evaluation*

function the network represented before it was fully trained had such a low fitness that even a full-width search failed to find the winning move.

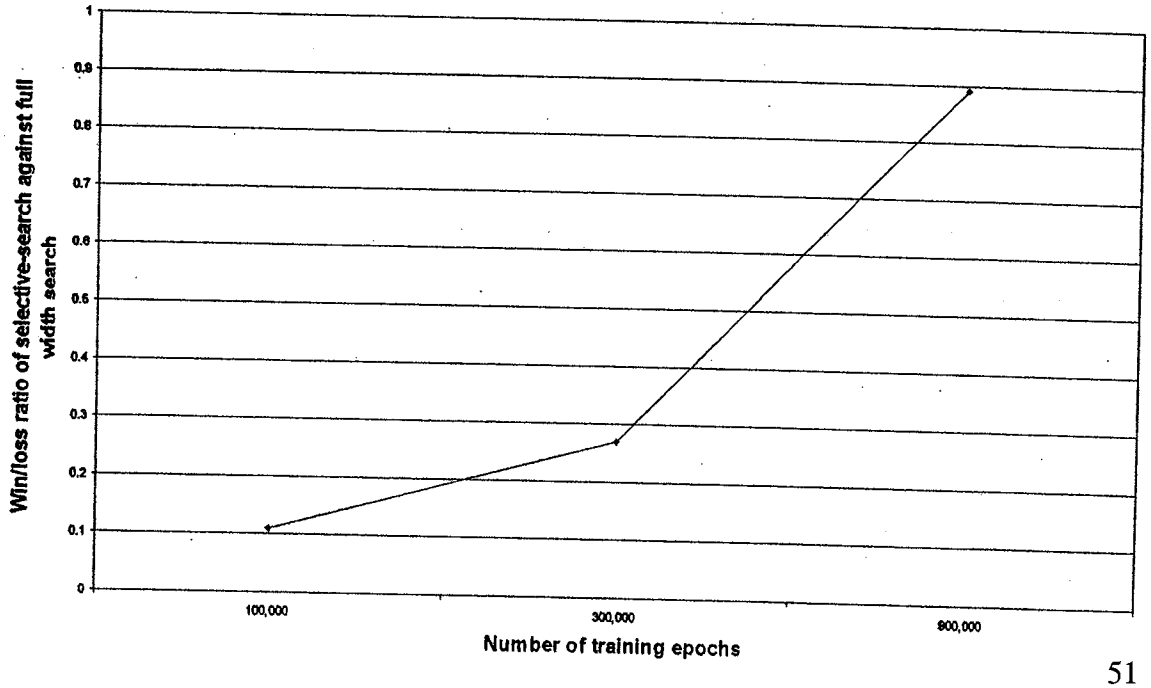
Performance of the fully trained network (approximately 900,000 training epochs) using selective-search was lower than when using full-width search. We believe that this is due to fact that even this network does not represent an evaluation function that is good enough to be used for pruning in the selective-search.

3.2 Second experiment

In our second experiment we used beam-search as our selective-search method. Beam-search examines the search-tree by searching only the n best nodes at each level of the search tree. Since beam-search only searches a fixed number of nodes at each level, the time-complexity of beam-search is polynomial. This allows beam-search to probe much deeper than full-width searches, which have exponential time-complexity.

Again, we measured the performance of our neural network during different stages of training. Two identical neural networks played against each other, one using beam-search and other using full-width search. The constant n for beam-search (width of the beam) was set to 700. As in the first experiment, each player was given 10 seconds to select a move.

Figure 2: Performance of beam-search against full-width search



When using beam-search, performance was lower then when using full-width search, but we can see that the win/loss ratio increases as the evaluation function gets better. This strongly suggests that, given a better *evaluation* function, beam-search would surpass the performance of full-width search.

4. CONCLUSION

Although our approach of using a neural network to implement selective-search engendered some interesting results, our fully trained network could not surpass performance of a classical full-width search. We believe this was mainly due to unsatisfactory performance of our neural network which acted as an *evaluation* function. Increasing performance of this network would require increasing the size of the hidden layer, and perhaps even adding a second hidden layer. However, training such a network would surpass the abilities of most modern computers, so we could not test this hypothesis.

We were able to significantly increase performance of our network by using a hybrid approach, and other researchers reported similar results (Tesauro, 1995; Scott, 2002). One possible expansion of our work is to try to increase the performance of our network by adding more hard-coded features.

Unexpected results in the first experiment suggest that a selective-search may be beneficial if we have a relatively poor evaluation function. When using full-width search, it is known that searching deeper with a poor evaluation function generally achieves better performance than using a better evaluation function and searching to lesser depths (Scott, 2002). Our experiment suggests that this is true even if we prune most of the search-tree during our deeper search, and this result could be useful in problems in which it is hard to create a good evaluation function.

The second experiment showed that performance of beam-search approaches the performance of full-width search as the *evaluation* function improves. We expect that, at some point the performance of beam-search would surpass performance of full-width

search. However, as noted above, due to time constraints we were unable to construct and train the neural network that could reach such a level of performance.

Although research in artificial neural networks has come a long way, our inability to implement larger neural networks remains a significant problem. Although there are possibilities such as using neural networks that are not fully connected, and using batch learning try to overcome this problem, ultimately, a final solution must come in the form of more computing power or true parallel computing.

5. REFERENCES

Tesauro, G. (1995). Temporal Difference Learning and TD-Gammon. *Communications of the ACM.* Vol. 38, no. 3.

Haykin, S. (1999). *Neural Networks: A Comprehensive Foundation.* New Jersey: Prentice Hall.

Skapura, D. (1996). *Building Neural Networks.* New York: ACM Press.

Scott, Jay. (2002). *Machine Learning in Games.* <http://satirist.org/learn-game/>.

License Plate Detection and Comprehension Using Image Processing and Neural Networks

Leonid Shklovskii
Department of Computer Science, Yale University
New Haven, CT 06520

Abstract
We describe an algorithm and its implementation to locate and read a license plate in a photo of a vehicle. Several image processing techniques combined with an innovative fast Hough transform allow the program to simplify the original complex and busy images into a set of potential candidate plate locations. Once this is done a relatively simple character recognition neural network then allows the program to identify the characters as well as resolve the actual location of the license plate in the photo.

Keywords: Hough transform, edge detection, cars, license plate, neural network, annealing.

1. INTRODUCTION

The problem of machine vision has existed for well over fifty years. At the very beginnings it seemed like a relatively easy task and one that was initially assigned to an undergraduate student as a summer project. As that poor student quickly realized, however, there is a tremendous variety of different factors that affect how the image is perceived. Effects such as lighting conditions, occluded parts of the image, inconsistencies in materials, and shadows can all create tremendous problems for any kind of vision analysis system. Because of this complexity, the general vision problem must be broken down into smaller, task-oriented pieces. The specific problem addressed here is locating and reading a license plate on a photo of a vehicle.

Because the zoom on the exemplar photos varies from shot to shot, and the individual license plates come in several different styles, it is not clear how to define the license plate in terms that the computer can use and search for. Previous approaches have side-stepped this difficulty by focusing on detecting the letters in the image directly[3]. The letters used on license plates are all of one font and are all one solid color. They also have the advantage of being largely unaffected by the lighting of the photo. Looking directly for the letters trades off the benefit of knowing the location of the license plate (and thus

restricting the search location for reading the actual letters) for a simpler and more easily defined search for the letters. By ignoring the license plate, these approaches require extensive supporting steps to deal with numbers and letters appearing in other sections of the image.

Because all license plates (of a particular state) have a roughly similar design, once the license plate is located, it is relatively easy to identify the letters that make up the license plate and read them. The raw amount of information in the photo from which the license plate must be detected and read is tremendous. A pixel consists of three 8-bit color values and there are more than 300,000 pixels in the image. Running a neural network on this kind of scale would incur an incredible time cost in training, and a prohibitive memory cost[2]. This program attempts to reduce that number to a much more manageable set of inputs. In order to reduce this huge amount of data a variety of choices, such as mask selection, were made on the basis of general principles of image processing and experimentation.

2. DATA

The data used for this project were photos of the backs of cars in the York Square parking lot in downtown New Haven, CT. They were taken early in the afternoon with a Canon PowerShot S200 digital camera. A resolution of 640x480 at the SuperFine level of JPEG compression provided adequate information without making the images too large. The images were then converted from TrueColor (24 bit) images to 256 color indexed PNG (portable network graphics) files for ease of use.

A majority of the photos were the standard Connecticut plates that were instituted in 2000 (Table 1).

109	Shots of the rear of vehicles	96	Standard CT plates (### · ZZZ)
1	Shot of the front of a vehicle	11	Combination CT plates (??????)
		2	Fancy CT plates (A ## · ZZZ)
		1	Standard Minnesota plate
(the A on the fancy CT plates is a lighthouse)			

Table 1. Breakdown of the photos taken

The license plates on the photos range in size from 100 pixels to 400 pixels wide. Lighting conditions are roughly similar, but several different light balances were used on

the camera. The locations of the license plates in the photos also varied from being exactly in the middle to being within 50 pixels of the edges of the photo (Figure 1).

