Neural Network Applications

Willard L. Miranker

April 2006
TR-1357

Table of Contents

Effects of Variable Learning Rates

Erik Brown
Yale University, Computer Science
New Haven, CT 06520

Abstract
We study the convergence behavior of neural networks as a model for long term human learning and the effects of neural atrophy. The neural networks undergo supervised learning in which the learning rate parameter, η, is varied with each epoch of training. In particular, the learning rate parameter is exponentially decreased over time to simulate the effects of neural atrophy. The convergence properties of these nets are then compared to those of networks undergoing the same supervised training, but which have a constant learning rate parameter for all of the epochs. We found that the networks with a variable learning rate generally took longer to converge to a final configuration; however their output was more accurate.

1. INTRODUCTION

The topic of human learning has been at the forefront of cognitive science since its infancy. There is something alluring about the many intriguing peculiarities in the learning behavior of humans. For example, it is commonly known that learning new cognitive tasks, especially linguistic tasks, is performed more quickly by children than middle-aged and elderly adults. However, it is also widely hypothesized that elderly adults make up for short comings in memory capacity and speed of information processing by using contextual cues from the environment (Jerger & Martin, 2005). This leads to the question why does the ability to learn new information tend to decrease with age?

Past research using similar models has shown how various factors influence short term development of learning tasks. These studies focused on factors such as other children in the household and also age-weighting the instructions of different family members (Barley, 2004; Logan, 2004). Since these studies attempted to understand short term development of children, they only allowed the children to learn at one rate for all training inputs. In this research we have developed a model of long term learning that incorporates neural atrophy. In other words, as the learning process is continued throughout a lifetime the ability to learn new material decreases. This is modeled by continually decreasing the value of the learning rate parameter, η, during the learning process (see section 2.2). With this model we hope to test hypotheses of age-related learning differences such as that mentioned above.

2. THE MODEL

2.1 Roles

The experiments involve two agents, a student and a teacher. A student is a person learning a new task while the teacher is the one responsible for providing feedback to the students to aid in the learning process. Each student is represented as a neural network and the training of that network corresponds to the student learning a task. Learning is modeled as the student's ability to recall information that the teacher has previously presented. Namely, each student is trying to learn the correct association between an input and its corresponding output supplied by the teacher. During the process the teacher's behavior never deviates: given the same stimulus at different times, the teacher will always produce the same desired reaction. On the other hand, the behavior of the student will change dramatically. Initially, the students are uninformed and know nothing about what they will be learning. Over time, they will learn the task and start to emulate the responses of the teacher.

2.2 Neural Atrophy

Neural atrophy refers to loss of plasticity in the brain. As neurons age they lose the ability to change their synaptic response to action potentials and hence learning of new information is degraded. To model this behavior, we decrease the learning rate parameter, η, during the learning process. This simulates the onset of neural atrophy since the smaller η is the less the network changes its weight configuration. Since atrophy is a process that takes years to set in, our research gives a long term model for associative learning of new tasks.

3. SPECIFICATIONS

3.1 Network Configuration

Our work uses networks being trained to reproduce a desired output to model students learning a task from a teacher. To keep the model as simple as possible, the neural networks only have one hidden layer. The number of nodes in the hidden layer was initially varied to find the configuration that was the most stable. We chose to use a network with four binary input nodes, four nodes in the hidden layer, and one node in the output layer.

3.2 Training

The training of the neural networks uses the back propagation algorithm in which the change in the weight vectors is proportional to the gradient of the error surface (Haykin, 1999). Furthermore, this algorithm uses credit assignment to calculate how much each layer contributed to the total error of the output layer, and then changes the weights of each layer accordingly. An anti-symmetric activation function with near unity gain at the origin is used. Both the initial weights and the desired outputs are chosen to lie in the linear portion of the activation function to avoid early saturation of the weights during the learning process. Since we use four binary input nodes in this model, there are only 16 different inputs possible and therefore 16 different desired outputs that need to be chosen for the one output node.

For each experiment, 100 trials with different student-teacher pairs are conducted and the quantities of interest are ensemble averaged to negate the effects of the randomly sampled initial conditions. All of the different experiments are trained with the same set of 100 student-teacher pairs so that comparisons can be made between experiments. We use sequential training for all of the experiments, in which a student is able to learn from each individual input. For each epoch of learning, the students are presented with a random sequence of the 16 possible inputs.

3.3 Variable Learning Rate

For the experiments conducted with a variable learning rate, it followed a set evolution as the epochs of training progressed. Namely, it went by the equation:

$$\eta(t) = (\eta_0 - \eta_\infty)e^{-t/\tau} + \eta_\infty \,, \qquad (1)$$

where t is the epoch index, η_0 is the initial value, η_∞ is the asymptotic value, and τ is a characteristic time constant. For these experiments, this equation was reduced to a one parameter family by setting η_∞=0.005 and τ =2000, leaving only η_0 to be varied.

3.4 Convergence

The criterion for convergence of an experiment is determined by measuring the running standard deviation of the ensemble averaged error as a function of epochs of training. Each run is limited in length to 10,000 epochs of training regardless of the final error. The running standard deviation is computed using a window size of 500 epochs. An experiment is defined to be converged when its running standard deviation stays below the threshold of $8\text{x}10^{-4}$ for the remainder of the 10,000 epochs. The index of the last epoch for which the running standard deviation goes below the said threshold is when convergence is defined to occur. This definition requires that each run be continued for the maximum number of epochs, and the epoch where convergence occurs will be calculated after the run is conducted. In this way, for a net that converges we can associate to it the number of epochs to convergence as well as the ensemble averaged error at convergence. The error is defined by:

$$\left|\vec{e}\right|^2 = \left|\vec{y} - \vec{d}\right|^2 \,, \qquad (2)$$

where y is the output from the student and d is the desired response given by the teacher. If a run does not satisfy the convergence criteria, it is defined to have a convergence time of 10,000 epochs with its associated error at that point.

4. EXPERIMENTS

4.1 Constant Learning

This group of experiment serves as the control group of the study. The learning rate is kept constant, and the learning process is monitored. In particular, we record the ensemble averaged error in the students' responses compared to their teachers' error as a function of time (epochs). As stated above, the student-teacher interactions are continued for 10,000 epochs. We conducted 13 experiments, each one utilizing a different learning rate parameter in the interval [0.001, 0.2]. For learning rates above 0.2, convergence was not attainable for any initial configuration tried.

4.2 Learning with Atrophy

This group of experiments proceeds in a similar manner to the first, however this time the learning rate of the students is varied as a function of time as defined in equation (1), with parameters as defined in section 3.3. Again, we record the ensemble averaged error in the students' responses compared to their teachers' as a function of time (epochs). We conducted 12 experiments in which the initial value of the learning rate parameter was varied over the interval [0.01, 0.2]. For initial learning rates above 0.2, convergence was not attainable for any initial configuration tried.

4.3 Comparisons

Since the two groups of experiments are fundamentally different in that the learning rate parameter is treated differently in each case, it is difficult to compare data from each group. For our analysis, we have chosen to compare the two groups based on the initial value of the learning rate parameter. Although other schemes for comparison can be envisioned, such as the average value of the learning rate parameter, we believe that the initial value is most in line with the aims of this study. Since we are trying to investigate the effects of neural atrophy, it makes sense to look at two students who start with the same learning rate and then let one of them undergo atrophy by way of the decreased learning rate.

5. RESULTS

5.1 Learning Time

One value of interest is the time in which it takes the two groups of experiments to reach convergence. A natural hypothesis would be that the cases with atrophy would take longer to converge than the corresponding case without atrophy since the students in the atrophy group cannot reduce their error as fast. Figure 1 plots these results from all of the experiments. For small values of the learning rate, the two groups of students tend to learn the task in the same amount of time. However, as the initial learning rate is increased we do see that the atrophy group (those with a variable η) does take significantly longer to learn the task.

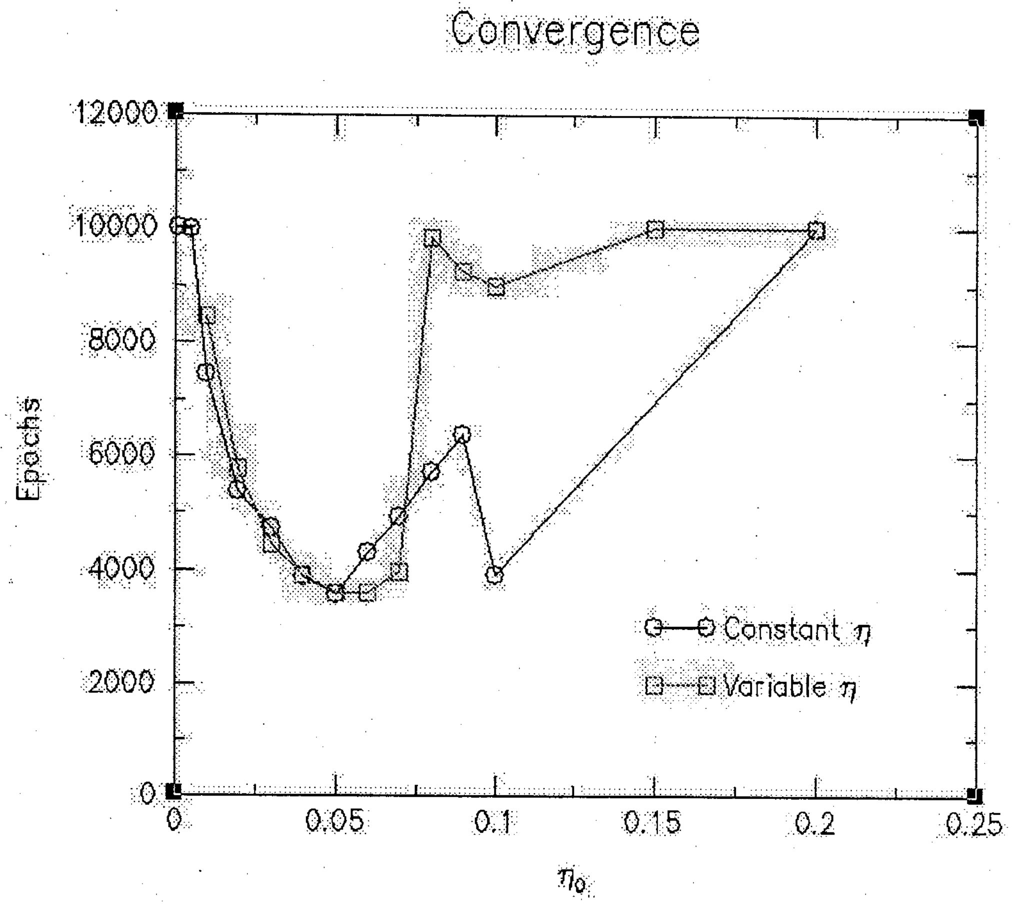


Figure 1. This figure shows the ensemble averaged number of epochs to convergence for the two groups: constant and atrophied learning.

5.2 Learning Accuracy

Another variable of interest from the experiments is how accurately the students have learned the task. This is reflected in the ensemble averaged error at the point of convergence. Figure 2 shows these results from all of the experiments. Again, for small initial learning rates, both groups perform about the same. However, the results at higher learning rates are unexpectedly reversed – the students with neural atrophy actually learn the tasks better. The average error for the atrophy group is significantly below that of the control group.

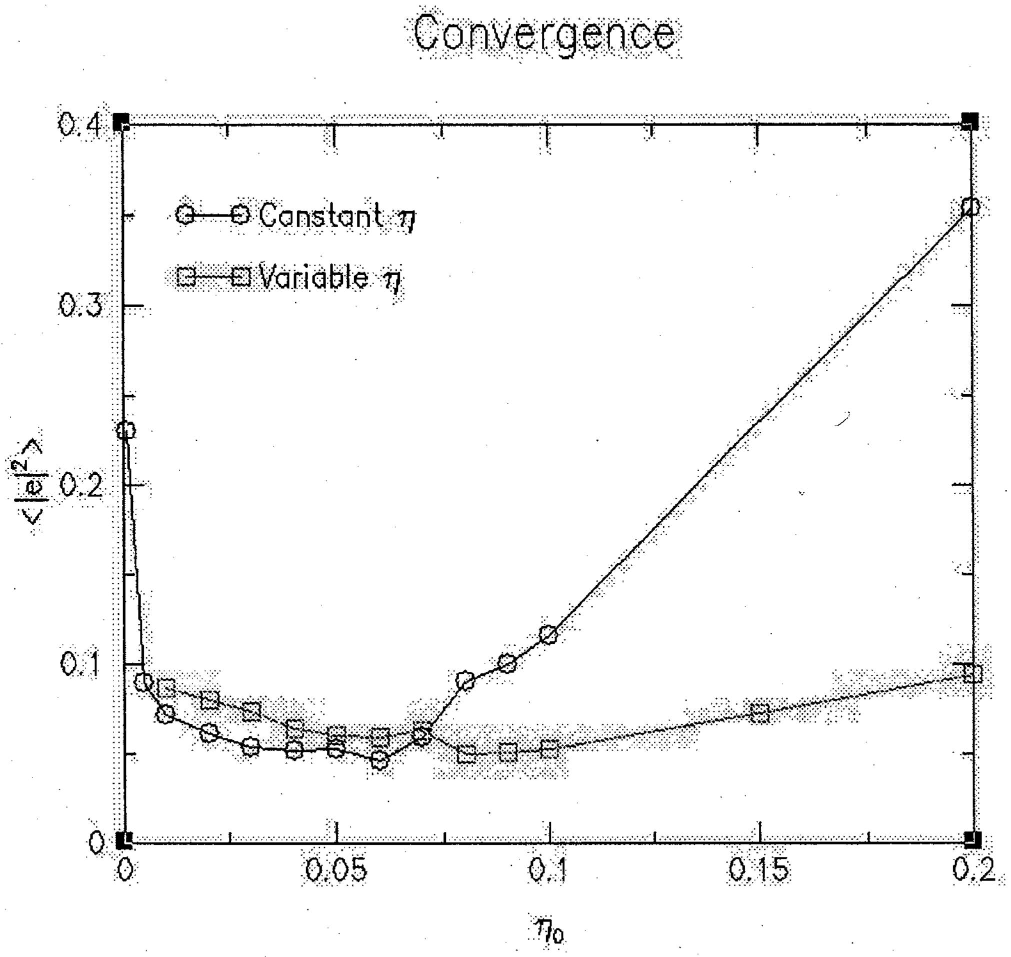


Figure 2. This figure shows the ensemble averaged error at the time of convergence for the two groups: constant and atrophied learning.

5.3 Algorithm Effects

For both figures 1 and 2, all of the curves have a minimum somewhere in the middle of range of the learning rate. Since this effect shows up in both groups, it does not have anything to do with the topic of neural atrophy. Rather, it is an artifact of the learning algorithm being implemented. The back propagation algorithm is a gradient descent method as described earlier. In this context, the learning rate parameter, η, is acting as the step size taken on the error surface. Convergence corresponds to the algorithm finding a local minimum on the error surface. If η is too large, the algorithm can jump back and forth between different sides of a valley and never find the minimum. On the other hand, if η is too small the algorithm will head in the correct direction, but convergence will be slower based on the fact that each step is only changing its position by a small amount. This effect is responsible for the u-shaped curves seen in the convergence data.

5.4 Staged Learning

Another interesting result from this study was the pronounced appearance of staged learning in the atrophy group. As shown in figure 3, staged learning appears as a very slow initial decrease in the error followed by a very abrupt large decrease, and then another region of slower decrease or convergence. This effect is seen in a majority of the atrophy group but rarely occurs in the control group. As a comparison, figure 4 shows a representative graph of results from the control group. Although staging is frequent in the atrophy group, the sharp drop in error occurs at different times. The staging effect is only seen in data from individual student-teacher pairs. When the data is averaged over the 100 different pairs, the staging behavior is effectively averaged out since the characteristic sharp decline in the error occurs at different times for different pairs.

