Neural Net Applications

Willard L. Miranker
February 2004
TR-1273

Table of Contents

Using Neural Networks to Model Competitive Behavior

Vladimir Barash
Yale University, Computer Science Department
New Haven, CT 06520

Abstract

A model of competitive behavior, consisting of actors descending a mathematical 'problem space' to a 'decision space,' is built. Simple algorithms for descending the problem space are surveyed and rejected. A feed-forward back-propagating neural network is used as an alternative algorithm. Several functions are implemented to add complexity to the model and to test the network's performance in non-standard minimization environments.

Keywords

competitive behavior, neural networks, descent, simulation

1. INTRODUCTION

This paper proposes to model competitive behavior. That does not imply, however, that it will model active, aggressive competition between individuals. The sort of competition to be analyzed involves actors working aggressively towards a common goal, but not necessarily impeding each other's progress along the way. Students at college, for example, are all aggressive and competitive; this does not imply that students at college push each other doer down and hurt each other's academic record as they strive for diplomas and graduation. Similarly, armies seeking to occupy a strategic landmark may exhibit competitive behavior without ever fighting (fighting in this "race for the hill" situation usually begins after one army has occupied the landmark in question).

Competitive environments have three crucial characteristics: the actors populating them; the problems those actors are trying to solve; and the solutions to the latter problems. These three characteristics surface in, for example, the competitive environment at a four-year American university. The actors are the students; the problems they are trying to solve can be assignments for a given class, or extracurricular commitments, or something else. Each of these problems corresponds to a solution.

When analyzing these and similar environments, one can organize actors, problems, and solutions within them in the following manner: every actor starts out with a set of unsolved problems; the solutions to those problems help the actor reach an

"ultimate goal," an event or condition where all problems are solved. In the university example, all students start out with a certain number of assignments to complete, extracurriculars they want to sign up for, and so on. As they complete assignments and finish classes, students slowly work their way towards a diploma – the "ultimate goal" that marks the end of their college problems (and their departure from the environment).

A good model of competitive environments, organized in the manner just described, might yield insight into the factors, determining individual success; the effect of other actors on individual performance; and the effect of irrational factors (such as stress, hope, or anxiety) on the ability of actors to cope with problems as they slowly proceed towards their "ultimate goal." In the case of the university, these results may help determine what helps, and what hurts, the individual student's chances of graduating, or how stress affects his or her progress. A general enough mathematical model would allow one to draw similar results for any applicable environment.

The model proposed situates actors, problems, and solutions in a 3 (or higher) dimensional manifold, which has strong local maxima in one portion, and flattens out to a global minimum everywhere else. The maxima represent the problems to be solved; actors (0-dimensional points) are placed at or near these maxima, and, in the course of the simulation, descend towards the global minimum (which stands for the actors' "ultimate goal"). In the course of descent, an individual actor's behavior is determined by three factors: his knowledge of problems he is currently attempting to solve; similar information from the rest of the actors; and the value of the actor's stress function. These factors are processed and weighed in a system of neural networks that computes the progress of all the actors over time.

2. THE MODEL

2.1 Manifold and Actor Set-up

The model consists of a three-dimensional manifold (a manifold of n dimensions may be used, but 3-D lends itself well to visualization) with the following characteristics: the manifold has enough discrete local maxima to situate as many actors as desired, and the manifold gradually slopes off to a flat "solution plane." The actors on the manifold are represented as individual points. Throughout the simulation, the actors are initially aware only of four pieces of data: the numerical gradients in the four cardinal compass directions around them. In other words, an actor at point *(m,n,o)* on the manifold is aware of (*dz(m+1, n)/dx; dz(m-1, n)/dx; dz(m, n+1)/dy; dz(m, n-1)/dy*) – where z is the variable name for the function describing the manifold. This restriction of local-only awareness is suggested by March and Olsen's (1972) work on organizational choice: the actors in March and Olsen's simulations are never aware of the entire scope of problems to be solved, but only of the problems immediately before them at a given point in time.

The manifolds used in this experiment fall into two categories: initial, the ones used to train the system of neural networks; and experimental, the ones the trained system was used on. The surface equations of the manifolds in the first category are:

(1) $z = \sin(x)/x + \sin(y)/y$

(2) $z = y*\sin(y)+x*\sin(x)$

The surface equations of the manifolds in the second category vary; most are deterministic and given by trigonometric functions of x and y; some are non-deterministic, for example:

(3) $z = \sin(x)/x + \sin(y)/y + k*\text{rand}(0,1)^{i}$

(4) $z = y*\sin(y)+x*\sin(x)+k*\text{rand}(0,1)^{ii}$

where k is a constant between 0 and 1 (several different values were used in the course of the experiment).