Photos #51 – 100 were used to train the neural network and to fine tune the image processing and the Hough transform modules. Photos #1 – 50 were used as new exemplars for the whole program.

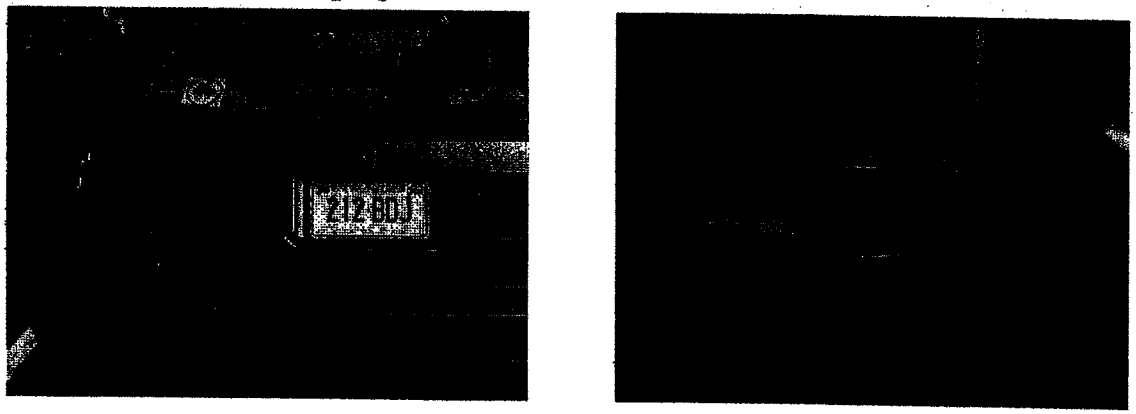


Figure 1. Example photos

3. IMPLEMENTATION

In order to solve this problem, a program was written in ANSI C and executed on Dual Xeon 1.4 GhZ Dell computers running Red Hat Linux. With the exception of the graphics library GD (versions 2.0.8 and 2.0.9) no external code was used. The GD library provided the image handling tools, such as loading, resizing, cropping and saving.

The outline of the program is as follows:

3.0 Preparatory

Before the program can be run, it needs to be given a definition of a license plate. This is done through an image that is a composite of the license plates from images #51-100. The license plate is cropped out by hand, rescaled to the same size and each active pixel is assigned an alpha of $1/50^{th}$. Then, all of these images are merged and thresholded to create a generic license plate. This license plate is at a resolution of 100x50 pixels and will be later scaled in the Hough transform to fill the whole 100-400 pixel width range in 15 pixel increments (leading to 20 different possible sizes)

3.1 Initialization

The program loads the image to be processed as well as the generic license plate sample(s). More than one sample at a time can be used however, as will be explained, multiple samples can dramatically slow down the performance of the application. This is due to the computational complexity of the Hough transform and will be discussed in

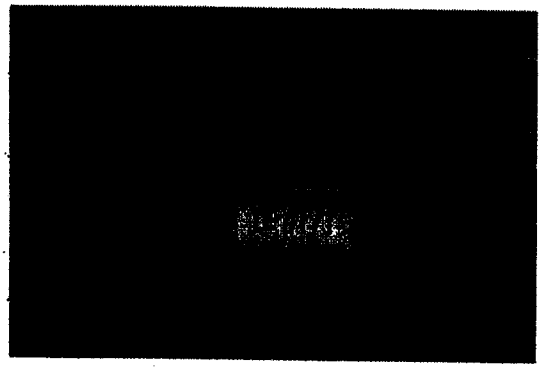


Figure 2. Photo #90

section 3.3. The initialization also creates the accumulator matrix for the Hough transform. Because of the scale of the problem approximately 32MB of memory is required to hold the relevant variables in memory. The whole process will be demonstrated on photo #90 (Figure 2).

3.2 Image Pre-processing / edge detection

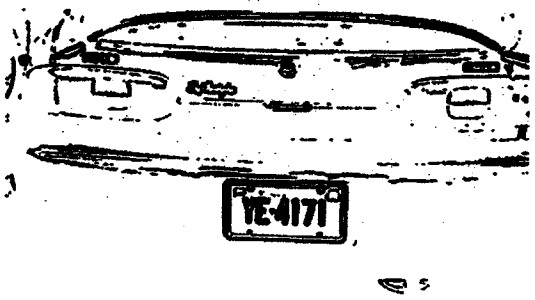


Figure 3. Edge detection of photo #90 using the expanded Sobel matrix

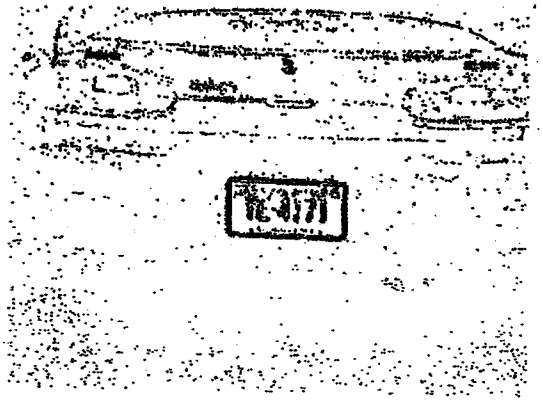


Figure 4. Edge detection of photo #90 using Robert's Cross matrix

The image is converted to grayscale and a Sobel-like convolution matrix (Table 2) is applied to it. The matrix is applied in both the vertical and horizontal alignments and the two resulting images are combined and their brightness is normalized to preserve the brightness of the original photograph. There are a variety of different methods for detecting edges in an image and nearly all of them exploit the same idea. A pixel that is on an edge is close to a pixel of a different intensity (assuming a grayscale image). Doing pixel operations such as with these matrices makes these differences more prominent. The expanded version of the Sobel matrix that was used in the program accentuates vertical edges in its normal state, and horizontal edges in its transposed state. Since these are the edges that define a license plate, it's an efficacious choice (Figure 3). Other matrices that are often used for edge detection tend to be more distracted by noise in the images. Because of the conversion of the photos to indexed color and the lighting differences, many areas that should be uniform showed up as false edges when other matrices, such as Robert's Cross were used (Figure 4).

-1	-1	0	1	1
-1	-1	0	1	1
-2	-2	0	2	2
-1	-1	0	1	1
-1	-1	0	1	1

Table 2. Vertical version of expanded Sobel matrix

3.3 Hough Tansform

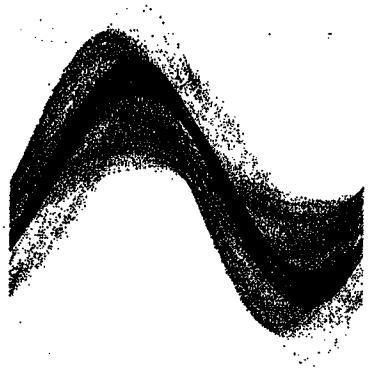


Figure 5. A Linear Hough transform of figure 2

Once the edge detection is complete, a Fast Generalized Hough transform is performed on the image. This function maps out the possible locations in feature space from 20 (one for each of the different possible sizes of the license plate) maps in feature space of license plates based on a template. The general Hough transform takes a line or another pattern and converts an image into a feature space. In this feature space the value of each point is the probability that the license plate is actually in that spot[4]. The accumulator array that was created at initialization time stores this information. Hough transforms have been used in previous research to reduce the complexity of an image[5]. The initial plan was to use the Hough transform to identify the vertical and horizontal lines in the photo and then use those lines to find the license plate. The number of pixels that were classified as edges by the detection algorithm created an extremely busy feature space graph. The sheer number of edges detected made it impractical to use a neural network directly on the Hough transform (Figure 5). Fortunately, the same principle is just as applicable to circles, and any other shape that can be defined in feature space. The Hough transform is extremely good at detecting shapes even with large amounts of noise or missing points. A major limitation of the general Hough transform is that it can be extremely slow. One general Hough transform requires the whole template image to be scanned for each pixel in the image that's being transformed. Because the license plate sample image can not be defined as a function parametrically, differently scaled samples that could match license plates of different dimensions must be used. This grows even more computationally expensive as the sample images get larger and have more pixels that must be compared. Another difficulty that exists is that each size must have its darkness normalized. A license plate sample that is twice as large as another can contain up to four times the number of pixels, which means that without normalization it will have more pixels voting for it. When the size of the license plate to be detected in the photo was specified and the appropriately scaled sample was specified manually, the success rate of correctly identifying the plate location was 78%. Figure 6 shows an example, where the darkest pixel in the transform (roughly in the middle of the fuzzy rectangle) is the most likely location for the upper left corner of the license plate.