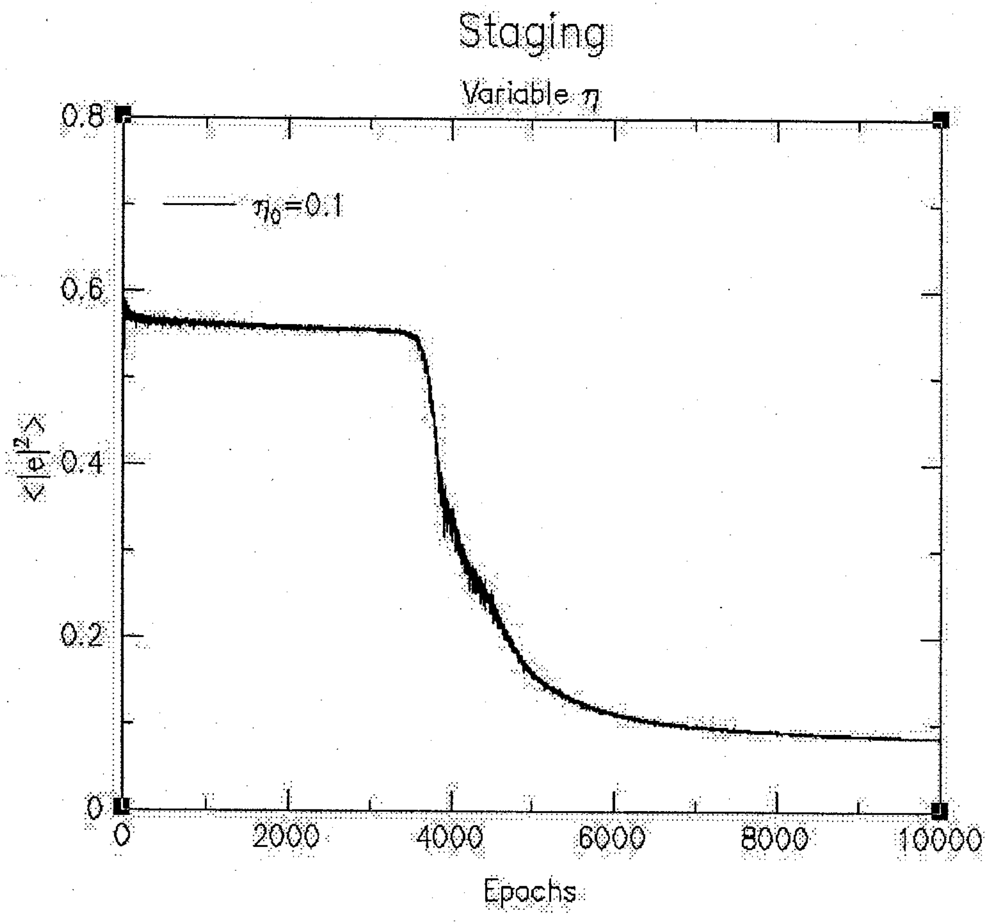


Figure 3. This figure shows the appearance of staged learning for one student-teacher pair in the atrophy group.

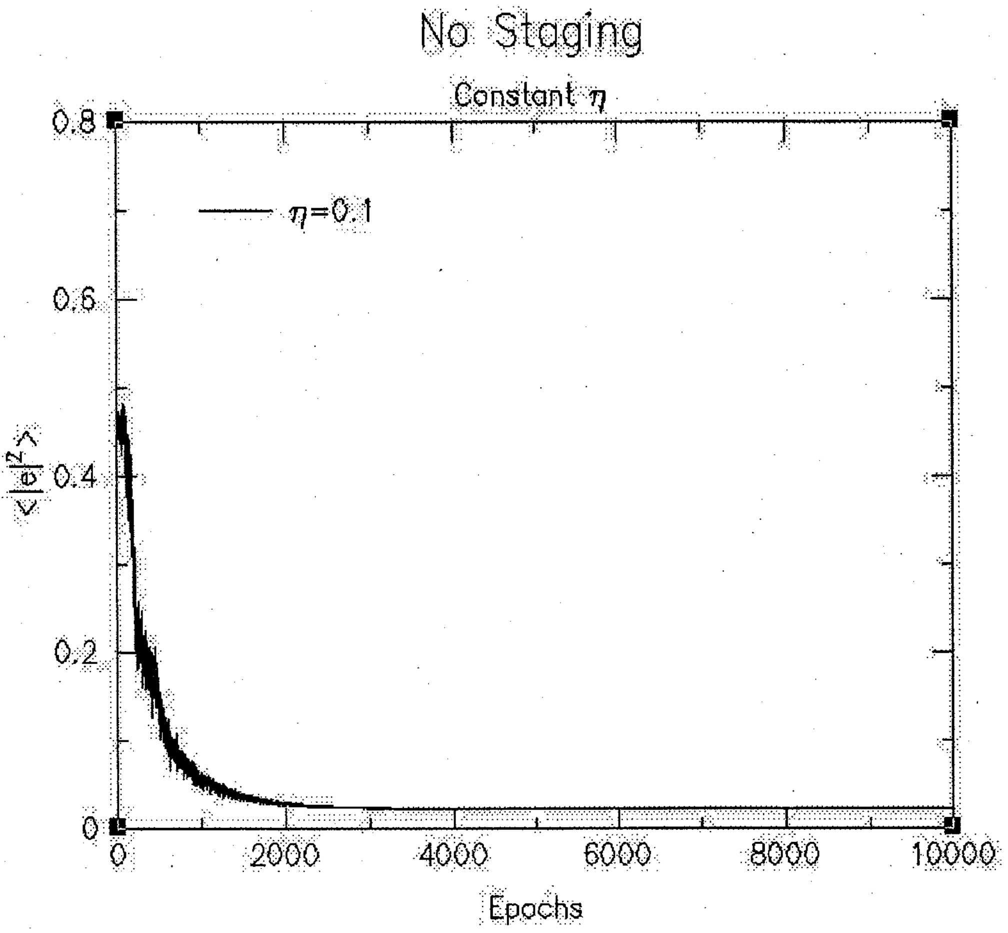


Figure 4. This figure shows a typical learning curve for one student-teacher pair in the control group.

Although humans are known to undergo staged learning, the results of this study do not indicate a link between staged learning and neural atrophy. Again, this phenomenon is most easily described as an artifact of the gradient descent. If the initial learning rate causes jumping between sides of a valley on the error surface, the learning is going to proceed very slowly. For the atrophy group however, the decreasing learning rate is able to stop that behavior abruptly as the step size goes below a threshold. After that point, the learning will occur rapidly since the network can easily follow the bottom of the valley down to a local minimum.

REFERENCES

1. Jerger, J., & Martin J. (2005). Some effects of aging on event-related potentials during a linguistic monitoring task. International Journal of Audiology, v.44, 321-330.

2. Barley, W. (2004). Modeling child development with colony learning. Yale University.
3. Logan, J. (2004). A game theoretic and genetic algorithm approach to modeling the emergence of mind in early child development with exposure to semi-developed parents. Yale University.
4. Haykin, S. (1999). *Neural Networks: A Comprehensive Foundation*. New Jersey: Prentice Hall.

A Modification of the Kohonen Learning Algorithm to Model the "Perceptual Magnet Effect"

Haben Michael
Linguistics Department, Yale University
New Haven, CT 06512

Abstract

A self-organizing feature map (SOFM) is used to model the *perceptual magnet effect*, a claim in the theory of learning sounds that a listener is more sensitive (in a sense to be made precise) to sounds farther away from the most commonly occurring sounds. From the standpoint of the feature map, the problem is to modify its associated learning algorithm to capture this *anti-Hebbian* effect: neuron density is to be least around the most frequently occurring input patterns, greatest around the least frequently occurring. On the other hand, the network still needs to categorize the sounds and organize the resulting categories according to acoustic similarity. This is achieved by the judicious pruning of neurons: the *Kohonen learning algorithm* is modified so that a certain number of neurons nearest to a certain number of the most active neurons are removed.

1. INTRODUCTION

When the vocal tract is articulated to produce a vowel, certain formants (resonant frequencies) are associated with the tract's physical configuration. It has been shown that the lowest three of these formants suffice for a listener's identification of particular vowels and allow differentiation among them; if a buzz like the one originating from the larynx is sent through a tube that is physically manipulated to resonate the same lowest three frequencies that are associated with a vowel, then the resulting sound at the other end of the tube will usually be perceived as that vowel. Therefore, these formants are often used to encode the vowels of a given language. (Borden, 2002; Stevens, 1972; In fact, in most cases the first two formants suffice; see Fig. 1, for such a mapping of the state of Canadian English vowels).

In a spoken language, there are continuous regions of formants that correspond to (i.e., "sound like") the vowels of the language. Somewhere near the center of each of these regions is the point corresponding to the prototypical or most frequently heard quality of a vowel. Therefore, the space of possible formants can be thought of as being divided into several regions, with the formants within a fixed region corresponding to the possible articulations of a fixed vowel. American English is often said to possess 12 of these regions, or "phonemes," whereas the example of Canadian English shown in Fig. 1, using formant region mean values to specify the prototypical vowel points, has 10. The mapping of the continuous formant space into these partitions is an aspect of what the experimental literature refers to as "categorical perception." Given the many environments that a vowel may occur in, and variations in its formants that a speaker may produce, a fluent listener almost without fail associates that production with the intended phoneme (Estes, 1994; Lacerda, 1995; Kuhl, 1991).

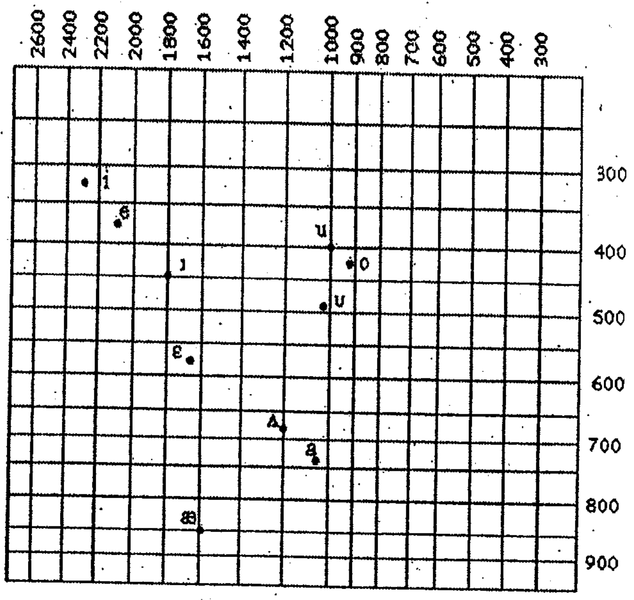


Figure 1: A graph of the space of formant pairs showing the vowels of Canadian English. The vertical axis represents the first formant and the horizontal axis represents the second formant (both in Hertz); the points represent prototypical values, or category centers, of the labeled phonemes. (A scaling conventional in phonetics is used here, which makes the layout of vowels here approximate the layout of the positions of the tongue when articulating these vowels.) http://www.ic.arizona.edu/ lsp/Canadian/canphon2.html.

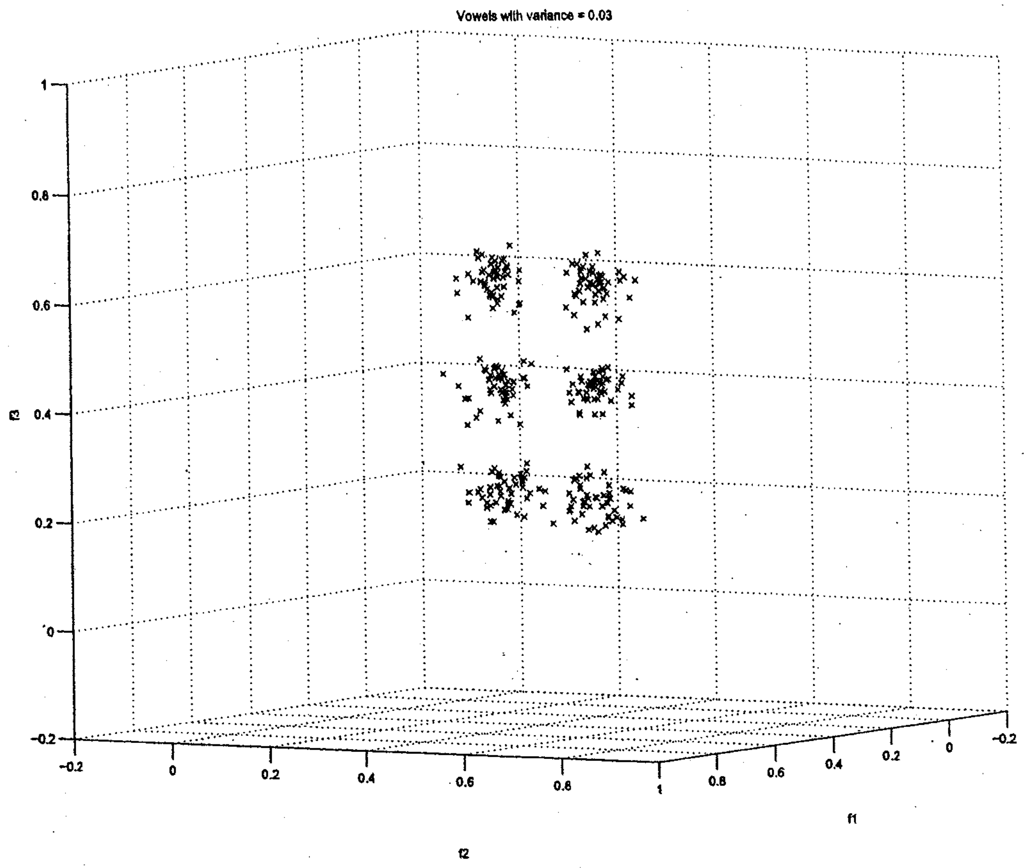
An observed effect of this categorization is that a listener is not equally *sensitive* to all regions of the formant space, in the following sense. According as acoustic signals (encoded by their formants) are nearer to a category center (the prototypical value of a vowel in the language), a listener is less likely to be able to differentiate between them. This situation is referred to as the *perceptual magnet effect*. In the 3-formant approximation the magnet effect makes the following testable implication. Let a and b be a pair of points a fixed distance l from each other in formant space, and let c and d be another such pair, also separated by a distance l. Supposing that both members a, b of the first pair occur closer to a category center than both members c, d of the second, then a listener will be less likely to distinguish a from b and more likely to distinguish c from d. It is claimed in the experimental literature that this effect is observable by the age of 6 months (Kuhl, 1991; Sussman, 1995).

2. PROBLEM

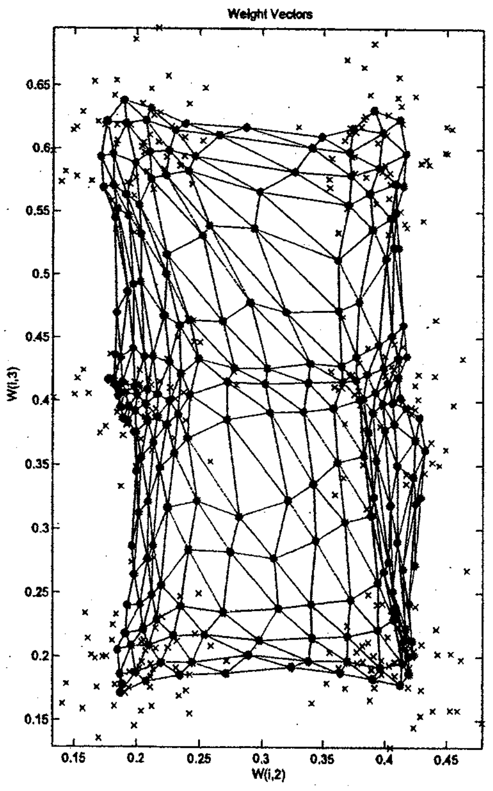
The use of *Kohonen* or *self-organizing feature maps* to obtain a graphical representation of the categories of sounds (phonemes) of a language can be traced at least as far back as (Kohonen, 1983). In the present case, the problem is to determine whether an SOFM can perform the sort of learning required to exhibit the perceptual magnet effect: in terms of the neural network, this means fewer weights nearer the centers of the categories, and more near the periphery. This is the opposite of the typical behavior of an SOFM. To develop this learning alorithm, the initial experiments were done on a simplified input space. Rather than actual data in the form of vowel formant clusters, experiments were run on a set of six normally distributed clusters in 3-space centered at (0,0,0), (0,1,0), (1,0,0), (1,1,0), (0,0,1) and (0,1,1) (figure 2(a)). The six points, treated as means, replace prototypical vowel values, and the normal distributions replace the distribution of vowel qualities within a region, as discussed in section 1. For the validity of this approximation, see Lacerda, 1995.

There are two other considerations in which the empirical facts about learning vowels are factors, in addition to the perceptual magnet effect. The first is in the selection of a feature map, as opposed to some other neural network. There are various network types that can learn patterns and with some noise in the input obtain the nearest matching patterns; with the learned patterns corresponding to prototypical vowels and the varied input corresponding to nearby vowel qualities, such a network would simulate categorial perception. The choices are, however, limited by the learning method. A network using a supervised learning algorithm is undesirable since there is nothing in an infant's learning the vowel space by 6 months that corresponds to target or correct values, against which outputs are checked for the correction of the network. Second, it was a goal to preserve the general usefulness of the SOFM learning algorithm by modifying it as little as possible in modeling the magnet effect. It would be possible, for instance, to have a typical SOFM learn the input space (fig. 2(b)), and then attach code to invert the density distribution of neurons. Though this would produce a model of the memory that demonstrated the perceptual magnet effect, it wouldn't be "learned" by the network, just directly organized by computation. To the extent that a neural network is supposed to be a general model of learning, something so specific as a routine to invert the density distribution was undesirable. A direct modification to the learning algorithm, developing a new general algorithm, is the goal.

Similar problems have been discussed in the psychological literature on *examplar models* of memory (Hintzman 1988). When these models are given sound samples as inputs, each sample or "examplar" is stored in a multidimensional acoustic space and phonemes are taken to be statistical modes in certain areas of this space. Category effects do emerge from the clustering of examplars, and certain effects vaguely similar to the perceptual magnet effect have been interpreted (Goldinger, 1998; Pierrehumbert, 2002). The magnet effect was not conclusively observed, however. Moreover, these are not learned networks, but "hard-coded" models of memory, directly programmed to exhibit these effects.



(a) 6 normally distributed clusters/categories of 200 points in 3-space used to test the SOFM.



(b) A self-organizing feature map (Kohonen map) using the standard algorithm to learn the six clusters of points shown in 2(a). The red points are neuron weights, the blue points are the original input points, and the blue lines connect adjacent neurons. The axes and coordinates refer to formant space; the space of fig. 2(a) is rotated for a direct view of the f2-f3 plane (the W(i,2)-W(i,3) plane in terms of the weight matrix). Density of neuron weights increases/decreases with the density of the input space. This does not exhibit the perceptual magnet effect, as "discrimination" (neuron density) is greater nearer category centers. (10x10 neurons in a hexagonal lattice.)

Figure 2: Test data and network trained using the Kohonen algorithm.

3. IMPLEMENTATION

An SOFM usually corresponds to a set of neurons organized on a line or in a two-dimensional lattice, whose weights during the training phase come to approximate the layout of data in the input space. Let $w_i(t)$, $i = 1...l$, denote the weight vector of neuron i at time t, and let x_j, $i = j...N$, denote input vectors. The algorithm is a form of competitive unsupervised learning and proceeds as follows

(Table 1: SOFM algorithm):

1. At time t, select an input vector x at random.
2. Determine the argument $i_x(t)$ of the weight vector that best matches x:
$$i_x(t) = arg \min_j \|x - w_j\| \tag{1}$$
3. Update weights according to their proximity to $i_x(t)$:
$$w_j(t+1) = w_i(t) + \eta(t) h_{j,i_x(t)}(t)(x - w_j(t)) \tag{2}$$
where the *neighborhood function* $h_{j,i_x(t)}(t)$ is monotonically decreasing in the distance $\|j - i_x(t)\|$ (e.g., the inverse exponentional function centered at $i_x(t)$), and $\eta(t)$ is a *learning-rate parameter*
4. Iterate ORDERING_PHASE times + TUNING_PHASE times.

The local effect is *Hebbian.* A winning neuron at time t (determined in step 2) has its weights brought nearer to the the input $x(t)$ (step 3), and is accordingly more likely to be the best match in future iterations of inputs that are near $x(t)$. Further, in step 3, neurons that are near the winning neuron in the net configuration (be it line or lattice) also have their weights updated to be nearer to the input neuron $x(t)$, according to their proximity to the winning neuron (the monotonicity of h in step 3). As indicated, the neighborhood function and the learning rate parameter vary with time. During the first ORDERING_PHASE iterations, the neighborhood function is usually modified to decrease steadily so that it finally has a value of 1. Specifically, $h(t)$ is the integer nearest $h_0 * (t/ORDERING\_PHASE)$ during the ordering phase, where h_0 is the initial size of the neighborhood function (usually just large enough to cover the entire network) and t is the current iteration. Therefore, only the neurons immediately adjacent to the winning neuron are updated at all at step 3 during the subsequent tuning phase.

When studied analytically the input vectors are modeled by a continuous space X with a probability distribution f_X. The result proposed by Kohonen is,

> The point density function of the weight vectors tends to approximate the probability density function f_X of the input vectors, and the weight vectors tend to be ordered according to their mutual similarity.

The second condition asserts that the layout of the input space is reflected in the layout of vectors in the feature map (he phrases this in terms of "mutual similarity" because, if there is a reduction of dimension in going to the output space, the actual physical layout of the weights may look different from the input space, but still vectors that are nearer each other in the input space will be represented by weights in the output space that are nearer to each other, and conversely). The first condition asserts that, within this layout, a denser area of the input space will correspond to a denser area of the output space, and conversely. Figure 2(b) demonstrates these properties. In particular, the weights are denser according to their proximity to a category center. Earlier work on SOFM's supposed that the point density of the output space (of weights) is directly proportional to f_X (Kohonen, 1982); later analyses have suggested a distortion, that the point density is actually proportional to $f_X^{1/3}$ or $f_X^{2/3}$. In modeling the perceptual magnet effect, however, the point density must actually decrease as the input probability distribution increases. [1]

Initial attempts to invert the probability distribution involved manipulation of the neighborhood function, but were unsuccessful. For instance, making $h_{j,i_x(t)}(t)$ monotone increasing in the distance $\|j - i_x(t)\|$, so that neurons nearer to the winning neuron were updated less than those farther away, did not have the intended effect but only delayed convergence. Next, to determine which neurons are firing most, a running count is maintained for each neuron of how many iterations it won. A routine is then added to the update mechanism, which looked at the "winningest" neurons at each iteration of the tuning phase, and removed a neighborhood of the nearest (in the neuron configuration) neurons from around them. This, however, only had the effect (Fig. 3) of equalizing the density throughout the weight space, as one might expect, since clustering occurs when neighboring neurons develop similarly.

The next and final attempt also involved a running count of each neuron's wins as an approximation to the density of the input space. After updating, a fixed number of neurons around a fixed number of the most active neurons were removed. This, presumably, is close to the intuition behind the perceptual magnet effect. If a given vowel point in the formant space has been heard very frequently, when a vowel different but near to it is encountered it is identified with the former vowel. The observed consequence that discrimination is lower near the frequently heard vowel. The standard SOFM algorithm given in Table 1 may be modified to incorporate the following step 3.5 between steps 3 and 4. Let l be the number of neurons, and let $wins$ be a vector of length l with $wins(i)$ being the number of times neuron i has been selected:

3.5.

1. Update the counter of $wins$:

$$wins(i_x(t)) = wins(i_x(t)) + 1 \qquad (3)$$

[1](Lacerda, 1995) has in fact suggested an inverse proportion on the square or cube of the f_X, though I couldn't find any justification for these values.

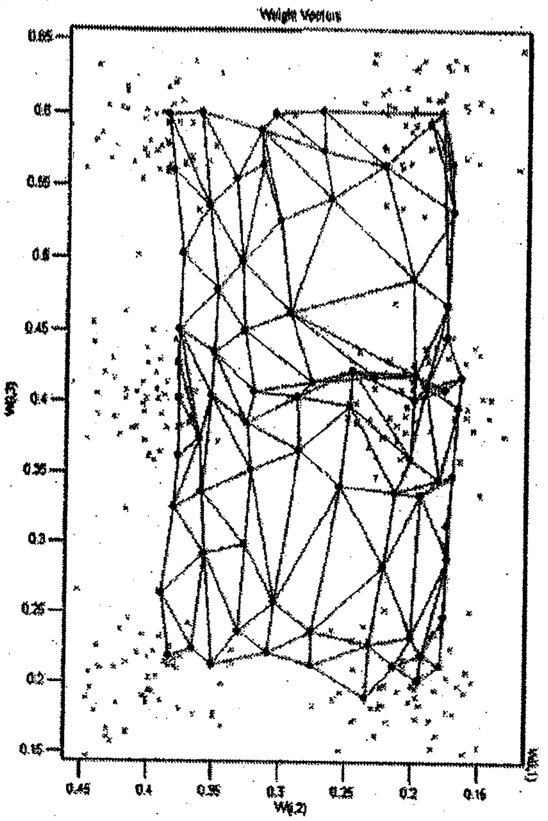


Figure 3: The first attempt to modify the SOFM algorithm so that weight (red points) density decreases with input point (blue points) density. Comparing this with Figure 3, the dense regions at the centers of the six clusters are no longer present. The density distribution isn't reversed however; it is just somewhat more uniform over the formant space. (Initial parameters: 10x10 neurons in a hexagonal lattice; using this algorithm there are typically fewer neurons in the final network.)

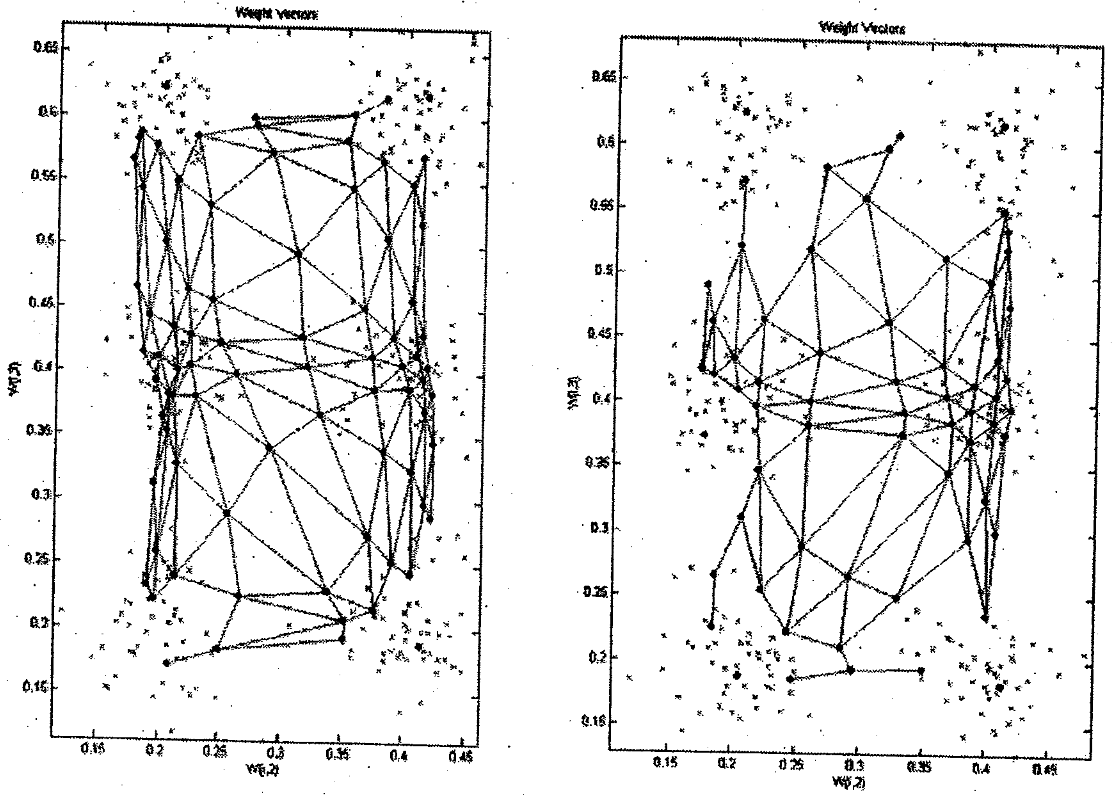
2. Setting k to each of the greatest N_WINNERS(t) values in the *wins* vector, remove the N_REMOVE(t) neurons whose weights are nearest to $w_k(t)$, i.e., remove

$$j_0 = arg\ \min_j \|w_j(t) - w_k(t)\| \qquad j = 1...l$$
$$j_1 = arg\ \min_j \|w_j(t) - w_k(t)\| \qquad j = \{1...l\} - \{j_0\}$$
$$...$$
$$j_{N\_WINNERS} = arg\ \min_j \|w_j(t) - w_k(t)\| \quad j = \{1...l\} - \{j_0...j_{N\_WINNERS-1}\}$$

4. RESULTS

4.1 Using Test Clusters

Demonstrations are presented of feature maps trained on the developed algorithm in figures 4(a)-5. As described in Section 3, two parameters are introduced into the system by this algorithm, N_WINNER being the number of the most active neurons to treat as "magnets" and N_REMOVE being the strength of these magnets, i.e., the number of the neurons nearest to the magnets to remove. Appropriate values for these parameters were determined empirically. Figures 4(a)-4(b) display results for the two values of N_WINNERS, 3 and 6. At the category centers (indicated in the figure by the six greatest clusters of "x" marks), only one or a few neurons' weights (indicated by filled circular dots) remain and to this extent the input space distribution has been reversed. On the other hand, the size of the network (10x10 neurons) seems too small for the value of N_REMOVE, so that the distribution of weights is too sparse to really see the trends in the distrbution. In figure 5 the number of neurons in the system has been increased from 100 to 225, giving a more demonstrative picture. The effects are particularly good at the middle two clusters, at each of which there is just one neuron with weights at the center, while toward the density of neuron weights can be seen to increase in the areas between the adjacent categories; consequently, any sounds in the region vacant region toward these category centers will all be identified with the single central neuron, while sounds in the area between categories will be more likely to be identified with different neurons (discussed further below).



(a) 10x10 neurons in a hexagonal lattice, N_WINNERS=3,N_REMOVE=1.

(b) 10x10 neurons in a hexagonal lattice, N_WINNERS=6, N_REMOVE=1.

Figure 4: Using the modified Kohonen algorithm on test data

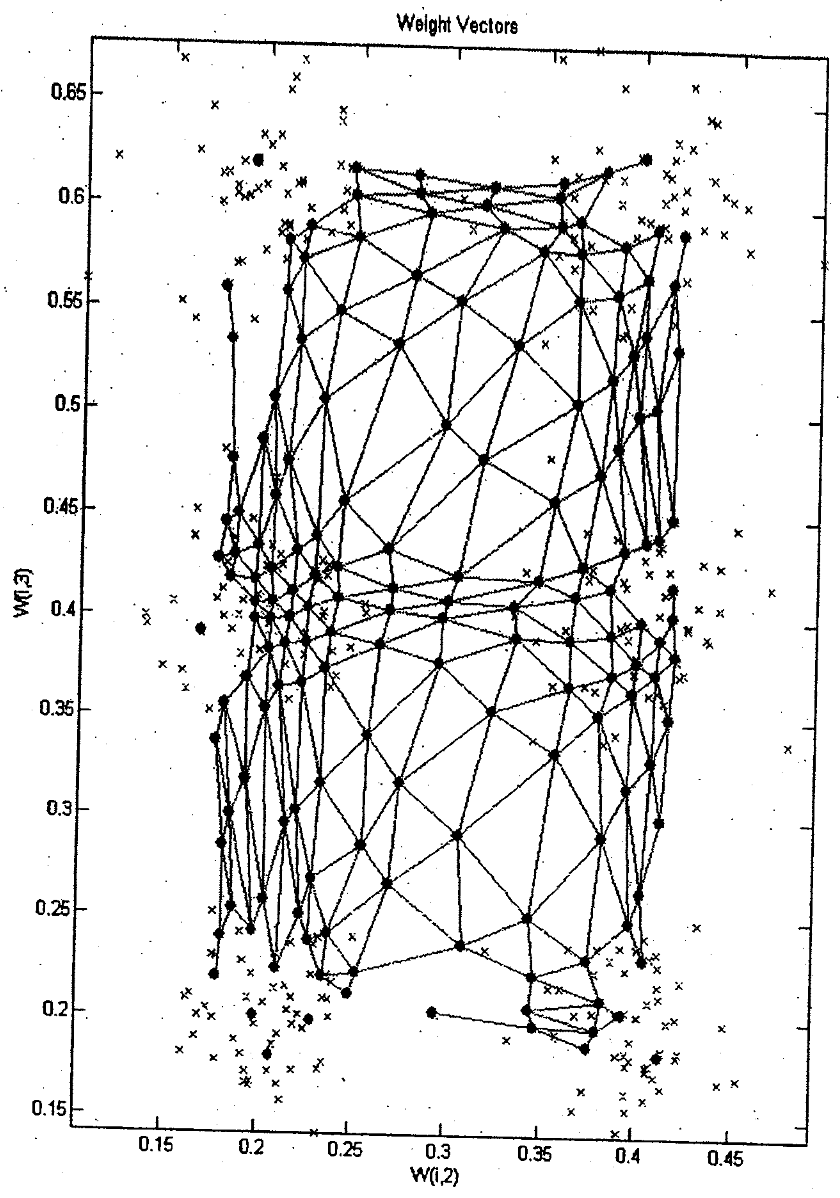


Figure 5: Using a network with more neurons than figures 4(a),4(b). (15x15 neurons in a hexagonal lattice, N_WINNERS=6, N_REMOVE=1.)