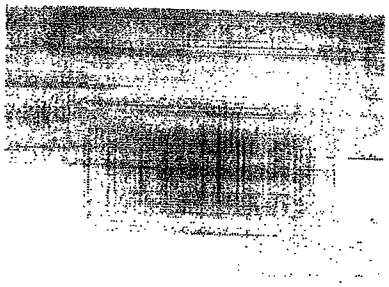


Figure 6. A Hough transform of the photo in Figure 2, based on the license plate sample graphic

In order to identify the arbitrarily sized license plate in a reasonable amount of computation time, the program uses only one sample image and scales the image's individual pixels inserting empty space in between the now spread out pixels. Figure 7 demonstrates this on a set of pixels. This drastically reduces computational cost of doing the Hough transform and allows only one sample image to be used in order to find a license plate of any size. The program automatically performs the scaling and the larger versions of the license plate sample are essentially as effective as the original. This 'Fast General Hough Transform' (FGHT) is crucial in keeping the running time of the program reasonable (within a minute).

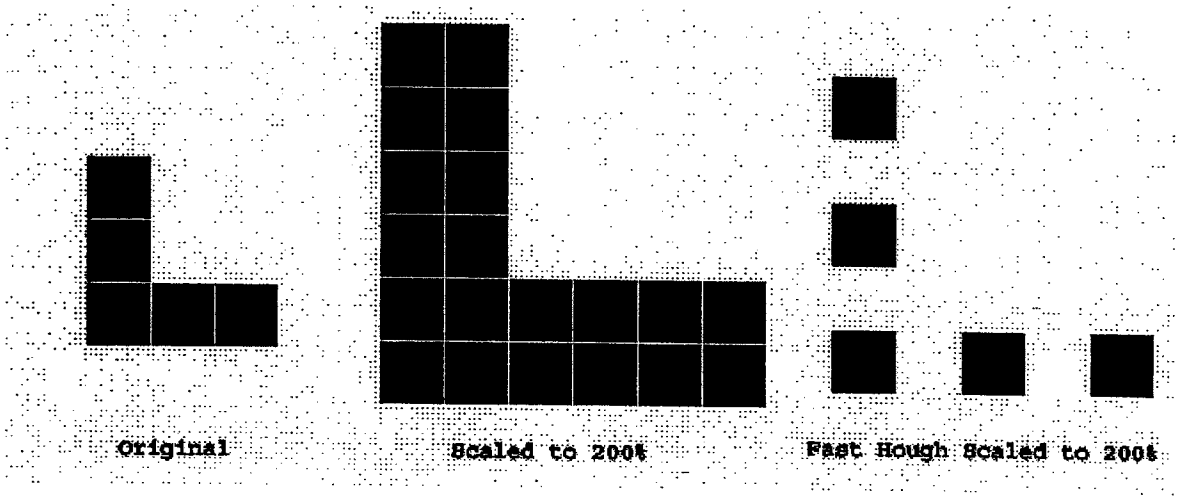


Figure 7. Fast Hough Transform example

Running the Hough transform with each of these scaled samples yields 20 different maps, each with a best location for the license plate if a license plate of that size existed in the image. Unfortunately a side effect of the scaling is that the accuracy of the Hough transform is reduced so that it is not possible to algorithmically determine which size (and corresponding candidate location) is the correct one. If the correct size can be identified, then the certainty is very high that the Hough transform of that size has correctly located the actual license plate. Discovering which of the license plate candidates is the real one is done by sending all 20 of them to the neural network in an attempt to figure out which one has more recognizable letters. Ideally, the size that's the best match will generate the best post-processed image and thus have more recognizable letters.

3.4 Post-processing

Each of the rectangles that were identified through the FGHT is rescaled and cropped to a standard size of 240x66 pixels (Figure 8). A dynamic threshold is then performed on the license plate candidates to make the letters on the license plates

Figure 8. Automatically cropped license plate candidate from figure 2

stand out. The thresholding is done on the assumption that the darker letters cover less than half of the area of the license plate and as such will be darker than the average pixel in the license plate. In practice setting the threshold value to 1.2 times the average brightness gave excellent results for most of the training license plates. Individual letters are then separated out, cropped to a size of 34x66 pixels and sent through the character recognition system. There are several methods to find the letters in the cropped license plate, each with its associated tradeoffs. Because of time constraints an extremely simple algorithm that converted the candidate to black and white, and then performed a scan by columns to find the first column that had pixels in it. This would identify the leftmost letter and the rest of the letters can be extrapolated since the license plate font is monospaced and un-kerned. Unfortunately many license plates had noise on the edges that caused problems for this algorithm. Other successful approaches have included creating a histogram of the license plate candidate but require much more time to implement and tune[1]. Even if no characters are detected in the image, it is still cut up into character size pieces and passed on to the recognizer, but each of these will generate very low confidence scores from the neural network. That in turn will result in a low certainty score for that license plate candidate. This can be seen in the results table.

3.5 Letter recognition using a neural network

The letter recognition system is a 3 layer feed forward network. The input layer consists of 2244 elements, each of which is connected to a pixel of the character block are the individual pixels of the cropped characters. They connect to the hidden layer of half that size which leads to the output layer of 37 neurons. Those 37 neurons correspond to the letters of the alphabet, the digits 0-9 and the symbol • which is used in between the halves of the license number. Because of the rather simple character detection routine in this version, the • is not ever sent to the recognition system. The neural net gain function used is linear and the weights are randomly initialized in the range: [-1, 1]. A linear gain function is used because it is important to know how well the letter was actually matched relative to the other letters. If the match is poor, we may expect that the 34x66 block of pixels probably comes from a rectangle that is not a license plate and does not actually contain a valid character. The output neuron with the highest value (along with its value) is reported back to the program.

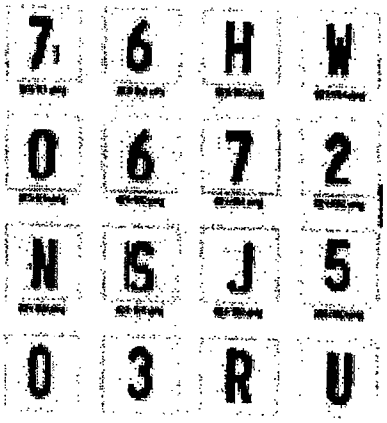


Figure 9. Examples of letters that were used to train the neural net

The neural net was trained by a secondary program that took in as the input the letters that were found by the main program in photos 51 through 100. These were picked out from the rather large amount of non-letter containing blocks that the program identified

and saved to disk as an intermediate step before running the neural network. Because the 50 license plates used were insufficient to provide all of the letters in the alphabet and the digits 0-9, several additional sample letters were created in Photoshop. A total of 226 letters were available for the training and figure 9 shows examples of some of these. The network was then trained by running 5,000,000 iterations using the following relaxation technique: For each iteration a test letter was selected and run through the neural net. 500 weights were selected at random and perturbed by a random amount between -½ and ½. If these perturbations improved the value of the correct output neuron, or lowered the average value of the others, they were kept. If not, the perturbation was reversed.

The first approach that was used in this neural network was to train it with a back-propagation algorithm. Unfortunately, the back propagation algorithm had problems dealing with character inputs that were not perfectly framed. Within the 34x66 block of pixels, the characters could be several pixels off of the training stimuli resulting in a confidence scores that were indistinguishable from random noise, which in turn made choosing the correct license plate size impossible. The relaxation technique that was used could identify letters that were horizontally shifted much more effectively.

3.6 Output

Once letter recognition has been attempted on each of the potential rectangles, the rectangle which generated the best confidence results is identified as are the associated letters that were recognized by the neural network.

4. RESULTS

The program was run on images 1-50 with the following results.

Photo	Actual plate	Characters identified correctly	Plate found?
1	774·PCP	774	Yes
2	193·AGL	None	No
3	846·PZW	8 6	Yes
4	26C·691	I	No
5	890·KOL	None	No
6	212·BDJ	2I2 BDJ	Yes
7	505·MBU	SO	Yes
8	797·REA	797 R A	Yes
9	986·PGV	None	No
10	398·FSA	39 FSA	Yes
11	132·NZW	W	No
12	(A) 795·LAV	None	No
13	619·MPN	619	Yes

14	359·POY	3S9	Yes
15	253·PUZ	None	No
16	AW·827	AW 827	Yes
17	478·RDE	478 RDE	Yes
18	617·RFU	6I7 RF	Yes
19	406·DFR	4 R	Yes
20	387·RTS	R 5	Yes
21	655·GVL	65S L	Yes
22	879·RHW	879 RH	Yes
23	724·NZS	724	Yes
24	CXB·079	CXB 079	Yes
25	803·HVP	80 H	Yes
26	387·KHN	387 N	Yes
27	3CA·995	None	No
28	992·NMN	9	No
29	377·NJR	None	No
30	176·HWO	176 HWO	Yes
31	672·NSJ	672	Yes
32	407·PHL	None	No
33	503·RUL	None	No
34	503·RUL	5O3 RUL	Yes
35	824·RVF	824 RVF	Yes
36	6C·3795	None	No
37	559·PBM	None	No
38	520·PMZ	52O PM2	Yes
39	335·RNK	None	No
41	556·NMG	None	No
42	891·RRE	8I R	Yes
43	355·GGP	None	No
44	668·NPG	None	No
45	WN·9504	WN 504	Yes
46	410·PSE	4IO PSE	Yes
47	124·NOF	None	No
48	178·JKZ	None	No
49	(A) 298·LCT	LC	Yes
50	616·HAW	HAW	Yes
Total		Average characters: 2.48	% Yes: 58

Table 3. Data from program

5. OBSERVATIONS AND CONCLUSIONS

As can be seen from the data in Table 3, the program was generally successful in identifying the location of the license plate. The character recognition, however, failed to

read most of the license plate. The reason for this is the character selection function (once the potential license plate rectangle is selected and cropped) is extremely crude. A visual inspection of the intermediate files that the program produces revealed that a majority of the correct license plate candidates were cropped and thresholded correctly (see Section 3.4), and did indeed show the value with the correct dimensions. This implies that the character selection operation simply didn't pick them out correctly. When they were manually extracted and fed to the neural network, they produced high confidence answers. The following figure demonstrates the plate from photo #48 and the breakdown of the license plate cropping.