4.2 Using English Vowel Clusters

The same network is trained on data drawn from normal distributions around mean formant values of the five English vowels /iy/,/ey/,/ae/,/ow/,/uw/, shown in figure 6(a). The results (figures 6(b), rotated and magnified) were similar to those in Fig. 4. One way to graphically interpret the perceptual magnet effect is given in figure 7. A vowel of quality /uw/ is selected at the point labeled (816.9,1711,2496)in 7(a). Since, according to the magnet effect, pairs of vowels a fixed distance apart ought to be more discriminable according as their formants are farther from a category center, vowels on a sphere centered at the labelled point ought to be discriminated by the network when they occur on the farther side of the category center. In other words, with the points on a sphere equidistant from the center, the pairs consisting of the center with a point on the surface can be taken as the pairs in the test of the magnet effect described in section 1: a pair consisting of the center with a point on the side of the region of the sphere closest to the category center should be less likely to be discriminated (i.e., the network should return the same output for both of them) than a pair consisting of the center with a point on the region of the sphere farthest from the category center. This in fact occurred, as shown in 7(b). A script is written to simulate the network on 225 points on a sphere centered at the labeled /uw/-like vowel, and to plot the points that the network differentiated (i.e., returned different weights for) from the vowel at the center. The incomplete portion of the sphere corresponds to points that, in accordance with the magnet effect, were not differentiated, and as shown they all lie on the side of the sphere toward the category center.

ACKNOWLEDGEMENT

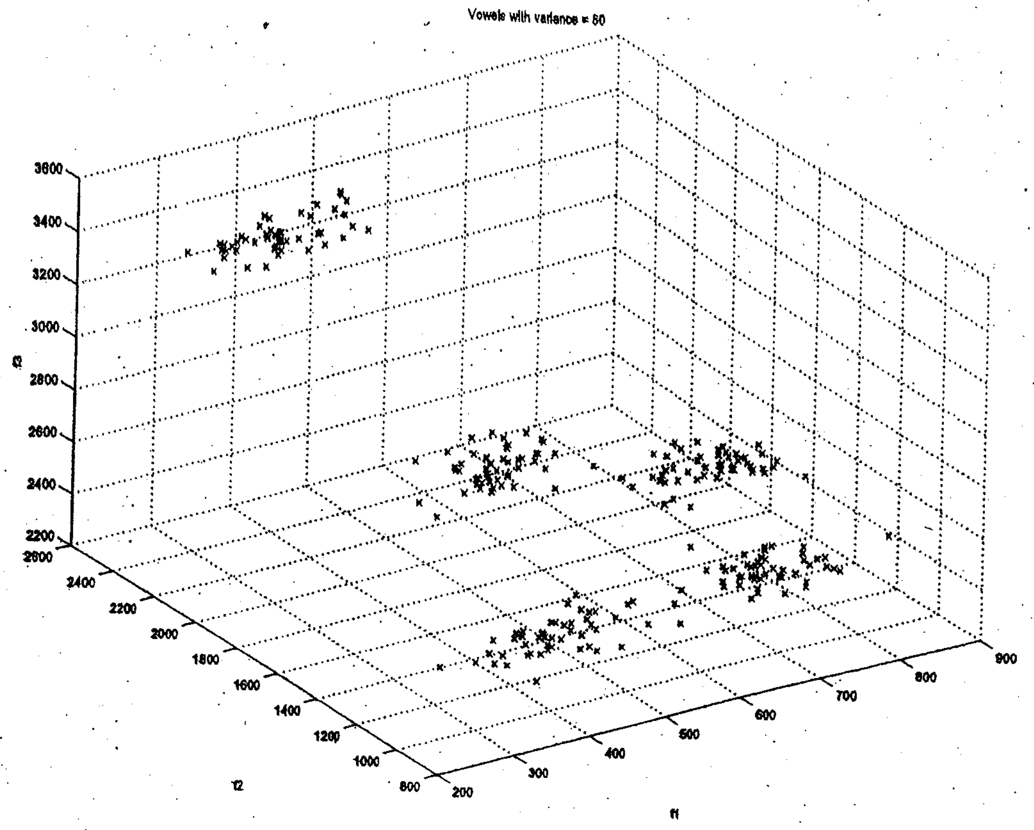
The author would like thank Prof. Willard Miranker for guiding this paper through numerous revisions.

REFERENCES

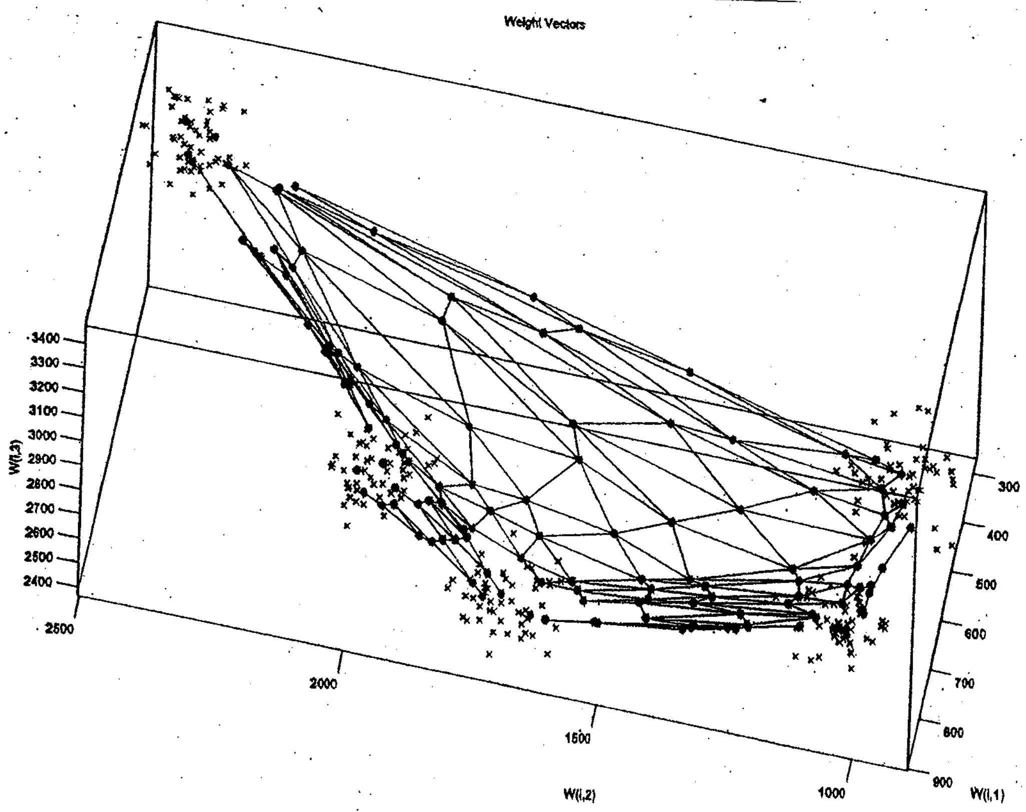
1. Borden, G, K. Harris and L. Raphael (2002). Speech Science Primer. Baltimore: Williams and Wilkins.
2. Estes, W. (1994). Classification and Cognition, Oxford University Press, New York.
3. Goldinger, S. (1998). An Episodic Theory of Lexical Access. Psychological Review 1998, Vol. 105, No. 2.
4. Haykin, S. (1999). Neural Networks: A Comprehensive Foundation. Prentice Hall.
5. Hintzman, D. (1988). Judgments of Frequency and Recognition Memory in a Multiple-Trace Model. Psychological Review 1988, Vol 95, No. 4.
6. Kohonen, T. (1983). Self-Organization and Associative Memory. Springer-Verlag.
7. Kuhl, P.K. (1991). Human adults and human infants show a 'perceptual magnet effect' for the prototypes of speech categories, monkeys do not. Perception & Psychophyics, 50, 93-107.
8. Lacerda, F (1995). The Perceptual-Magnet Effect: An emergent consequence of exemplar-based phonetic memory. In Elenius, K. and P. Branderyd, eds., Proceedings of the XIIIth Int'l Conference on Phonetic Science. 140-147.
9. Pierrehumbert, J. (2002). Word-specific phonetics. LabPhon Proceedings. Cambridge University Press. 102-139.

10. Stevens, K. (1972). "The quantal nature of speech: Evidence from articulatory-acoustic data". in Denes, P. and David Jr, E. (Eds.), Human communication: A unified view. McGraw-Hill, New York, 51-66.

11. Sussman, J. and Lauckner-Morano, V. (1995). "Further tests of the perceptual magnet effect in the perception of [i]: Identification and change/no-change discrimination". JASA 97, 539-552.

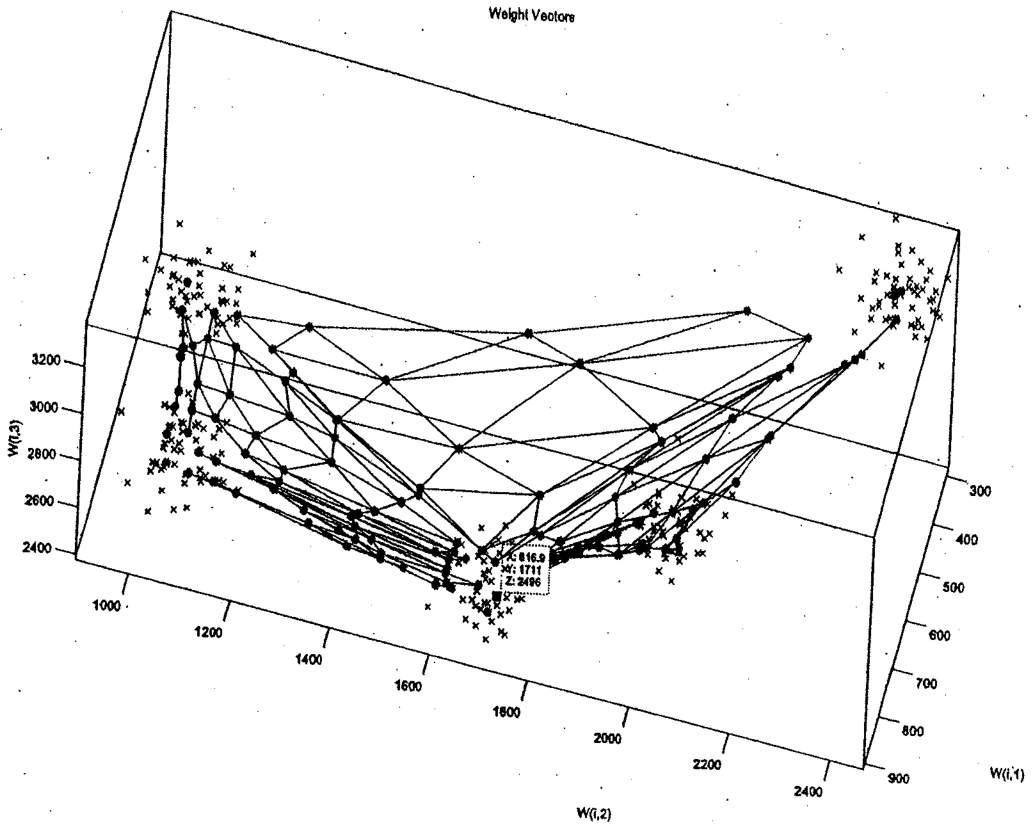


(a) Clusters around mean formant values for English /iy/,/ey/,/ae/,/ow/,/uw/.

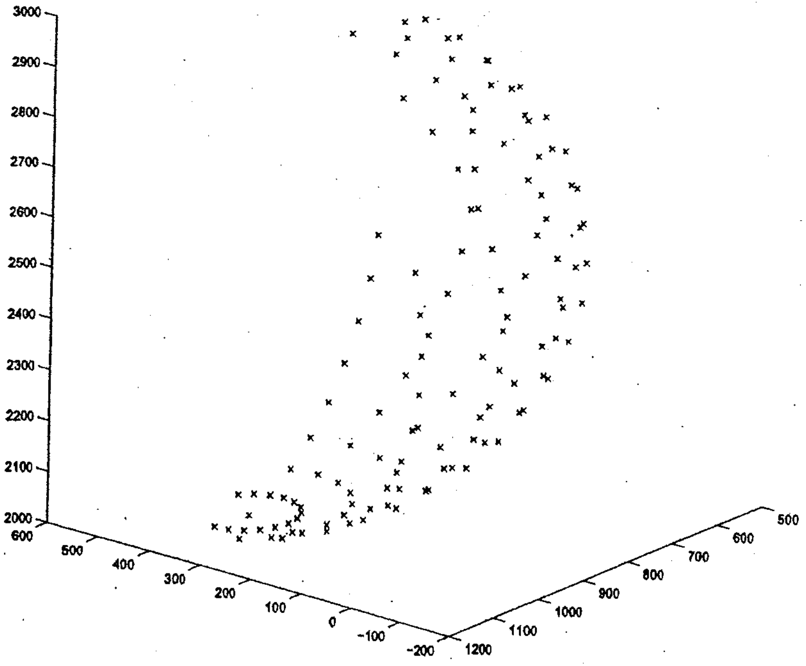


(b) 15x15 neurons in a hexagonal lattice, N_WINNERS=6, N_REMOVE=1; rotated view

Figure 6: Using the modified Kohonen algorithm on clusters centered at the five mean English vowel centers



(a) A sample vowel of quality /ųw/; the network is simulated on this vowel, and then on 400 vowels on a sphere of radius 200 centered here.



(b) Sphere showing which vowels were distinguished from the vowel in figure 7(b). The incomplete portion of the sphere corresponds to vowel that were not distinguished, as a consequence of the magnet effect.

Figure 7: A graphical demonstration of the magnet effect, as exhibited by the developed network.

Biasing Techniques in Genetic (Evolutionary) Algorithms

Semih Salihoglu
Yale University, CS Department
New Haven, CT

Abstract
A Genetic Algorithm (GA) is a search technique used to approximate solutions for optimization and search problems. A GA maintains a population of individuals, which are abstract representations of candidate solutions that evolve towards better solutions. The conventional way of creating the initial population of a GA is to randomly select a set of points from the solution space. If we know that a set of the dimensions of the solution space are more important than another by some measure, we would prefer our initial population to span as wide a range of these dimensions as possible. We call such a population a "biased population", in that certain genes of the individuals are favored. We want to study the effects of three biasing methods (found in Neural Networks literature) on the evolution of a GA: a) principal components analysis, b) sensitivity measure (Shin, Yun, Kim, & Park, 2000), and c) saliency measure (Shin, Yun, Kim, & Park, 2000). Although biased populations start off with fitter individuals, both biased and unbiased populations yield similar solutions.

1. INTRODUCTION

1.1. Genetic Algorithms

Genetic Algorithms (GA) are a branch of evolutionary computing, a rapidly growing area in artificial intelligence. GAs are inspired by Darwin's theory of biological evolution; by the idea of the "survival of the fittest", in particular. They were initially introduced by (Rechenberg, 1965) but studies in the area gained momentum only a decade later through the extensive contributions of (John Holland, 1975).

A GA is a search technique used to approximate solutions for optimization and search problems. GAs usually maintain a population of individuals, which are abstract representations of candidate solutions. These solutions evolve to generate superior solutions. GAs are intended to exploit the following idea stated by John Holland in his introductory book (Holland, 1975): "Computer programs that 'evolve' in ways that resemble natural selection can solve complex problems even their creators do not understand." Unlike deterministic methods of problem solving, such as brute force enumeration of the solution space, GAs are randomized. While they may rapidly converge, the limit may be suboptimal. (Holland, 1975) provides a more detailed discussion of GAs for the interested reader.

There are four main components of a GA: population, fitness function, reproduction method and selection method. The population consists of individuals, which are represented by virtual chromosomes. A chromosome is a pointer to a solution to the problem. Each individual has an associated value calculated by the fitness function. This value represents the fitness of the individual. Individuals that are better solutions to the problem should get better fitness values. The population goes through a reproduction phase in which a group of individuals are matched in a predefined way to produce new individuals. Finally, weak (less fit) individuals from the population are replaced by fitter offspring. The population evolves in this way until a set of stopping criteria are met, the most popular of which are bounds on the number of reproductive phases the population goes through.

```
initialize population;
while stopping criteria not met
  reproduce new generation
  mutate individuals with certain probability
  calculate fitness of individuals.
  partially replace population by
  offspring
end
```

Fig 1. A generic genetic algorithm.

In Fig. 1. a generic genetic algorithm is shown in pseudocode. The algorithm is executed with the expectation that the new population will contain fitter individuals than the old one. The fittest individual from the final generation represents the solution found by the algorithm.

Genetic Algorithms have been used to solve many different types of business problems in functional areas such as finance, marketing information systems and production operations. Within these functional areas GAs have been used in a variety of applications, such as tactical asset allocation, job scheduling, machine-part grouping, and computer network design. An example of a real, practical application of a GA to aircraft design is found in (Minga, 1986).

1.2. Using Neural Networks to Bias the Initial Population

The conventional way of creating the initial population of a GA is to randomly select a set of points from the solution space (the set of possible solutions). If we know that a subset of the dimensions of the solution space are more important to the solution by some measure, then we prefer our initial population to span as wide a range of these dimensions as possible. For example, suppose we have a 3 dimensional solution space (x, y, z). If we know that x is the most important dimension according to some measure, then we would prefer our population to contain individuals that have as many different x values as possible. In the extreme case when x is the only determinant of the solution, we would want x to vary as much as possible to span a wider range of solutions. Our hypothesis is that a GA initiated with a favorably biased population converges faster (in fewer reproductive steps) to better solutions than a conventional GA.

This approach introduces the question of how to determine the important dimensions of the solution space. In this paper, we introduce ideas from neural networks to measure the importance of dimensions. We determine the importance of each dimension by PCA (principal components analysis), and two network pruning methods introduced by (Shin, Yun, Kim, & Park, 2000). We bias the initial population according to the most important dimensions and compare results obtained between biased and unbiased populations.

2. MODEL

2.1. Optimization Problem: Multidimensional Knapsack Problem (MKP)

We will test our hypothesis on the multidimensional knapsack problem (MKP). We will solve the MKP through implementation of a GA, and compare the results we get from biased populations and unbiased populations. (Chu, & Beasley, 1998), and (Gottlieb, 2000) are some examples of research on GAs used on MKP.

The NP-hard 0–1 multidimensional knapsack problem is a generalization of the 0-1 simple knapsack problem. There are a set of n items, and m knapsacks. Each item has an associated value v_j. Each knapsack has a capacity and has an associated cost for each item. The problem is to select a set of these n items, such that none of the knapsack capacities is exceeded and the total value from the items is maximized. MKP can formally be stated as follows:

$$\begin{array}{ll} \text{Maximize } \sum_j \mathbf{v_j x_j} & \\ \text{subject to } \sum_j \mathbf{c_{i,j} x_j} \leq \mathbf{C_i} & \text{for } i = 1,2,\ldots,m \\ \text{where } \mathbf{x_j} \in \{\mathbf{0}, \mathbf{1}\} & \text{for } j = 1,2,\ldots,n \end{array}$$

Fig. 2. Definition of MKP

n is the number of items. **m** is the number of knapsacks. $\mathbf{x_j}$ is a binary variable that indicates whether or not item **j** has been selected. $\mathbf{v_j}$ represents the value from item **j**. $\mathbf{C_i}$ represents the ith knapsack's cost capacity, and $\mathbf{c_{i,j}}$ represents the cost incurred by knapsack **i** when item **j** is selected.

2.2. Specifics of GA Implementation

2.2.1. Individual Representation

We represent an individual as a string of n bits, each of which is a 0 or a 1. 1 specifies a selected item, 0 specifies an unselected item. So an individual is an encoding of the selected items. A population consists of N individuals. N was set to 128, 64, 32 or 16 for each problem instance. We tried various different sizes for n: 20, 40, 60 and 100.

2.2.2. Fitness Function

The definition of the fitness function is given in Fig. 3.

$f(x) = (\sum v_j x_j) - (\max(\text{violation})\max(\text{value}))/\min(\text{cost})$, $j = 1,\ldots,n$ where

$\sum v_j x_j$ represents the total value of selecting the items represented by ones in x.

$\max(\text{violation}) = \max(cv_i)$, $i = 1,\ldots,m$ is the maximum constraint violation defined as;

$cv_i = \max(0, C_i - \sum c_{i,j} x_j)$, $j = 1,\ldots, n$

$\max(\text{value}) = \max(v_j)$, $j = 1,\ldots, n$ is the value of the most valuable item

$\min(\text{cost}) = \min(c_{i,j})$, $i = 1,\ldots,m$, $j = 1,\ldots,n$ is the minimum cost of any item in any knapsack.

Fig. 3. Definition of the fitness function.

While this fitness function is not guaranteed to yield negative results for an individual violating at least one constraint of a knapsack, it almost always does. Our implementation outputs the fittest individual that does not violate any constraints as its solution.

2.2.3. Selection and Reproduction Methods

The parents to be mated are selected by the conventional tournament selection. In tournament selection two pairs of individuals are randomly picked. The fittest of each pair are then selected to form the parents to be mated.

We used the crossover reproduction method in mating selected parents. The method generates a new individual by copying each gene of the individual from parents a and b, with 0.5 probability each. After the crossover is terminated, the child goes through a mutation process, where each gene of the child is flipped with a mutation probability of 0.01.

2.2.4. Replacement Method

For MKP problems, the common replacement method used is what is known as the "steady-state" replacement algorithm. For each generation, only one new child is reproduced. This child is compared against the least fit individual of the population and the fitter of the two remains in the population. As a result, at each reproductive step, a single individual may be replaced at most.

The algorithm runs for 50,000 steps and at each step, the fittest individual that does not violate any constraints is outputted. This is repeated for 50 times and the results of each run are averaged.

3. PRUNING TECHNIQUES

3.1. PCA Based Dimension Weighting Method

One of the three methods we used to measure the importance of a dimension is based on principal components analysis. We first generate a random 2000 individuals for the given problem. An individual consists of n + 1 dimensions: n inputs and the fitness. The input dimensions are random 0s or 1s, which collectively represent the chromosome of the individual. The fitness dimension is the fitness of the chromosome, as specified by the n input dimensions. We then do a PCA on this data set of size 2000, and take the resulting principal component and project it on each of the dimensions 1 ,..., n. We then take the ranking of the absolute values of these projections as the measure of dimension importance. At times, the principal component was a vector of 0s followed by a 1 or -1. The last dimension is the n+1st dimension (the fitness dimension). In such cases, we took the second principal component in our method. Since we are only interested in the relative rankings of the input dimensions, we normalized the fitnesses to [0,1] before doing the principal component analysis. The result for this analysis for the 20-10 MKP (20 items and 10 knapsacks) was 21 13 3 10 16 7 20 17 18 12 1 9 15 11 5 6 8 4 2 19 14. The last dimension is the fitness dimension and obviously is the most significant one.

3.2. Sensitivity Measure

For the "Sensitivity" and the "Saliency" measures, we train a feed-forward net by backpropogation. The net has n input nodes, 1 hidden layer consisting of M neurons and a single output neuron. We set M to 10 or 30 depending on the size of the problem. The training set consists of bit strings of size n, representing a random individual in the population, and the fitnesses corresponding to these n-bit chromosomes.

The "Sensitivity" measure was introduced by (Shin, Yun, Kim, & Park, 2000). It measures how much the output changes with respect to a change in an input. It is calculated by removing the input node from the trained network, and then taking the difference in the network's outputted value between the time the input node is removed and the time it is left in place. The node is removed by setting all the connected weights to 0. The sensitivity S_i of an input node x_i is given by:

$$S_i = \frac{\left(\sum_L \frac{|P^0 - P^i|}{P^0}\right)}{n}$$

Fig. 4. Formal definition of "Sensitivity"

P^0 is the value the trained network outputs when node i is in place for a training instance. P^i is the corresponding value the network outputs when the input node i is removed. L is the set of training data and n is the number of training data. We simply set the input node to 0 to calculate P^i. The results for the 20-10 MKP were 3 7 16 10 17 18 15 13 1 12 9 20 11 8 5 6 4 19 2 14.

3.3. Saliency Measure

This is the second measure proposed in [5]. The saliency of a weight is measured by estimating the second derivative of the error with respect to the weight. The saliency of an input node is proportional value of the weight squared. It is formally specified as follows:

$$\text{Saliency}_i = \sum^{M}_{j=1}((w_{ji})^2\cdot(w_j)^2)$$

where w_{ji} denotes a connecting weight from x_i to to z_j, where z_j is the jth node in the hidden layer. w_j is the weight from the jth hidden neuron to the output neuron. Proceeding analogously from section 3.2., the results for the 20-10 MKP was 16 7 3 10 17 15 18 13 9 20 8 1 12 5 11 4 6 19 14 2.

4.IMPLEMENTATION AND RESULTS

We used Java to implement the genetic algorithm specified in Section 2. The source code files are on the Project CD, together with the relevant documentation. The PCA and backpropagation are both implemented in Java as well, but because of its robustness and simplicity, we preferred Matlab to do the actual computation.

We did the tests on multiple sample problems, (m – n) = 20-10, 40-5, 60-30, 100-10, etc. While we realize that this data set is insufficient to conclude for or against any hypothesis, this paper is intended as an introductory engagement of the idea, rather than a conclusive one. The graphs of the inputs can be found in Appendix A. The data sets are on the Project CD.

When biasing a population, $\log_2$(population size) was used as the number of biased dimensions. In order to bias the initial population, we would set the biased bits of the ith individual to the binary representation of (i-1). For example, suppose that the PC ranked 1, 2, 3, 4, and 5 as the five most important dimensions. Then, when testing the problem for a population size equal to $2^5 = 32$ individuals, we would set the ith individual's first five bits to the binary representation of i-1. As a result, the first individual would start with 00000, the second with 00001, the third with 00010, etc. This assures that all of the combinations of these five dimensions are included in the population. The rest of the dimensions would be set randomly.

The tests were run on all problems for initial populations of sizes: 16, 32, 64 and 128. Regular PCA was run over the 20-10 MKP and 40-5 MKP and the principal component was taken to extract the importance of the ranks. The fitness dimension was ignored, since it is not part of the input dimensions. For other MKP instances, we used the second principal component for our analysis, for the reasons cited earlier. The backpropagation algorithm used in training the feed-forward net was run for 10,000 epochs with a training function that updates weight and bias values according to the Levenberg-Marquardt optimization, instead of using gradient descent. This decision was taken upon experimenting on learning speed, rather than using theoretical reasoning. The "Sensitivity" measure was calculated by setting the connecting weight of input node i to 0 rather than setting the input to 0.

Aside from the 20-10 problem instance with 128 individuals, all biased populations (PCA, activity, and saliency), start off with better solutions than an unbiased population. Over the first 50 runs, the biased populations consistently produced better solutions. Over 50,000 runs, they all catch up and no single one dominates the other. Furthermore, it was not the case that the biased populations converged to an optimum faster. There was no obvious trend in how populations converged to a solution. The only consistent trend was that the biased populations started off with better initial fitness than the unbiased ones.

One interesting fact about smaller problem instances was that PCA, activity and saliency measures yielded very similar rankings for the importance of dimensions. In particular, for the test when the population size was 128 for the 20-10 MKP instance, 6 dimensions out of the 7 biased dimensions were overlapping. The activity and saliency yielded input node 15 as one of the 7 dimensions, whereas PCA yielded input node 20 as one of the 7 dimensions. This was the only case where the unbiased population started off with individuals that were fitter. Yet as with the rest of the cases, they converged to almost the same solution with similar convergence times.

5. CONCLUSIONS AND FUTURE WORK

This was an introductory work on biasing techniques in genetic algorithms rather than a conclusive research on the area. Although the results did not confirm our hypothesis, we can say that our method positively biases the population in a consistent manner. That is, the biased populations contain better initial individuals than the unbiased one, as expected.

At this stage, however, there are many open questions. This approach has to be tested on multiple problems and on different selection and replacement methods, with different pruning techniques. It might be the case that biasing does not work well with a steady-state replacement algorithm or a tournament selection method.

Furthermore, our input weighting methods are primitive in nature. It is likely that we are not getting the actual information we want to extract from our analysis. For PCA, one problem is that all the dimensions aside from the fitness dimension are made of randomly selected 0s and 1s. It is therefore very likely that the information extracted from the PCA is misleading. The fact that the projection of the principal component on a dimension is higher than another may not imply a greater importance. There may exist a more sophisticated analysis than mere vector projection to extract the information we want.

Also for the 100-10 and 60-30MKP instances, the backpropagation algorithm failed to return consistent rankings. This may be a consequence of poor learning of the network. Future work has to test the rankings on different network structures, epoch numbers, and learning rates. We may have failed to teach the net the training set sufficiently. This was definitely not the case for the 20-10 and 40-5 MKP instances, since the rankings were run on 10 different training sets of size 2,000 and there was extreme consistency in the outputted rankings. In particular for both activity and saliency measures, the first seven and last four rankings were always the same in all 10 trials. The same was true for PCA.

PCA results were more consistent in the 20-10 MKP instance than the 100-10 MKP instance.

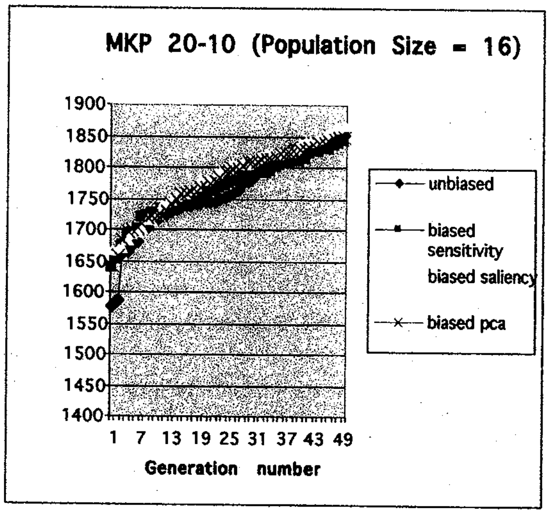
The ultimate goal of the research should be to dynamically set the number of input dimensions that need to be biased. Our approach was primitive in that we biased the log(population size) dimensions irrespective of the actual importance of dimensions. The fact that we can rank the input dimension does not mean that we should bias all log(population size). It may be the case that only a subset of these dimensions is important. We need to be able to grade the inputs in addition to ranking them. Ultimately we would want to have a dynamic way of biasing the actual important dimensions.

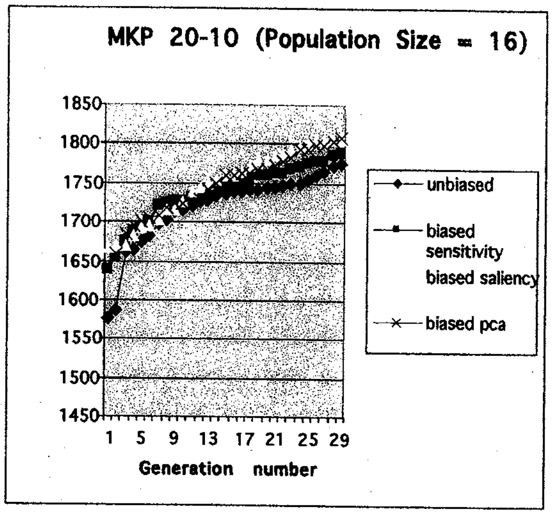
Future work that builds on this introductory research should focus on finding a reliable measure of input weighting, and a better understanding of the problem types where biasing could yield better solutions in a faster way. Although our initial results do not confirm our hypothesis that a biased population would converge faster to better solutions, the fact that we could actually bias the population should encourage further research.

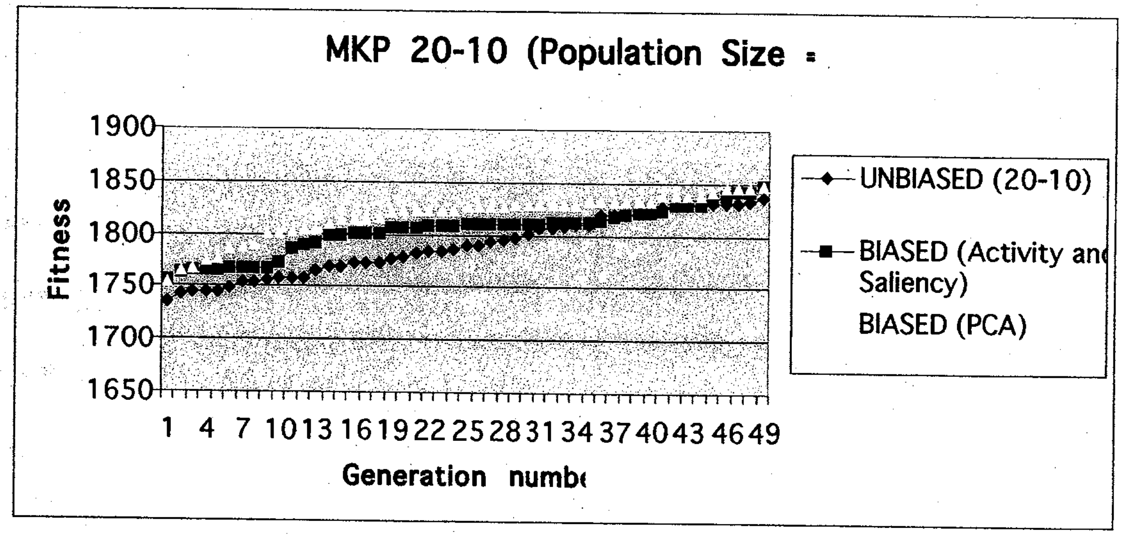
REFERENCES

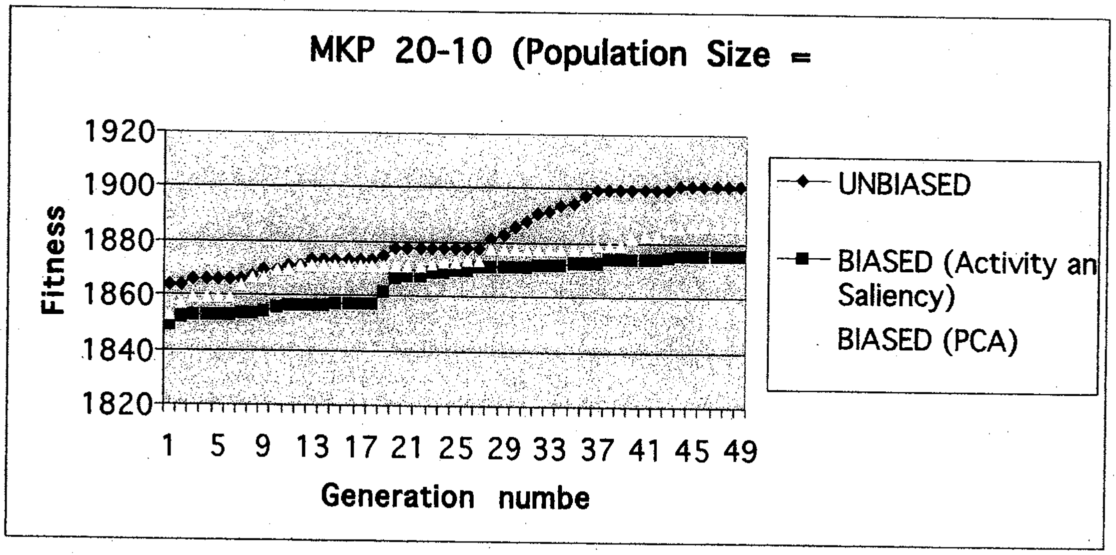
1. Holland, J. (1975). *Adoption in Natural and Artificial System.*
2. Minga, A. K. (1986). *Genetic Algorithms in Aerospace Design.*
3. Chu, P.C., & Beasley J. E., (1998). *A Genetic Algorithm for the Multidimensional Knapsack Problem*
4. Gottlieb, J. (2000).*Permutation-based evolutionary algorithms for multidimensional knapsack problems.*
5. Shin, Yun, Kim, & Park. (2000). *A Hybrid Approach of Neural Network and Memory-Based Learning to Data Mining.*
6. Rechenberg. (1965). *Cybernetic Solution Path of an Experimental Problem*

APPENDIX A.