Figure 10. Character detection failure

The use of the neural network to select the appropriate rectangle, however, is justified by the fact that only in three (#4, #11, and #28) of the trials in which a rectangle that contained letters that had been correctly identified was not reported as the location of the license plate. The other candidate rectangles which did not contain pixels that resembled letters scored much lower confidence levels in all cases but these three. A next step might be to require a minimum confidence level for the program to respond with an answer. The neural net was also in many cases incapable of distinguishing between letters and similar numbers, common mistakes included: 1 = I, 2 = Z, 0 = O and 5 = S.

Both of these limitations motivate future work. The basic proof of concept has been demonstrated, and with improvements to the individual modules of the program the recognition and detection rates should be improved considerably. The neural network could potentially be made smaller without sacrificing accuracy. The letter detection algorithm needs to be redone and a variety of other performance tweaks can be performed.

7. REFERENCES

1. Draghici, S., (1996). A neural network based artificial vision system for license plate recognition. http://www.cs.wayne.edu/~sod/ijns1997.pdf
2. Haykin, S., (1999) Neural Networks: A comprehensive foundation, Prentice Hall, Upper Saddle River, New Jersey

3. Marzuki, K., Tahir, A., Tay, Y.H., Yap, K.M., (1998) Vehicle license plate recognition by fuzzy artmap neural network. http://www.cairo.utm.my/publications/ksyap_wec99.pdf
4. McDonald, J., (1998) The Hough Transform: Explained and Extended, http://www.cs.may.ie/~johnmcd/SNHT/
5. Ran, B., Liu, Henry X., (1999) Development of A Vision-Based Vehicle Detection and Recognition System For Intelligent Vehicles. http://www.its.uci.edu/~hliu/TRR_99_Vehicle.pdf
6. Skapura, D.M., (2002) Building Neural Networks, ACM Press, New York, New York.

Olfaction with Neural Networks

Boting Zhang
Yale University, Department of Computer Science
New Haven, CT 06520

ABSTRACT

The nonlinearity of neural networks suits them well for categorization tasks. One particular application is olfaction, which, in its one-dimensionality, perhaps lends itself more immediately to the use of networks than does image processing. Below, I'll be discussing the procedures I used to train a neural network to distinguish among the vapors of 14 beverages. Two chemical sensors were used — one for organic solvents and another for combustible gases. In its simplest form, the network is a basic feed-forward network with a single hidden layer, but the network can be expanded to less basic variants to allow for memory of previously sensed data. Backpropogation was used to train the network; a successful network was able to correctly classify the scent.

1 INTRODUCTION

1.1 Data Acquisition

Three sensors were used in this experiment: (1) a carbon monoxide sensor, taken from a home carbon monoxide detector; (2) a sensor from Figaro Electronics for combustible gases, including hydrogen; and (3) another Figaro Electronics sensor, this one detecting food vapors like ethanol. The maximum voltage output of each of these sensors was set to 5V for reasons to be discussed later. A high sensor output voltage indicated a greater concentration of the vapors it detected. Details about the three sensors are given in Table 1(a).

Nonalcoholic beverages were chosen because they differed quite subtly in sensor readings, making the experiment more interesting. Since these samples emit relatively low concentrations of the vapor, calibrating the sensors to 5V resulted in a voltage output of no more than 1V in the presence of a beverage. Table 1(b) lists the beverages chosen; sensor data with no beverages present was also recorded.

To keep the measurements controlled, all three sensors were attached to a common frame. When the frame was propped atop a tripod of three capped film canisters, this allowed a beverage-filled open film canister to be slid directly underneath, with about 3mm of space between the top of the canister and the sensors. Such exact conditions ensured reproducible sensor readings.

The sensor readings were interfaced to a computer through a data acquisition card, which read the output voltages once every 100ms and wrote the data to a file. Regular 100ms samples of the output for each beverage were taken for periods of 15–25 seconds, resulting in 150–250 sensor readings per beverage type. The range of these chemical sensor readings for each beverage is graphed in Figure 4; further sampling produced no information, for the readings remained within the ranges depicted.

1.2 Objectives

The nature of the vapors made the carbon monoxide sensor of little use; in fact, its irrelevant data produced inaccurately good results, because it recorded mostly extraneous environmental information, which unfairly helped the network due to the limited sample size. For this reason, carbon monoxide sensor data was left out for the latter part of this experiment, reducing the number of sensors to two. What makes the subsequent findings most interesting is the capability of neural networks to identify different substances, given only these two sensors.

The tasks for the networks fell under three different categories: identifying the beverage, identifying its brand, and categorizing the flavor of the beverage. For each type of task, categories were assigned unique numbers. So a network that successfully distinguished among beverages would output the beverage's unique identification number upon being given as input a sample of the beverage's sensor data. Further detail on this is given in Table 1(b) and the next section.

2 OLFACTION WITH A BASIC FEED-FORWARD NETWORK

2.1 Setup

The current applications of neural networks in olfaction [1] are simple feed-forward networks with as many output nodes as there are different vapors. The perfect output, which the network trains itself towards, indicates the presence of a vapor in category i (where i is as listed in Table 1(b)) with a value of 1 at output node i. If vapor i is not present,

output node i should be 0.

In determining the identity of an odor with the simple olfactory neural network, only a single data sample is input into the network. And so the system looks like this:

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \longrightarrow \boxed{\text{Network}} \longrightarrow \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{15} \end{pmatrix}$$

where x is a vector of sensor readings at some time t, and y is the network output as described above.

So during training, exemplars are randomly chosen from the set of all vapor samples taken at all times. An exponential threshold function was used, and a momentum constant of .4 introduced to speed the training process. The threshold function required the very low scaling factor of .25 for the network to converge.

2.2 Results

Given this simple training, the network was able to distinguish six vapors (Fig. 4) extremely well after 1 million backprop iterations, and could even distinguish all 15 vapors (Fig. 4) fairly well.

The outputs of these networks could be easily modified to categorize the beverages by brand or flavor, merely by changing the output against which they were trained. The networks performed better categorizing by brand (Fig. 4) than by flavor (Fig. 4). This makes some sense; to a system with only two sensors, the common chemical content of beverages made by the same company is probably more noticeable than subtle flavor distinctions.

One very interesting outcome arose when the network was asked to categorize vapors it was not trained on. For this, the network was trained to identify all the beverages by brand except Vanilla Coke and week-old Coke, the exemplars of which it was not allowed to see during the training process. Figure 4 graphs the network output when it was then run on all 15 vapor types. Week-old Coke was correctly grouped with the fresh Coke, but Vanilla Coke — which should also have been grouped with this set, since the network was categorizing by brand — mapped to something sharing the qualities of Cokes and Frappuccinos (most notably, the Vanilla Frappuccino). This may have been the result of the caffeination of the two beverages; however, it may also have been an interesting flavor recognition when it ought to

have been categorizing by brand.

3 BASIC FEED-FORWARD NETWORK WITH MEMORY

3.1 Setup

One property of olfaction is the length of time required by humans to identify certain scents. Is it perhaps the case that many samples taken over a period of time would help our network in correctly classifying flavors?

In a network with memory, the network has access to information about current sensory inputs *as well as information about sensor data from previous time steps*. This method is much more robust because it smooths the error caused by air-current-induced fluctuations in sampling data, and would perhaps help extract the relevant data.

The simplest implementation of memory in a network is a vector of sensory inputs taken from contiguous vapor samplings. The number of sets of sensory data taken at a time corresponds to the size of the network's memory, which we'll call M. The network, then, is in effect taking as input a matrix, the size of which is proportional to its memory size M.

The neural network was trained the same way, because this is still a simple feedforward network trainable by backpropagation:

$$\begin{pmatrix} x_1(t) & x_1(t-1) & \dots & x_1(t-M) \\ x_2(t) & x_2(t-1) & \dots & x_2(t-M) \end{pmatrix} \longrightarrow \boxed{\text{Network}} \longrightarrow \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{15} \end{pmatrix}$$

3.2 Results

When trained to identify beverages by brand, this memoried network was able to perform much better than the network without memory. Although this is not directly visible from the distribution of its output across all input 2-second blocks (Fig. 4), an examination of the distribution of this output (Fig. 4) shows that nearly all of the test cases were correctly classified, and only a few were incorrectly categorized. Categorization by flavor was also more successful, as shown

in Figure 4.