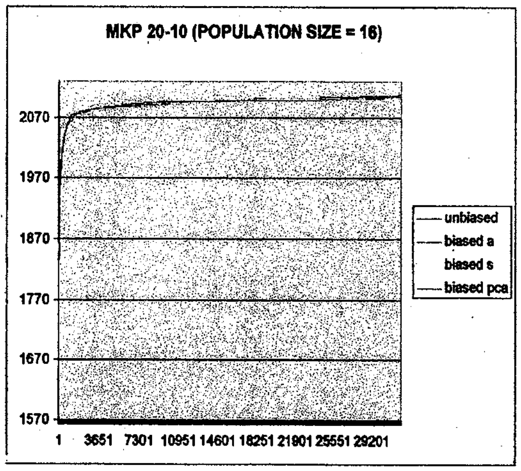

MKP 20-10 (Population Size = 16)
1900
1850
1800
1750
1700
1650
1600
1550
1500
1450
1400
1 7 13 19 25 31 37 43 49
Generation number
unbiased
biased sensitivity
biased saliency
biased pca

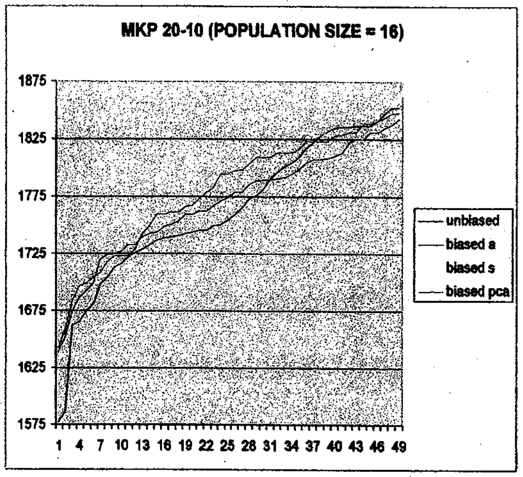

MKP 20-10 (Population Size = 16)
1850
1800
1750
1700
1650
1600
1550
1500
1450
1 5 9 13 17 21 25 29
Generation number
unbiased
biased sensitivity
biased saliency
biased pca

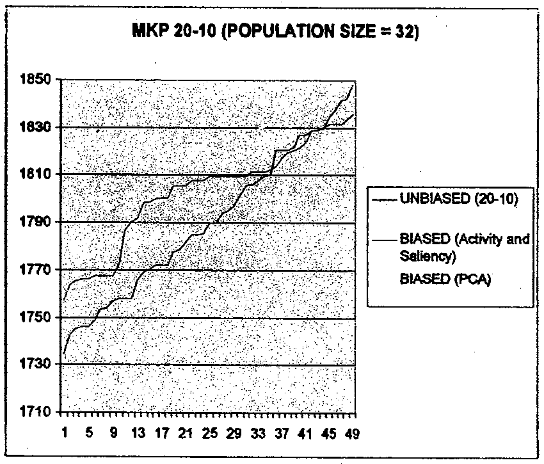

MKP 20-10 (Population Size =
Fitness
1900
1850
1800
1750
1700
1650
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49
Generation numb
UNBIASED (20-10)
BIASED (Activity an
Saliency)
BIASED (PCA)

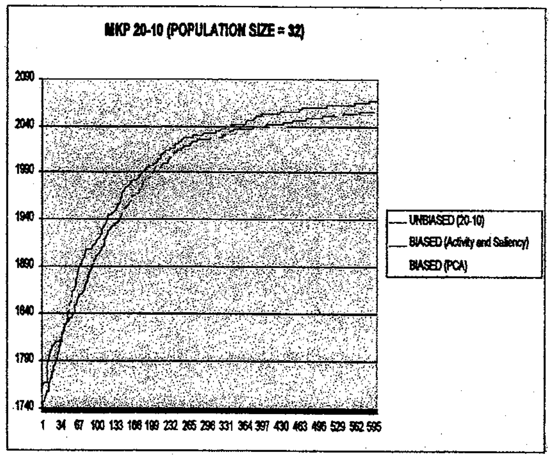

MKP 20-10 (Population Size =
Fitness
1920
1900
1880
1860
1840
1820
1 5 9 13 17 21 25 29 33 37 41 45 49
Generation numbe
UNBIASED
BIASED (Activity an
Saliency)
BIASED (PCA)

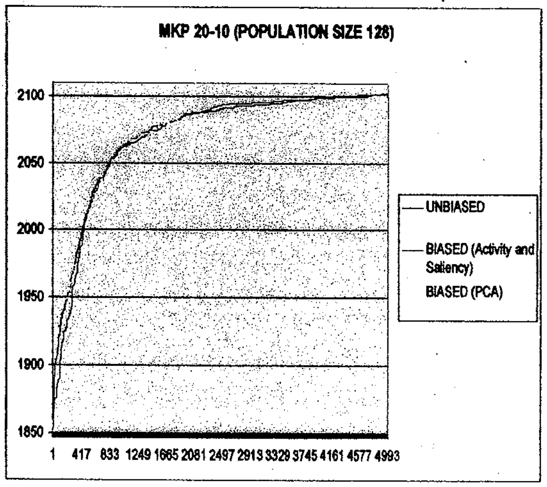
APPENDIX A.

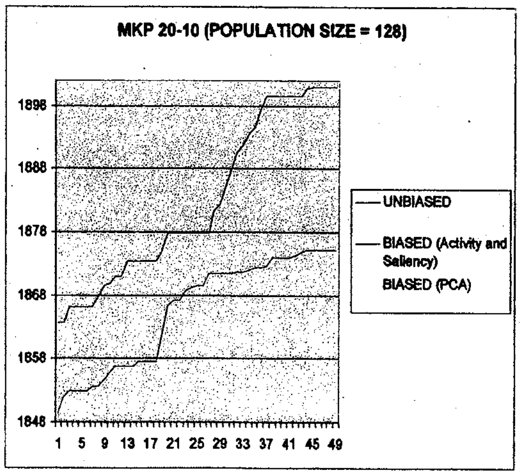

MKP 20-10 (POPULATION SIZE = 16)
2070
1970
1870
1770
1670
1570
1
3651
7301
10951
14601
18251
21901
25551
29201
unbiased
biased a
biased s
biased pca


MKP 20-10 (POPULATION SIZE = 16)
1875
1825
1775
1725
1675
1625
1575
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49
unbiased
biased a
biased s
biased pca


MKP 20-10 (POPULATION SIZE = 32)
1850
1830
1810
1790
1770
1750
1730
1710
1 5 9 13 17 21 25 29 33 37 41 45 49
UNBIASED (20-10)
BIASED (Activity and Saliency)
BIASED (PCA)


MKP 20-10 (POPULATION SIZE = 32)
2090
2040
1990
1940
1890
1840
1790
1740
1 34 67 100 133 166 199 232 265 298 331 364 397 430 463 496 529 562 595
UNBIASED (20-10)
BIASED (Activity and Saliency)
BIASED (PCA)


MKP 20-10 (POPULATION SIZE 128)
2100
2050
2000
1950
1900
1850
1 417 833 1249 1665 2081 2497 2913 3329 3745 4161 4577 4993
UNBIASED
BIASED (Activity and Saliency)
BIASED (PCA)


MKP 20-10 (POPULATION SIZE = 128)
1898
1888
1878
1868
1858
1848
1 5 9 13 17 21 25 29 33 37 41 45 49
UNBIASED
BIASED (Activity and Saliency)
BIASED (PCA)

Exploring the Effects of Dataset Pollution on Neural Network Model Complexity with Simulation

Andrew Smith[i]
Department of Computer Science, Yale University
New Haven, CT 06520

Abstract
This study addresses the problem of class pollution in datasets, i.e. datasets to be used for constructing multilayer perceptron (neural network) classifiers where some of the exemplars have their class labels mislabeled, and how it could be reduced or eliminated to improve prediction accuracy. The main result of the study is that it was empirically determined through simulation that various complexity measures of multilayer perceptrons tend generally to increase with increasing levels of class pollution in a given dataset. If this can be shown to hold very generally, then the complexity measures could be used as the basis of an optimization procedure for de-polluting datasets. The motivation for this work is a structural genomics dataset used for data mining which exhibits the class pollution problem.

1. INTRODUCTION

The National Institutes of Health (NIH) funds several centers in the United States to do *structural genomics* (i.e. high-throughput protein 3D structure determination) and to develop new methods and technology to do it more efficiently[5]. Proteins (which are produced in cells by reading and translating gene DNA sequences) are clustered into families based on evolution, and the main goal of structural genomics is to solve one or a few representative 3D structures of each protein family. Proteins in the same family are similar in sequence and generally have similar structures, and you can model the other members reasonably well if you know one family member's structure and

[i] andrew.smith@alum.mit.edu

sequence alignments of the other members to that one with known structure. It is quite laborious solving protein 3D structures (and hence the interest in developing techniques to make it more efficient), and a lot of it is still an art. Also note that ab initio protein structure prediction (i.e. prediction of protein 3D structure based solely on sequence) is not currently feasible and is one of the great open problems in computational biology --- modeling based on homology (i.e. sequence similarity) to known structures of similar proteins is the best we can currently do reasonably well.

In structural genomics proteins go through a "pipeline" of different experimental steps, with the main ones being cloning, expression, solubility testing, purification, crystallization, and crystal structure determination (this is the main, protein crystallization path; there are other steps followed for protein structures solved using nuclear magnetic resonance ("NMR")). The structural genomics community records status tracking information at the TargetDB site[8] in a simple standard format. One thing we want to do is to use the TargetDB dataset to try to determine the protein sequence features associated with the success or failure of reaching the various pipeline stages of structure determination, i.e. do data mining of TargetDB; we could then hope to predict which proteins are tractable or not for successful experimental structure determination and focus our attention on the likely tractable ones (and thus hope to avoid wasting time and resources on likely dead end proteins). In fact, previous analyses along these lines have been done using random forest and decision tree algorithms trained on the TargetDB dataset[1].

The important problem which is investigated in this study involves the nature of the TargetDB dataset. TargetDB does not contain any real negative information (i.e. knowledge of failure of proteins at the various steps), but is simply a snapshot of current status (i.e. which experimental stage proteins are currently at). Thus, if a protein is at some stage X then that is valid positive information, but you cannot conclude whether the protein failed to reach the next step after X or is simply waiting in the queue to be worked on. The previous published analysis with random forest / decision tree algorithms mostly ignored this problem and, for each experimental stage X (i.e. Cloned, Expressed, Purified, etc.), simply used proteins at stage X as positive training data and proteins at the stage just previous to X as negative data in constructing classifiers and rules to distinguish "Can reach stage X". This is a rough approximation but is somewhat valid in that it is known that there is high attrition at each stage and the negative set will in fact be enriched for true negatives, but it is not ideal. The structural genomics community has recognized the need for negative data, and is starting to collect and report it, but this won't ramp up for a while, and any true negative data that could be obtained now would just be a small fraction of the amount data available in the TargetDB dataset (almost 100,000 proteins).

Finally, while this limitation may be a particular aspect of the TargetDB structural genomics dataset, there might be other situations where it could arise. In general, genomics research is relying more and more on high-throughput experiments (microarrays, yeast2hybrid, etc.) and many of these techniques result in highly noisy datasets or lack of good positive or negative data. Also, high-throughput experiments might not give negative or positive information about specific instances, but only global percentages (e.g. in some high throughput experiment covering 10000 instances imagine it was determined that 8000 were positive but it isn't known which 8000). Thus, solutions to the problem of lack of negative or positive data might be more generally useful.

2. ABSTRACTING THE PROBLEM TO BE INVESTIGATED

The core problem can essentially be conceptualized as follows. Say there are two underlying populations, call them P and N, and you would like to determine what features distinguish P from N and build classifiers to distinguish between them. However, the available dataset consists of a reliable sample from P and another sample (which is the assumed N sample) that consists of a mix from both P and N (but it is known that it is enriched for examples from N and a rough upper bound on the percentage "pollution" by P might be given). In other words, the N sample is corrupted to some extent with samples from P that are mislabeled as N. There is also the symmetric (and equivalent) case of the P sample being corrupted to some extent with samples from N that are mislabeled as P. Finally, both the P and N samples could each be corrupted to some extent with mislabeled members of the other. This study will consider the two symmetric types of pollution, i.e. either the P or N sample is polluted to some extent.

In a previous project (unpublished) using logistic regression as the classifier of focus I investigated through simulations how various measures of prediction performance, such as overall accuracy, sensitivity, and specificity, degraded for classifiers constructed from artificial datasets as more and more random pollution was added to one or both of two classes to be predicted. Key results of this were that, up to about 40% random pollution in one class, prediction performance degraded by no more than about 15% to 20% on average. Interestingly, when both classes were randomly polluted equally the same percentage the logistic regression classifier was very robust, only showing significant degradation in performance starting at around 45% pollution. The likely explanation for this is that the symmetric pollution cancelled each other out leading essentially to the classifier model that would have been created had there been no pollution. An interesting corollary to this is that if it is known that one of 2 classes in a dataset is polluted and the percentage pollution is known, one can achieve close to the optimal, non-polluted classifier model by randomly polluting the other class to the same percentage. Possibly these results for the symmetric pollution case generalize beyond logistic regression to other classifiers such as decision trees or neural networks, although more investigation is needed.

In the project reported here, my goals were first to investigate the problem of dataset pollution with a classifier model more general than logistic regression, namely multilayer perceptrons (neural networks). Second, rather than just empirically determine how performance degrades as pollution is added, I aimed to focus on coming up with solutions for how such polluted datasets could be best handled for optimal accuracy in classification tasks. Specifically I am interested in ways to de-pollute such polluted datasets, in other words how can we determine reasonably well which exemplars have their class labels mislabeled so that we can flip them back to their correct values and create more accurate classifiers from the de-polluted datasets. How can we know which exemplars have incorrect class labels? Let us treat this as an optimization problem where we want to maximize the number of correctly flipped exemplar class labels. The problem then is that we need some function to optimize. We don't know ahead of time which exemplars are correctly labeled, so we can't just use an obvious "number correct flips – number incorrect flips" measure --- this is the underlying problem we're trying to solve and we need some function which correlates fairly well with this but which we can compute. The hypothesis of this project is that, motivated by Occam's Razor which says to prefer the simplest model fitting the data, "model complexity" (defined more precisely later) is such a measure and function we can optimize for the purposes of de-polluting datasets. The key result of this project is to show that percent pollution in a dataset seems to be correlated with model complexity, specifically increasing pollution of the class 1 or 0 labels leads to increasing model complexity, which is evidence in favor of using model complexity as the function to optimize towards de-polluting datasets.

3. ARTIFICIAL DATASETS FROM RANDOM NEURAL NETWORKS

In my previous project described briefly above I used logistic regression as the classifier model of focus, both to generate artificial datasets (by sampling randomly from randomly generated logistic regression models) for testing and to determine accuracy drop-offs at increasing levels of pollution. Logistic regression is basically a perceptron with logistic activation at output, where the outputs are interpreted as probabilities of class membership; it is in the class of linear models. The problem with logistic regression, particularly for generating random artificial datasets, is that it is not general in the sense that it is a simple model (linear) and many possible underlying functions (i.e. nonlinear) could not be represented by it. Thus, in using artificial datasets sampled from random logistic regression models, general results applicable to "all possible functions" cannot be obtained. Multilayer perceptrons do not suffer from this difficulty. In fact, it can be proven that any continuous function can be approximated arbitrarily closely by a neural network with one hidden layer. Furthermore, any discontinuous function can be approximated arbitrarily closely by a neural network with two hidden layers. Thus, artificial datasets sampled from randomly generated multilayer perceptrons could be used to get statistics about classifier performance applicable generally to any underlying function; i.e. random multilayer perceptrons could be used as a general basis for generating random artificial datasets[ii]. Thus, for this project I generated artificial datasets by sampling from randomly generated neural network models with one hidden layer.

More specifically, to generate random neural networks with one hidden layer I would choose a random number of inputs and hidden layer units (up to some maximum number --- practically, in order to finish the project on the computational resources available, i.e. a laptop, I had to bound parameters such as this). Then, in order to consider the possibility of inputs being categorical (i.e. discrete valued rather than continuous), I would choose a number uniformly randomly from the interval [0,1] and the chosen number would be the fraction of the inputs that were categorical, which would then be chosen randomly from among all the inputs. Then, for each categorical variable I would pick a random number between 2 and some maximum which would represent the number of levels of that categorical variable. Categorical variables were encoded

[ii] Another possibility is simply to generate completely random datasets, but it seems more reasonable to work with datasets that have in fact come from some underlying coherent, albeit artificial, model; also, neural networks are used for modeling datasets with coherent structure, not random data, and we should try to best match our analysis to this.

using so-called "1 of C" encoding, e.g. three levels would be represented "1 0 0", "0 1 0", and "0 0 1" respectively[iii]. Logistic activation functions were used at hidden nodes and the single output node, where the single output of the network is interpreted as the probability of membership in class 1. The generated networks are fully connected (including bias units) and have random weights assigned uniformly on a specified interval. A random dataset is generated from a random neural network so constructed by generating random inputs and computing the network output, setting the class label to 1 if the computed output is greater than .5, 0 otherwise. Some filters to this random dataset generating procedure had to be applied, however. First, some datasets so generated might contain all or almost all 0s or 1s, and classifiers cannot be constructed from such datasets (there needs to be a reasonable number of examples from each class); thus, generated datasets were checked to make sure that each class represented at least 20% of the generated exemplars. Second, some randomly generated datasets seemed to suffer from some kind of numerical ill conditioning, specifically neural networks constructed from them of increasingly larger number of hidden units would all have very low prediction accuracy; thus, a second filter was to make sure that a highly accurate neural network model could be reconstructed from them (at least 95% accuracy on the dataset). Thus, random artificial datasets were generated in this way and used in the experiments / simulations described next. All experiments / simulations in this project were done using the R statistical package[6].

4. MINIMUM DESCRIPTION LENGTH PRINCIPAL

Providing some justification for why model complexity could be an effective measure to optimize in de-polluting datasets is the minimum description length principal ("MDL") machine learning paradigm. The essence of MDL is that the best theory (classifier) $\hat{T}$ for a body of data E is one that minimizes the size of the theory plus the amount of information necessary to specify exceptions relative to the theory:

$$\hat{T} = \arg\min_{T} L[T] + L[E|T]$$

[iii] See the Neural Net FAQ for reasons why categorical variables should be encoded this way: http://www.faqs.org/faqs/ai-faq/neural-nets/part2/section-7.html

Here L stands for length in bits of the best encoding of the theory and exceptions (the condition for "best encoding" is where practical difficulties using MDL can arise, i.e. how can you know you have encoded in the best possible way?) It can be proven using ideas from information theory that:

$$\hat{T} = \arg\min_T L[T] + L[E|T] = \arg\max_T \Pr[T|E]$$

i.e. the theory $\hat{T}$ with the maximum a posteriori probability given the data E, is the one that minimizes the size of the theory plus the amount of information necessary to specify exceptions relative to the theory[iv]. In other words, this is a kind of mathematical justification for Occam's Razor, which says to prefer the simplest model consistent with the data. The usefulness of MDL is that it allows you to construct your classifier model using all of your data, rather than having to wastefully set aside parts of it for test and validation sets, and still be assured you have constructed the best model. Note that in a real sense we cannot construct test and validation sets from our polluted datasets. We only have polluted data to build a classifier model, and we want to use the constructed model on real-world, unpolluted data, so testing and validating on independent test and validation sets gotten from the polluted data will not be an accurate measure of real-word accuracy. We thus ideally would like not to have to construct independent test and validation sets, and MDL does not require these which thus makes it attractive as a basis for dealing with the problem of polluted class labels.

The basic idea of MDL is that to find the best model you should minimize the size of the model and exceptions relative to the model. While MDL is strictly meant to apply to assumed unpolluted datasets, it was my motivation for this project and my hypothesis is that the basic MDL idea can be extended for de-polluting datasets. Here, the addition to MDL is that an additional parameter (beyond the usual classifier model parameters per se) can be optimized, namely the dataset exemplars' class labels (possibly input values too, but I didn't investigate that possibility for this project) --- the relabeling of class values in the exemplars that leads to the minimum sized model (plus exceptions relative to it) is the correct relabeling. Clearly this optimization cannot be taken to its extreme, however, for example we could simply label all exemplars with the same class (or all but one) and then a simple hyperplane could easily be found to separate the classes --- we will need to preclude such drastic, large-scale relabelings. However, if exemplar class relabelings

[iv] For a proof (and an overview of MDL) see reference 11.

are restricted to small scale increments, such as one or a small number at a time, then the method should work --- correct relabelings should lead to simpler models and incorrect ones to more complex models. The results of this project are not intended to and do not provide theoretical justification for why the MDL idea can be extended as described for de-polluting datasets; I feel this could be an interesting research direction but it is beyond the scope of this paper. The purpose here is to do exploratory analysis with simulations to try to obtain some suggestive evidence in support of further investigation.

A more intuitive explanation for why model complexity goes up with class pollution is that natural order in the dataset is destroyed by the pollution requiring more complex models to fit. A key assumption of machine learning is that the world is not random and that there are in fact organizing patterns to be found in data, and that exemplars with the same or similar class values will have the same or similar values of their important predictive attributes, i.e. like-valued exemplars will tend to cluster together in feature space (it is precisely these patterns that machine learning algorithms aim to find). Thus, for example, whereas before pollution subsets of the exemplars would have the same class values and be clustered in feature space, requiring only simple boundaries such as bounding boxes to delineate, pollution will cause some of these clustered exemplars to take on the opposite, incorrect labeling, and will invalidate the simple boundaries of the unpolluted dataset --- these will need to be partitioned finer to achieve the same level of modeling accuracy and this finer partitioning is precisely what is meant by increased model complexity.

5. NEURAL NETWORK MODEL COMPLEXITY MEASURES

Model complexity has been presented so far in fairly general terms, and this section will give precise definitions for model complexity for multilayer perceptrons. One definition I found was the following:

> "One important quantity determining the appropriate network architecture is network complexity. Network Complexity is determined (usually) by the number of adjustable weights and biases in the network. A simple Network has fewer adjustable weights than a complex one."[10]

I devised and tried three measures of model complexity for multilayer perceptrons motivated by this definition. Let ω be the vector of network weights and η the number of hidden layer nodes (neurons). The three complexity measures are defined as:

1. η
2. $\sqrt{\sum_i \omega_i^2}$ - i.e. magnitude of the weights vector.
3. Let $\omega^{abs} = abs(\omega)$. Let $\omega^{absnorm} = \frac{\omega^{abs}}{\sqrt{\sum_i \left(\omega_i^{abs}\right)^2}}$. Then complexity is

   $-\sum_i \omega_i^{absnorm} \cdot \log\left(\omega_i^{absnorm}\right)$ - i.e. entropy of normalized absolute value weights

All three measures will in general increase with increasing numbers of adjustable weights and biases in the network but are slightly different. The first measure is clear but is fairly coarse-grained. The second and third measures are similar but differ in how they deal with the distribution of total weight sum among all the adjustable weights. The third measure will tend to treat many small weights as more complex than a set of weights with similar total weight sum but concentrated in only a few weights (i.e. there is more information in finding out the value of something that can take on many possible values each with some small probability than in finding out the value of something that can take on a small number of values each with proportionally higher probability); the second measure will generally not make this distinction as it simply sums up the sum of squares of the weights and then square roots this.

Recall that in MDL, total complexity is the sum of the model complexity plus an encoding of the exceptions to the model relative to the model. If this weren't the case and you could ignore the exceptions, you could find the best model in MDL by giving the empty model --- it doesn't classify anything correctly but takes up no space. So far, only the definitions of complexity of the constructed multilayer perceptron models have been given but it is important to also add in the complexity of the exceptions. It is not clear how best to encode exception exemplars relative to the constructed model (i.e. presumably the constructed model will give partial information towards the exemplar's correct classification, but it is not straightforward what this is or how to use it to more compactly encode the exception exemplars), so my approach is to try to try to eliminate or minimize misclassified exemplars so that only model complexity really matters; i.e. tradeoff instance complexity for model complexity by building ever more complex models, stopping when you reach sufficiently large accuracy on the training set (e.g. 90% --- going much higher than this makes the computation time excessive). Nevertheless, we should still try to consider the complexity of the exceptions and this was done in this project by scaling relative to the accuracy of the model; e.g. if the model achieved X% accuracy on the training set then overall complexity was defined as $\frac{ModelComplexity}{X/100}$. This seems reasonable and it seems like it could underestimate the true complexity (i.e. the remaining exception exemplars are the most difficult ones and should be at least proportionally as hard to accommodate as the ones handled by the model). In any case, while the neural network models (and correction for the exceptions) per se are not necessarily as compacted as possible (which is technically what MDL requires), they do seem to level the playing field and provide a consistent measuring stick --- if the measures of complexity given above do in fact order the models in the same order the "true" measures of complexity would then that should be sufficient for our purposes (i.e. optimization).

6. SIMULATIONS

Again, the goal of this project is to provide empirical evidence that the complexity of multilayer perceptrons constructed based on polluted datasets will increase with increasing levels of pollution. The following simulation was done to try to obtain this evidence:

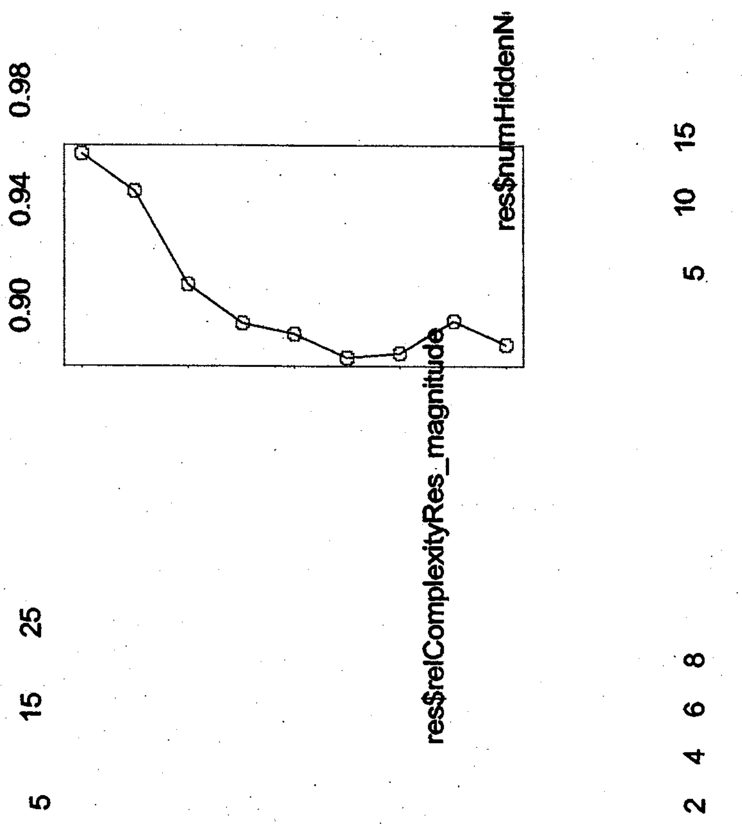
1. Generate a random multilayer perceptron model and random dataset sampled from it as described previously.
2. For pollution level = 0 to .4 by .05
    a. Create class 0 or 1 polluted dataset at current pollution level.
    b. For number of hidden units = 1 to large number
        i. Repeat up to 3 times (to address local minima problem): generate multilayer perceptron for polluted dataset with current number of hidden nodes. Measure accuracy on training set, exit loop if it is greater than .9.

c. Complexity at current pollution level = last generated multilayer perceptron complexity / accuracy.

In other words, we are looping through a series of pollution levels and for each one we search for the simplest model that gets 90% accuracy on the training data and compute its complexity (correcting for exception exemplars by dividing by accuracy); we then compare these complexity measures over the different levels of pollution and see if it increases with increasing pollution as hypothesized.

7. RESULTS

As stated previously, the simulations were performed using the R statistical package. I did not have the time or computational resources to do many runs and get detailed results and statistics. I did, however, run the code for the above simulation a number of times (10 – 20) and observed the tendency of the results. In general, my hypothesis is borne out and my complexity measures do increase with increasing levels of pollution, but not always perfectly (some runs show slight decreases in complexity measures for increasing pollution although the complexity of the most polluted is always higher than the not polluted case). The entropy-based measure (number 3 above) seems to be the best measure in that it seems to more reliably increase with increasing pollution; number of hidden levels is also good but, again, coarse-grained. Also, the results worsen as you lower the required accuracy threshold down from 90% (although smaller thresholds are quicker to compute), i.e. as the accuracy of the complexity measures of the exception exemplars relative to the model become more important. The following figure shows a typical result of a run of the simulation:



On all of the graphs, the x axis is the percentage of pollution added to the dataset. The upper left graph shows the prediction accuracy on the training set of the first (simplest) models found to have at least 90% accuracy versus pollution level. The upper right graph shows the number of hidden nodes, and the two bottom graphs the entropy-based complexity measure and vector magnitude-based complexity measure respectively, versus the pollution level. Only number of hidden nodes monotonically increases versus pollution level; the entropy-based complexity measure almost monotonically increases and seems visually slightly better than the vector magnitude-based measure (less and smaller decreases in complexity as pollution level gets larger).

8. CONCLUSION AND POSSIBLE FUTURE DIRECTIONS

In this study I have empirically determined that various complexity measures of multilayer perceptrons tend generally to increase with increasing levels of class pollution in a given dataset. If this can be shown to hold very generally, then the complexity measures could be used as the functions / measures to optimize in trying to de-pollute datasets using an optimization procedure. Future possible directions include the following:

- Do more simulation runs in order to compute statistics, confidence intervals, etc. and determine how general the result is.
- Investigate other interesting parameters of the problem; for example, does level of skew in the class distributions of the training set have any effect?
- Design and build detailed algorithms for de-polluting datasets based on the ideas from this project; compare them to other competing methods (e.g. one basic thing that can be done is to simply build a classifier with your training exemplars, and then remove any exemplars that are misclassified by it; this has been shown to simplify models and marginally improve prediction accuracy[3, 9, 11]). For example, use the neural network model complexity measures as the "fitness function" in de-polluting with genetic algorithms.
- Compare the performance of multilayer perceptrons optimized via MDL (using only training set) versus traditional techniques (e.g. training-testing-validation set protocol or 10-fold cross-validation).
- Consider feature selection and how it affects the problem. In other words, in this project I assumed we had knowledge of the exact set of relevant input features, but in reality you wouldn't (finding the right features is part of the art of data mining). Not using the right features is like adding noise, and de-polluting procedures could be confused if the right features aren't used; this argues for addressing feature selection and de-polluting in an integrated way.

REFERENCES

1. Goh C.S. et al. Mining the structural genomics pipeline: identification of protein properties that affect high-throughput experimental analysis (2004). *Journal of Molecular Biology*. Feb 6; 336(1):115-30.
2. Mitchell, T. (1997). Machine Learning. McGraw-Hill Science/Engineering/Math; 1st edition.
3. Muhlenbach, F., Lallich, S., and Zighed, D. A. Identifying and Handling Mislabeled Instances (2004). *J. Intell. Inf. Syst.* 22, 1 (Jan. 2004), 89-109.
4. Neural Network FAQ (2002). ftp://ftp.sas.com/pub/neural/FAQ.html
5. NIH Protein Structure Initiative (2005). http://www.nigms.nih.gov/Initiatives/PSI
6. R Project for Statistical Computing (2005). http://www.r-project.org/
7. Russell, S.J. and Norvig, P. (2002). Artificial Intelligence: A Modern Approach. Prentice Hall; 2nd edition
8. TargetDB structural genomics tracking site (2005). http://targetdb.pdb.org
9. Teng, C.M. Polishing Blemishes: Issues in Data Correction (2004). *IEEE Intelligent Systems*, vol. 19, no. 2, pp. 34-39, March/April, 2004.
10. University of Manchester UK CS2411 (2005). http://www.cs.man.ac.uk/~jls/CS2411/
11. Witten, I.H. and Frank, E. (2005). Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations. Morgan Kaufmann; 2nd edition.