4 CONCLUSION

Adding memory to the network proved promising. Although flavor was still harder for the network to identify than brand — a result attributable to the fact that companies maintain the same ingredient concentration throughout their line of products — performance was improved significantly. Brand identification accuracy, meanwhile, was improved to 98% (a result shown graphically in 4).

However, the implementation of memory here was rather impractical. It is clearly not biologically accurate: the body's chemical sensors do not have access to previous chemical concentrations. Rather, real memory must be implemented on the outputs. What would be interesting to see in the future would be how well olfaction is performed in a system of two networks, the first system of which operates on the direct sensor input, and the second system operating on the network output from the first system (the real-time output) and taking other input from previous outputs of the first system:

$$\begin{pmatrix} x_1(t) \\ x_2(t) \end{pmatrix} \longrightarrow \boxed{\text{Network}} \longrightarrow \begin{pmatrix} y_1(t) \\ y_2(t) \\ \vdots \\ y_{15}(t) \end{pmatrix} \searrow$$

$$Y(t)$$

$$\begin{pmatrix} y_1(t-1) & y_1(t-2) & \dots & y_2(t-M) \\ y_2(t-1) & y_2(t-2) & \dots & y_2(t-M) \\ \vdots & & & \\ y_{15}(t-1) & y_{15}(t-2) & \dots & y_{15}(t-M) \end{pmatrix} \nearrow$$

Now vector $Y(t)$, adopting the same category format as $y(t)$, is the true output of the system. In a real biological system with 10,000 sensors, this is more likely how olfaction is organized.

References

[1] P. E. Keller. Electronic noses and their applications. *1995 Northcon Conference Record*, October 1995.

Sensor		Main Chemicals Detected	Others
1		CO	
2	TGS814	CH_4, ethanol, propane, butane, H_2	CO
3	TGS822	butane, hexane, benzene, ethanol	CH_4, CO

(a) Sensors. Note that sensor #2 primarily detects combustible gases, while #3 detects food vapors. The numbers in the first column will be used to identify the sensors in this paper. The second column lists the part number for the Figaro Electronics sensors.

Vapor	Beverage	Brand	Flavor
None	1	1	1
Vanilla Frappuccino	2	2	2
Mocha Frappuccino	3	2	3
Cherry Snapple	4	3	4
Lime Snapple	5	3	5
Orange Tropicana	6	4	6
Apple Tropicana	7	4	7
Black Cherry Soda	8	5	4
Lime Soda	9	5	5
Strawberry Soda	10	5	8
Root Beer	11	6	9
Vanilla Coke	12	6	2
Coke	13	6	10
Week-Old Coke	14	6	10
Week-Old Orange Tropicana	15	4	6

(b) Vapors. Here are listed the 15 scents used for the experiment, along with their category assignment for the various identification tasks. A network that was categorizing by beverage, for example, would be expected to output an 8 upon being given as input a sample of sensor data recorded for Stewart's black cherry soda. If trained to categorize by brand, however, it would be expected to output a 5, alongside the other Stewart's sodas.

Figure 1: Number References. A list of numbers used to refer to the sensors and categories in the graphs throughout this paper.

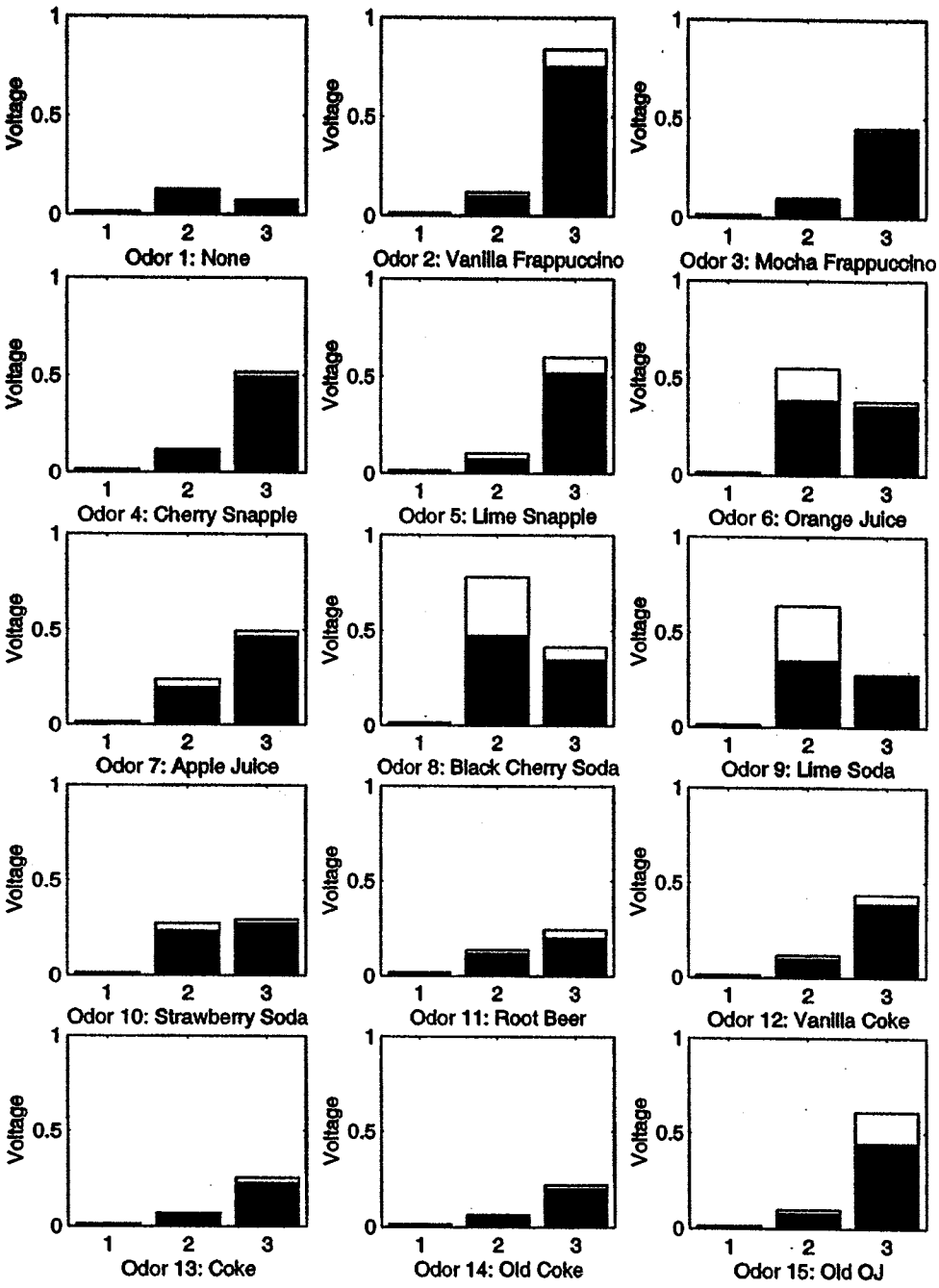


Figure 2: Sensor data distribution for various vapors. Graphed here are the sensor readings for each scent, where the sensors are identified by number. The first bar of each graph, for example, shows the carbon monoxide sensor voltage in the presence of a certain scent. Table 1(a) details the three sensors used. The darker bars in the graphs depict the minimum sensor readings for a scent. Meanwhile, the entire bar, outlined in black, shows the maximum sensor readings. The white boxes, then, depict the range of values taken on by the sensors for a given scent.