Language Identification from Written Text

Simon R. Sprünker
Yale University, Department of Computer Science
New Haven, CT 06520

Abstract
We build a self-organizing map that is able to identify a natural language from written text. We start training the net with Vietnamese and English and after successfully building and testing this net with random examples of both languages, we turn our attention to a net that can identify German and English. Training sets, built nets and additional material can be found at (Sprünker, 2005).

1. INTRODUCTION

With the growth of the Internet, access to information from almost every part of the world is available. Many documents are not written in a comprehended language. But this ought not prevent one from accessing information in a foreign language. The task is to translate a given document, but in order to do that, one needs to know in which language the text is written. So the user would want to have a service he transmits the text to, and which then identifies the relevant language. Alternatively the translation service does the identification, but at some point in time before translation the identification has to be done.

Another feature is that search engines that crawl the web have to determine the language of a text in order to index it. So when a user performs a search, he can choose to have displayed only texts in his native language.

1.1 Algorithms For Language Identification

If we want to identify languages, we need to look at the distinguishing features of languages. Some key features are

- the alphabet of the language
- the combinations of characters a language uses and
- the frequency of occurrence of these characters.

So an algorithm for identifying a language from these features would first build a reference table, that for each language contains the alphabet of that language and for each character of the alphabet the frequency of occurrence. To build a reference table that is representative of a particular language, we would use large texts from different areas of this language, such as novels, science books and newspaper articles. When presented an unidentified text, the algorithm would compute the frequency of occurrence for all characters in the given text and compare it to the reference table. It would then choose the language, whose statistics are most similiar to the given text as an answer.

Since the frequency of occurrence of one single character can be very similiar in cognate languages, it is advisable to also take into account the relative frequency of so called n-grams, character groups of size n. This is a common practice in cryptoanalysis where n is typically 2 or 3 (Waetjen, 2004). The disadvantage of this method is that due to its statistical nature we need large texts for analyses, since otherwise the computed statistics are not useful. For example consider the text "A ball rolls down a hill". The most common character in english is the "e", but in this sentence "e" does not even occur!

If we take this method one step further, we could look at the words each language has. We can build a database, in which we put all the words of every language and tag each word with the language it exists in. Below is an algorithm using this information to identify a text:

```
Input: Unidentified text
Output: Set of most probable languages
set counter $c_i$ for each language i to $c_i = 0$;
for each word in the unidentified text {
        lookup word in database;
        for each tag of word {
                $c_i = c_i + 1$;
        }
}
return $\{i \mid c_i \geq c_j; j \neq i\}$
```

The disadvantage of this method is that we need to build the databases with all words of every language in it, which requires a huge amount of space. Of course we can combine the two methods presented and improve lookup times and the accuracy of identification.

In this paper we build a Neural Network whose purpose is to detect the features of every language with unsupervised learning, and based on that to identify the language of a text input to the net.

2. IMPLEMENTATION

2.1 Data Representation

Although a neural net could identify any number of languages, we will start building a net, that can distinguish between English and Vietnamese. We choose these two languages, because the differences between them are greater than English and German for example.

The Vietnamese alphabet contains Latin characters and diacritical marks. The characters f, w, j and z were not used originally, but they now appear in borrowed words. Additionally Vietnamese has modified Latin characters not present in the English language, such as „Đ".

a ă â e ê i o ô ơ u ư y

Table 1 - Vietnamese vowels (http://en.wikipedia.org/wiki/Vietnamese_alphabet)

b c d đ g h l m n p r s t v x

Table 2 - Vietnamese consonants (http://en.wikipedia.org/wiki/Vietnamese_alphabet)

Diacritical marks can only be used with vowels and different marks can be combined, however not every combination is valid.

À ầ ã ằ ộ ố ủ

Table 3 - Examples of valid vowel-diacritical-marks combinations

This is an example of a Vietnamese text: „Việt Nam có diện tích 331.688 km², bao gồm khoảng 327.480 km² đất liền và hơn 4200 km² biển."

We have to feed English and Vietnamese texts into the neural net. For this purpose we will take the plain UTF-8 encoded text and convert it to its binary equivalent, as the net will only read binary.

UTF-8 (8-bit Unicode Transformation Format) is a character encoding like ASCII. UTF-8 uses a variable length to encode characters. The most common characters are encoded with eight bits, but other characters take up to 32 bits. The characters we use have between eight and 16 bits.

We will implement a converter that for each character c of the input text performs the function

$$\text{conv:}\{c\} \rightarrow \{b_i \mid i \text{ element of } \{8,16,24,32\}\}$$

where b_i denotes a binary value with i bits.

2.2 Neural Net

For the implementation of the neural net we use Joone (http://www.joone.org), an open source java framework for neural nets. Nets built with Joone require input to be a semicolon separated list of values. So the character „a“, whose binary Unicode representation is 01101100, will look like this:

0.0;1.0;1.0;0.0;1.0;1.0;0.0;0.0

The net we build is a self-organizing map (SOM), meaning that it will adapt its synaptic weights according to the features it recognizes in the input data. When we input text into the net, we need to make sure that the text contains enough statistical data to be analyzed by the net, so that the input text is characteristic for the language it is written in. To achieve this, we need to make sure that the text we input is „long enough“. We will use 4496 input neurons. That accords to 562 characters of the english alphabet, because one character is represented by eight bits as explained in section 2.1, and less than 562 characters of the vietnamese alphabet, because one Vietnamese character can be represented by eight or 16 bits depending on the character.

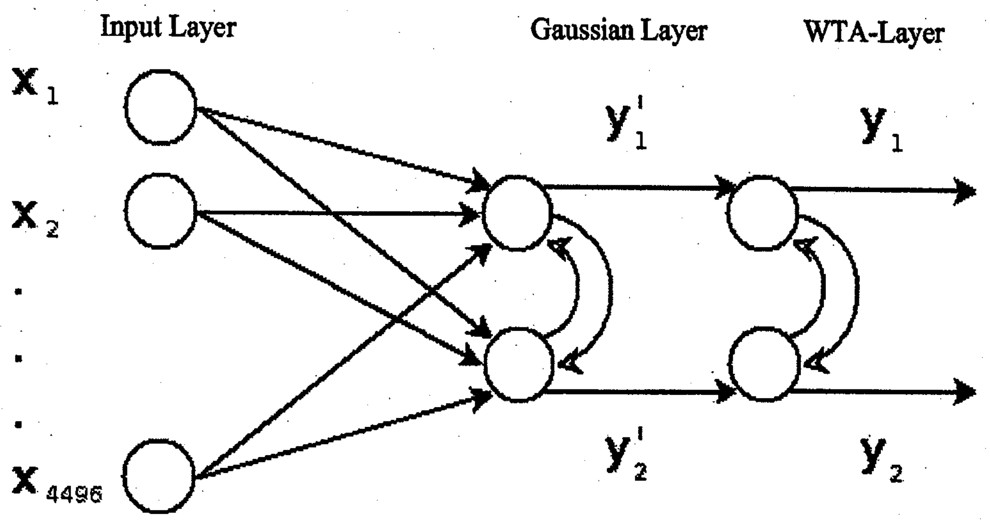


Figure 1 – SOM, that identifies Vietnamese and English

The net we build is shown in Figure 1 and described in detail in the next paragraphs. The input of the net is a vector

$$\mathbf{x} = [x_1, x_2, \dots, x_{4496}]^T,$$

where each x_i is 0 or 1.

The next layer is a *Gaussian Layer* with two neurons, that produces a vector

$$\mathbf{y}' = [y'_1, y'_2]^T,$$

where each $y'_i \le 1.0$. In our case the *Gaussian Layer* is one-dimensional, but normally it is two-dimensional. In a *Gaussian Layer* one neuron is the winning neuron. The winner is determined by comparing the weight vector $\mathbf{w}_j = [w_{j1}, \dots, w_{j4496}]^T$, $j = 1,2$, to the input vector $\mathbf{x}$ by computing the inner products $\mathbf{w}_j^T\mathbf{x}$, with $j = 1, 2$. The winning neuron i is the one with the largest inner product, and locates the center of a *topological neighborhood* $h_{j,i}$ of excited neurons, where a typical one of these neurons from the neighborhood is neuron j. All neurons in the neighborhood $h_{j,i}$ are excited with neuron i being excited the most and the other neurons in decreasing order depending on their *lateral distance* $d_{i,j}$, which is the lateral distance between neuron i and neuron j. That means that the *topological neighborhood* $h_{j,i}$ decreases monotonically with increasing *lateral distance* and time. A typical function for the *topological neighborhood* is the Gaussian function

$$h_{j,i(x)}(n) = \exp[-(d_{j,i})^2 / (2\sigma^2(n))],$$

which is also used by the Joone Network, where n is the discrete time. $\sigma(n)$ is called *Gaussian Size* and controls the degree of excitement in the neighborhood and is chosen according to (Haykin, 1999, p. 450) as:

$$\sigma(n) = \sigma_0 \exp(-n / \tau_1) \quad n = 0, 1, 2, \ldots$$

As output layer we choose a *Winner-Takes-All-Layer*, meaning that only y_1 or y_2 but not both will fire. So the output of the net will be either

$$y = [1.0, 0.0]^T \text{ or } y = [0.0, 1.0]^T,$$

where each element of vector **y** is one bit.

2.3 Training the net

To train the net we need to select a training set of Vietnamese and English text. To get representative texts wikipedia is very helpful, because there is a Vietnamese and an English version, the texts are already encoded in UTF-8 and the texts are representative since they are „real life" examples of both languages. We choose 20 pieces of text of each language of at least 562 Bytes length, and convert them to binary as described above.

Now that we have a training set, we need to choose the parameters of the net. In a SOM the weights of the synapses is set randomly in the beginning. The Joone Framework does this automatically for us.

As explained in the previous section Joone uses the same *Gaussian size*

$$\sigma(n) = \sigma_0 \exp(-n / \tau_1) \quad n = 0, 1, 2, \ldots$$

and *Gaussian neighborhood function*

$$h_{j,i(x)}(n) = \exp[-(d_{j,i})^2 / (2\sigma^2(n))]$$

as (Haykin, 1999, p. 452). We set σ_0 to 1.0, which is the radius of the vector y', and the time constant τ_1 to 200. This is the default in Joone, and it works for our problem.

The adaptive process, in which the synaptic weights of the SOM are adjusted consists of two phases: the ordering phase and the convergence phase. In the ordering phase the main adjustment of the synaptic weight vectors $\mathbf{w}_j$, j = 1, 2, takes place by using this formula:

$$\mathbf{w}_j(n+1) = \mathbf{w}_j(n) + \eta(n)h_{j,i(x)}(n)(\mathbf{x}(n) - \mathbf{w}_j(n)),$$

where $\eta(n)$ is the learning rate which is described below, and n is the time in iterations.

The initial learning rate η_0 is set to 0.7 and the ordering phase will last 1000 iterations, which both are default start values set by Joone. During the ordering phase learning rate decreases with the number of iterations as follows:

$$\eta(n) = \eta_0 \exp(-n / \tau_2) \quad n = 0, 1, 2, \ldots$$

where n is the number of the current iteration. We choose $\tau_2 = 1000$, so that $\eta(n)$ remains above 0.01 (Haykin, 1999, p. 452).

In the convergence phase the feature map is fine tuned. This is done by setting the learning rate $\eta(n)$ to 0.01 (Haykin, 1999, p. 453). Haykin states that as a general rule, the number of iterations in the convergence phase should „be at least 500 times the number of neurons in the network“. We have 4500 neurons in our net, resulting in at least 2,250,000 iterations.

3. RESULTS

3.1 English and Vietnamese

If the trained net is supplied with the training set, such that the first 20 pieces of text are Vietnamese and the second 20 pieces of text are English, we want to the 40 results of the Gaussian layer from Figure 1 to look like:

$$\begin{array}{lcl}
y'_1 = [0.8824969025845955;1.0]^T & & y'_1 = [1.0;0.8824969025845955]^T \\
\vdots & & \vdots \\
y'_{20} = [0.8824969025845955;1.0]^T & \text{or} & y'_{20} = [1.0;0.8824969025845955]^T \\
y'_{21} = [1.0;0.8824969025845955]^T & & y'_{21} = [0.8824969025845955;1.0]^T \\
\vdots & & \vdots \\
y'_{40} = [1.0;0.8824969025845955]^T & & y'_{40} = [0.8824969025845955;1.0]^T
\end{array}$$

Then the 40 results from the Winner-Takes-All-Layer should look like:

$$\begin{array}{lcl} y_1 = [1.0;0.0]^T & & y_1 = [0.0;1.0]^T \\ \cdot & & \cdot \\ \cdot & & \cdot \\ y_{20} = [1.0;0.0]^T & \text{or} & y_{20} = [0.0;1.0]^T \\ y_{21} = [0.0;1.0]^T & & y_{21} = [1.0;0.0]^T \\ \cdot & & \cdot \\ \cdot & & \cdot \\ y_{40} = [0.0;1.0]^T & & y_{40} = [1.0;0.0]^T \end{array}$$

We trained ten nets with 20 examples per language. Five nets were trained in 10,000 epochs and five nets with 5,000 epochs. All ten trained nets computed one of the above results, when presented with the training set. In addition all trained nets were presented with a test set, which contained 20 examples per language, but none of these examples were in the training set. They were taken randomly from Wikipedia. The nets identified the languages of all examples correctly. Notice that if the ordering phase lasts for 1000 iterations, then the convergence phase in this training lasts only 9,000 and 4,000 epochs, which is much less than the recommended 2,250,000 iterations from the previous sections.

Now one may ask, why we need the Gaussian Layer. Why not remove the Gaussian Layer and only have the WTA-Layer, since all we want is a clear answer? With clear we mean that we rather want to have an answer that looks like $y = [1, 0]^T$ which is produced by the WTA-Layer rather than an answer that looks like $y = [1.0;0.8824969025845]^T$ which is produced by the Gaussian Layer. The reason is that in Joone the Gaussian Layer and the WTA-Layer behave differently when the net is in the ordering phase. When we remove the Gaussian Layer, we get the effect that the net is not able to learn the training set. It will always produce the same output regardless of the input. This might be due to a maloperation by the author or an error in the Joone Framework.

3.2 German and English

The results in section 3.1 are very promising, but we made the task easy for the net by choosing long texts and by choosing two languages that have a great difference regarding the alphabet and the word stems. Now we will make it more difficult by choosing similiar languages.

Languages more similiar to each other are German and English. Lots of words have the same stem and the alphabet is almost the same. German only has the additional umlauts ä, ö, ü and the character ß, whereas English has the additional character ï, which is however rarely used and often replaced by i.

We trained the following nets:

# of nets trained	# of examples per language used for training	# of epochs used for training
5	20	5,000
5	20	10,000
3	20	20,000
5	10	5,000
5	10	10,000
5	10	20,000
5	5	5,000
5	5	10,000
5	5	20,000
5	2	5,000
5	2	10,000
5	2	20,000

Table 4 – Networks trained

The results of the net when presented with the training set were purely random. The nets were not able to identify the language of an example from the training set.

The reason why these nets do not work may be that we ignored the rule of thumb that we should have at least 2,250,000 iterations in the convergence phase as described in section 2.3. The problem with this much iteration is that the training of the net is excessively. For example: It takes 4 hours to train a net with 2 examples per language on a 3 GHz Intel Pentium 4, with 20 examples it takes 40 hours. To bring the training time to a feasible level we cut down the length of the examples to 400 input neurons. The training of five nets with 20 examples per language, five nets with 10 examples per language, five nets with 5 examples per language and five nets with 2 examples per language each with 2,250,000 epochs training leads to the same result, that is a purely random result when presenting a training example to the net.

4. CONCLUSION

There are statistical differences between German and English and we showed, that the net can learn to distinguish these differences. So is there a way to change the net with the result that it identifies German and English?

The fact that the results of the net trained with German and English were purely random suggests the interpretation, that the statistical features of examples from one language might be greater or equal to the statistical features from examples of the other language. In this case we would need to precompute the input, so that the differences between examples in one class are smaller than the ones between examples from both classes. Such methods exist, but we would need to evaluate the applicability of these methods.

REFERENCES

1. Sprünker, Simon. (2005), http://simon.spruenker.de/files/nnetlift.zip
2. Waetjen, Dietmar. (2004), *Kryptographie*. Heidelberg: Spektrum Akademischer Verlag GmbH
3. Haykin, Simon. (1999), *Neural Networks. A comprehensive foundation*. New Jersey: Prentice Hall
4. http://en.wikipedia.org/wiki/Vietnamese_alphabet