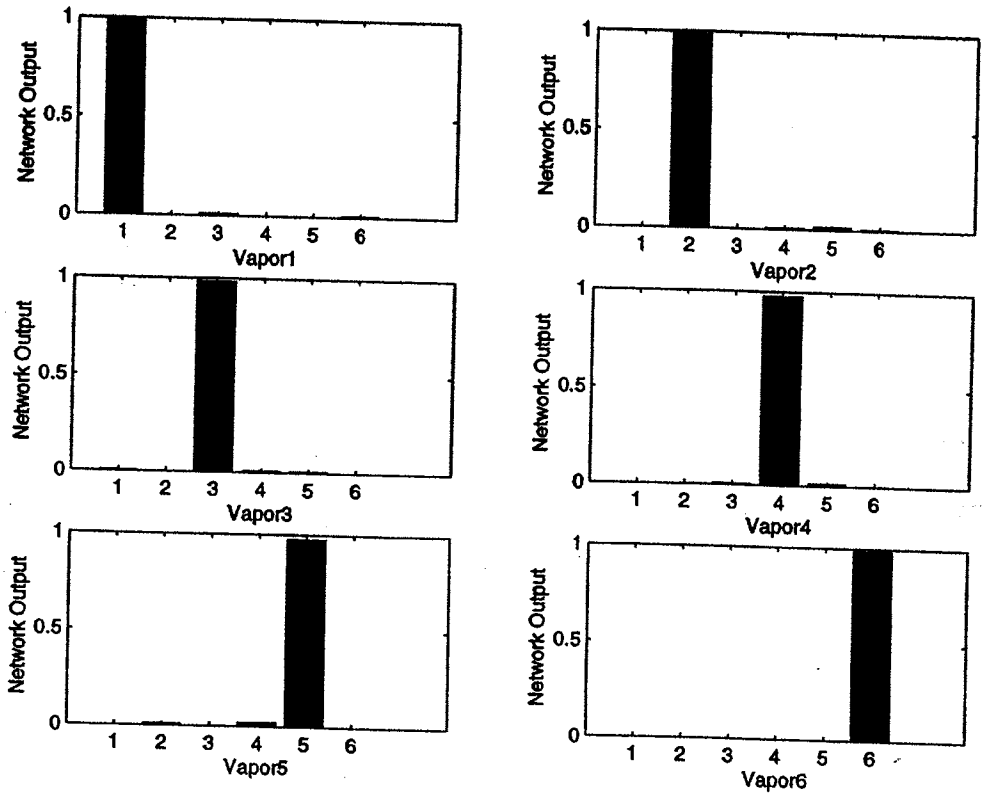


Figure 3: Six Vapors, Beverage Identification. This data graphs the output of the network with a randomly selected vapor sample as input, after having been trained on only these six for 1 million training cycles. The vapors, here identified by number, are detailed in Table 1(b).

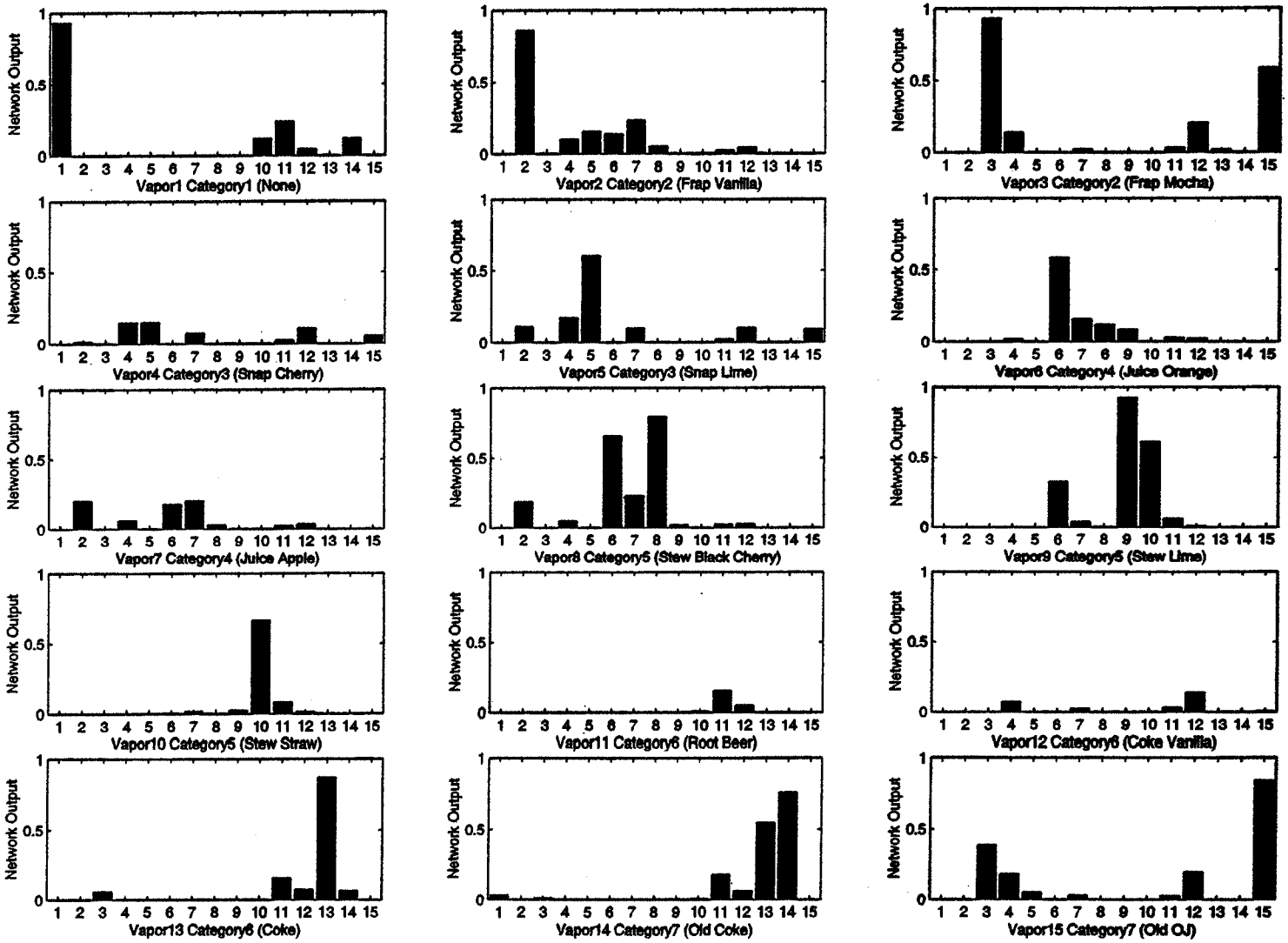


Figure 4: Fifteen Vapors, Beverage Identification. The network was trained on all 15 vapors for 10 million training iterations. Graphed is the network output from a single randomly-selected sensor data sample; that is, this shows the network output vector y given only sensor readings $x(t)$ at some random time t. The correct output is given by the Vapor number below each graph; the Category number is only for reference. The network performs rather well, although it had an understandable amount of trouble telling week-old Coca-Cola from the fresh kind, as well as some difficulty identifying the Stewart's sodas (Vapors 8-10).

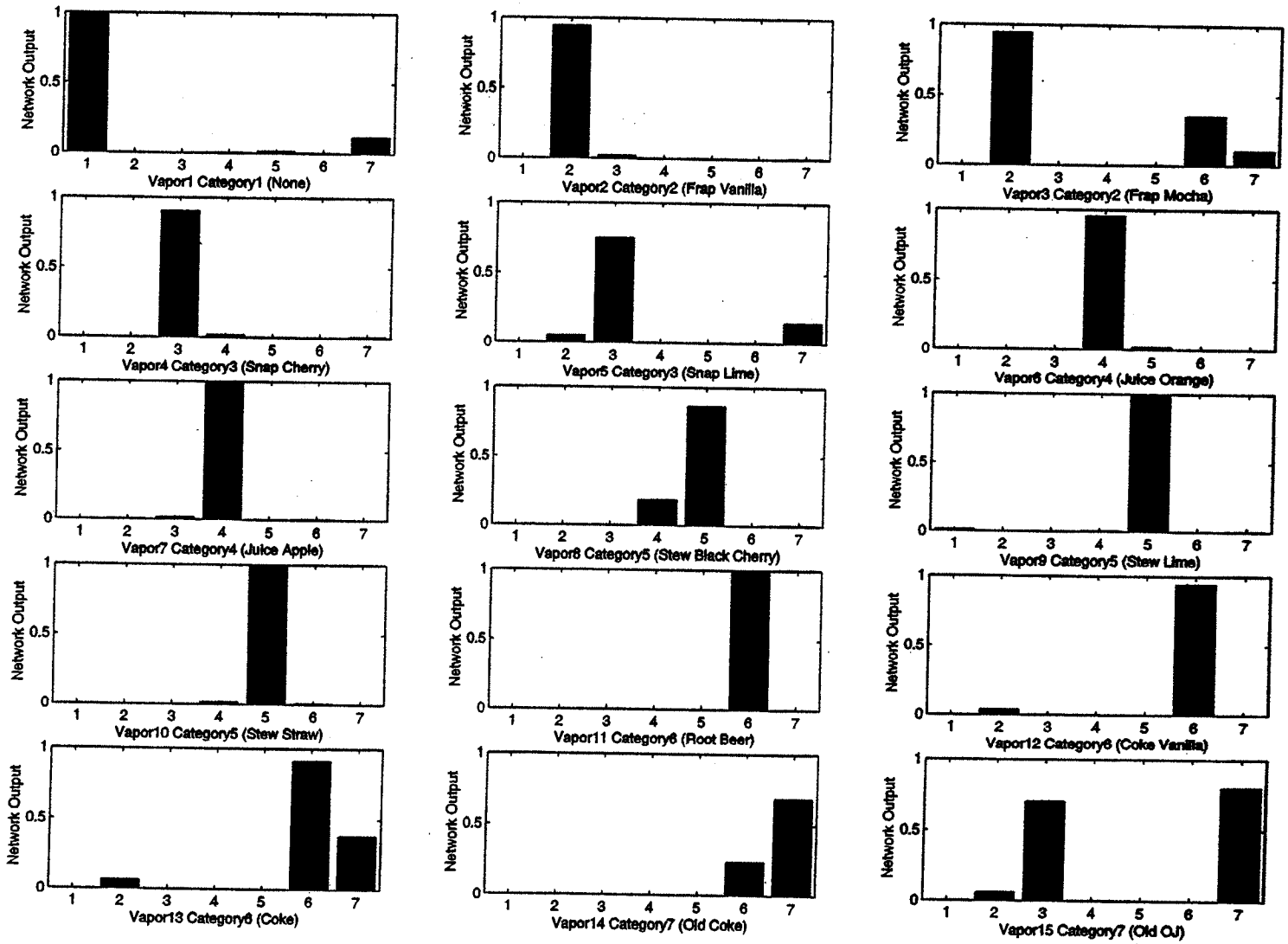


Figure 5: Brand Identification. This network was again trained on all the vapors, but was now trained to output the brand of the vapor. Again, this graph only depicts the outputs from a single data sample per scent. NOTE: here, week-old Coke and week-old orange juice were put together in a separate category, although they clearly do not belong together. This was done to ensure that these unexpectedly good results were legitimate, and not a fluke caused by an excessively small sample size. The graphs of the outputs does indeed ensure legitimacy of the other categorizations, for the network was much less successfully able to group the old beverages together, instead classifying the fermented orange juice with the Snapples and the old Coke somewhat with the fresh Coke.

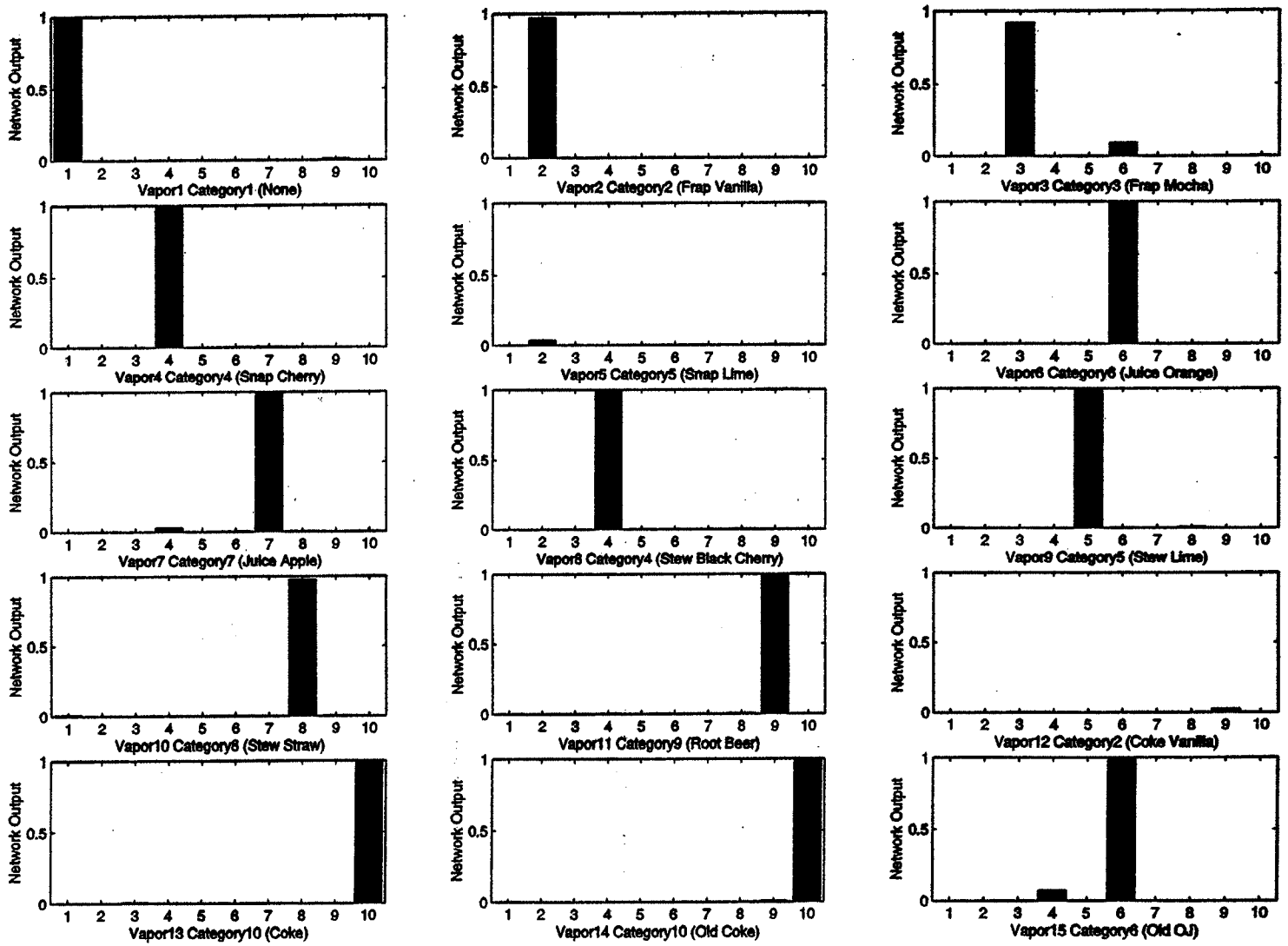


Figure 6: Flavor Identification. This network was trained to categorize by flavor. This approach was unsuccessful; the network was only able to categorize one member of a flavor category, but never both. This result is not surprising either, when one considers that the similarity of ingredients across the range of beverages made by the same company is probably greater than the similarities between Vanilla Coke and Vanilla Frappuccino. However, the next figure shows an interesting related result.

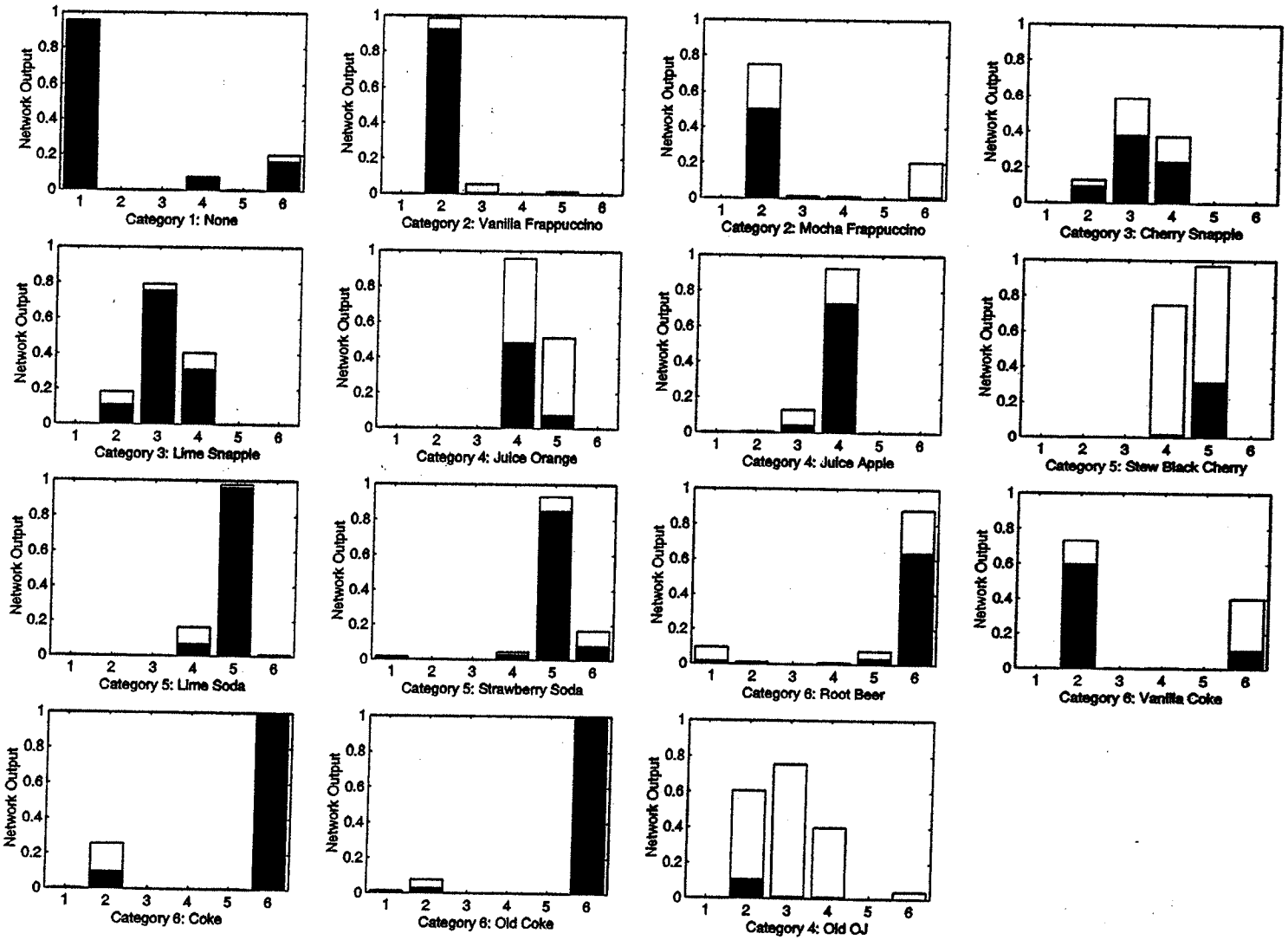


Figure 7: Brand Identification of Unrecognized Substances. When a network was trained on all the vapors except for Coke and Vanilla Coke, and subsequently tested on all vapor data, its output for Vanilla Coke wavered between Coke and Vanilla Frappuccino, tending much more towards the Vanilla Frap than did the fresh Coke, which had also not been trained for. The data on this graph is fuzzier than in previous graphs, because the carbon monoxide data has been removed; from here on, the inputs from only two sensors were used as input. Also, the range of output values for *each* sample data is graphed. Where previous graphs only depicted output for a single randomly-selected set of sensor values, this represents the network output for each $x(t)$, for all t. As in Table 4, the white boxes depict the range of values taken by the output during the entire time period.

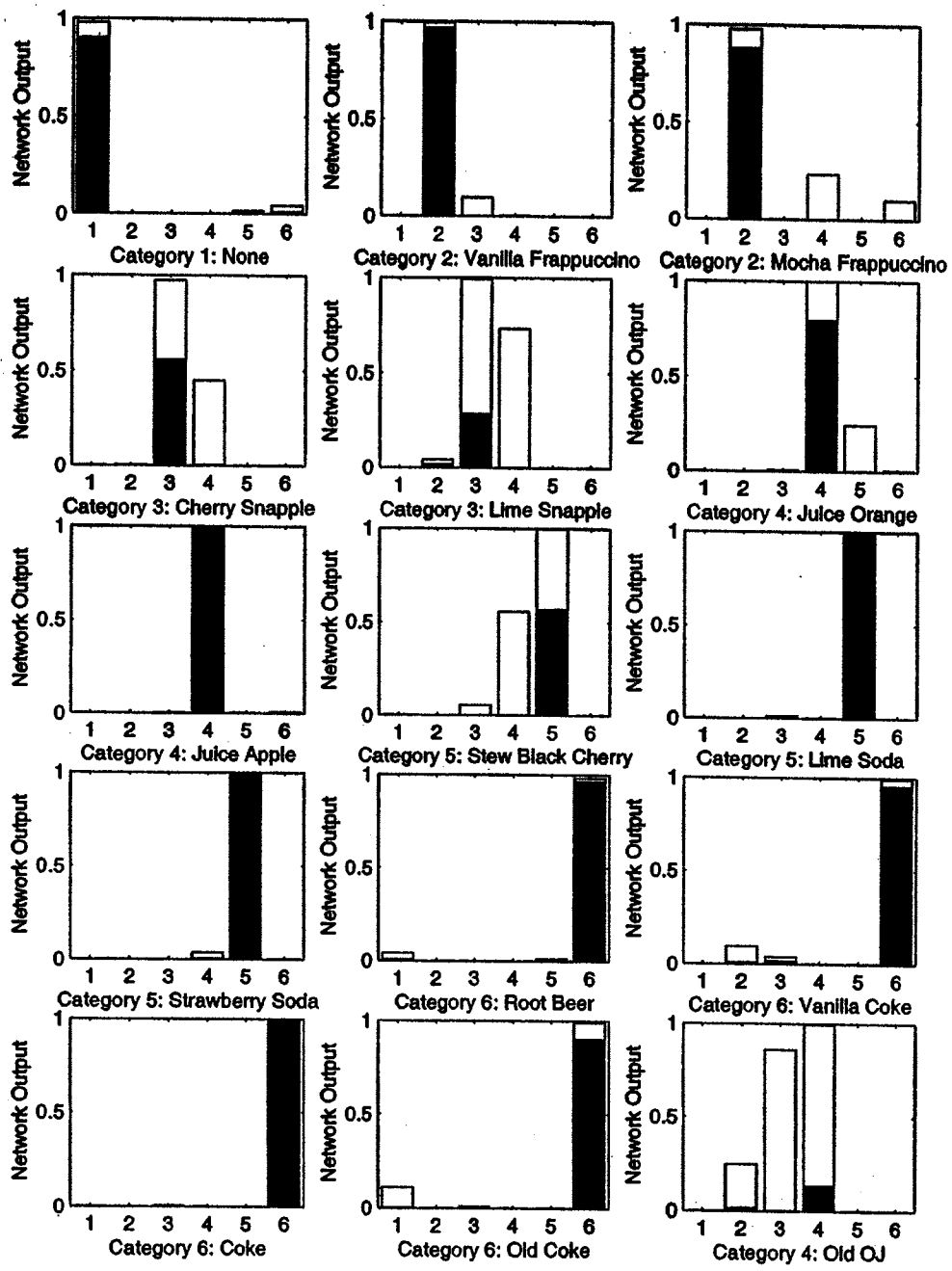


Figure 8: Brand Recognition With Memory. Even better than it already was. Although the best- and worst-case answers were not very good, they are extreme outliers. This can be better seen in the next graph.

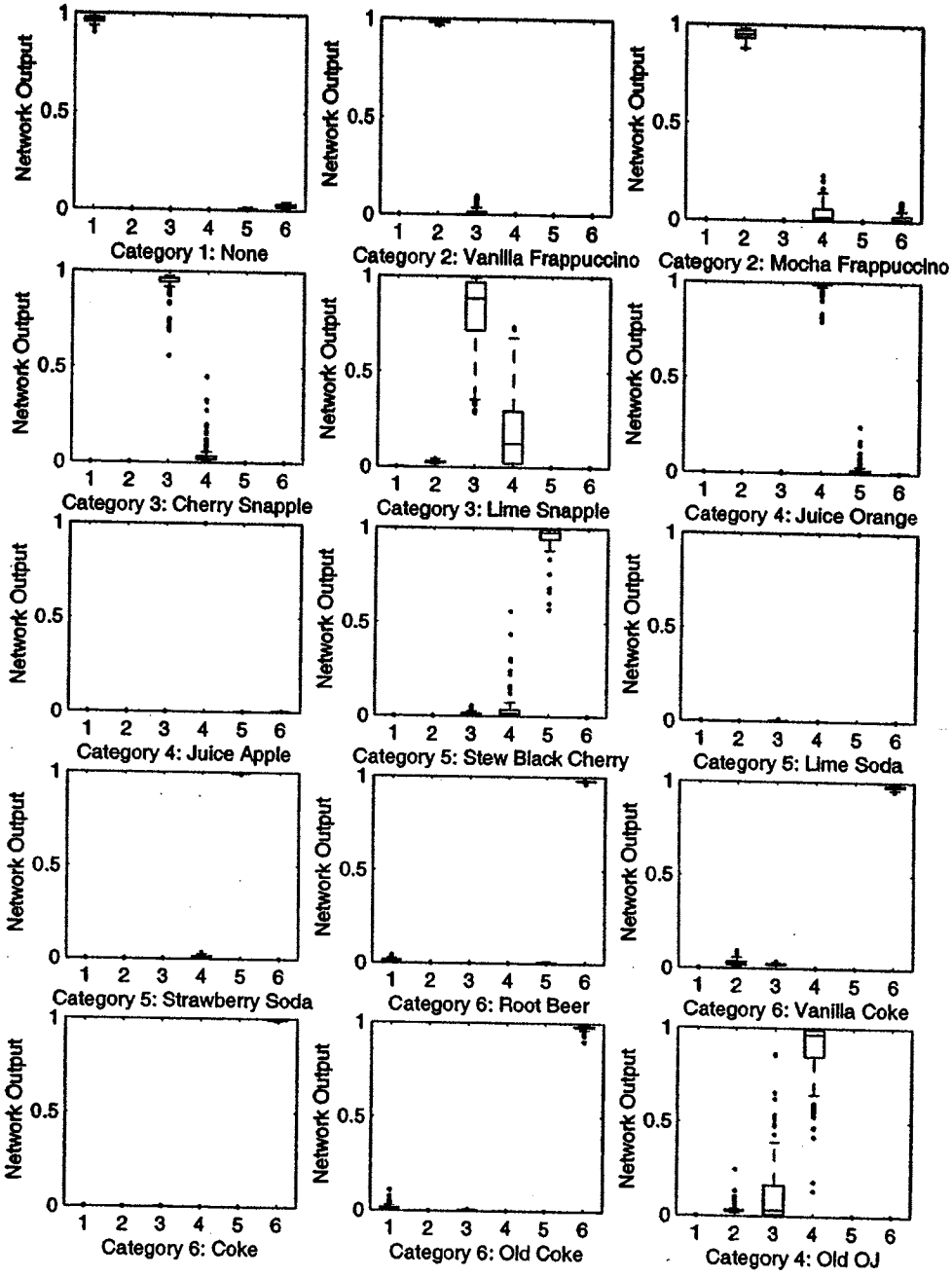


Figure 9: Brand Recognition With Memory. This plots exactly the same information as the previous graph, but shows the distribution of the outputs.