2.2 Actor Decision-making

Simulation on the model is done in fixed-step time. Each time step consists of two parts: first, every actor on the manifold evaluates the information he has and uses it to make a decision; second, the actor positions are updated according to the decisions they have made. The simulation ends when no actor makes a decision to move. At that point, the results were recorded quantitatively (in tables and figures) and qualitatively: any actors that came within the vicinity of the "ultimate goal" were considered to have nearly succeeded, while those who reached the "ultimate goal" itself succeeded completely.

It is important to note that the actors never impede each other's progress, in accordance with the assumptions outlined in the Introduction: if two actors ever meet, they go over the same ground (just as two students at college might take the same classes) without affecting each other's input data, decision-making, and so on.

The actors' decisionmaking ability was developed in the course of the experiment in roughly three stages. Originally, each individual actor made his or her own decision without the aid of a neural network, using a simple algorithm, sketched out below:

*Evaluate(*gradient_north, gradient_south, gradient_east, gradient_west*);*
*If(|average(*gradient_north, gradient_south)| >= |*average(*gradient_east, gradient_west)|*)*
    Newposition = *greater*(gradient_north, gradient_south);
Else
    Newposition *greater*(gradient_east, gradient_west);

End

where *greater* is a function that picks out the greatest element, among two, by absolute value. This stage was tested on its own manifold, defined by:

(5) $z = (x+.5)^2+(y)^2$

as well as on first category manifolds, described by (1) and (2).

The second stage of development did involve neural network architecture, but the actors still had no way to communicate with each other. Every actor corresponded to a whole system of three back-propagating feed-forward networks. The first two networks interpolated cardinal compass direction gradient data for each actor. Pre-formatting allowed these networks (dubbed *NorthSouth* and *EastWest*) to calculate the more favorable direction gradient (see Appendix 1). The networks then sent their outputs, in gradient format, to the third network, to be stored as elements of its weight matrix. The third network produced a final, most favorable gradient, and thus indicated the direction, in which the specific actor should move.

After the actor's position was updated according to the direction indicated, an external teaching source checked the net's progress by evaluating the distance above the solution plane for each actor at his updated position. If this distance = 0, the actor had reached his goal, and made no more movements for the rest of the simulation. If not, the third network was trained, via back propagation, with targets set at some constant $0<k<1$ * the absolute value of its outputs. The training yielded new weights, which were then passed on to the first and second network, respectively, as training targets. The weights were multiplied by a relatively large (of the order of 10^2 * the weight values) positive constant. This 'growth factor' resulted in a large numerical difference between the weights, which prevented the system from getting stuck in a spot where the numerical differences between input data were very small. As the networks were trained, their weight matrices changed, creating a network-intrinsic preference for one direction over another. This stage was trained on training manifold (1) and generalized to a sequence of non-deterministic manifolds of the form of (4), with k raised incrementally by from .1 to .9.

The third stage had a neural network architecture similar to the second, but actors could now communicate and influence each other's decision-making. The inputs to each of the first two networks were north-south and east-west data for all actors, respectively. The initial weights were set so that the data, corresponding to an actor's cardinal compass direction gradients, was given the most influence over the actor's decision-making (roughly fifty times the data, corresponding to other actors' gradients). Furthermore, the third network was trained with targets set as the most negative of its outputs (which

corresponds to the actor currently approaching the "ultimate goal" the fastest). Finally, the biases sent to each neuron in the first two networks were given values corresponding to the neuron's stress function (Appendix 2) at the given time[iii]. This stage was trained on all three manifolds in the first category separately, creating three trained network systems. The systems were then generalized to six second-category manifolds: three deterministic and three non-deterministic.

3. RESULTS

3.1 Stage One

Stage one decision-making served as a baseline for actor performance in this experiment: actors in this stage acted by a set of simple, constant rules and underwent no learning at all. The learning-based models in stages two and three of decision-making had to perform better than this baseline, or there would be no cause at all to use neural networks instead of simpler methods in the experiment.

The simulations of stage one decision-making used a set of nine randomly chosen actors, who performed descent in fixed time until they were either successful or 100 simulation steps had elapsed. The actors' progress was collected as distance from zero vs. time in a table, and represented graphically by slightly elevating points on the manifold's surface where the actors were located in the course of the simulation.

The manifold stage one decision-making was first simulated on (described by (5)) did not meet the model requirements described in 2.1, and was used entirely to test the validity of stage one's algorithm on some surface. The algorithm used allowed eight out of ten actors to perform nearly successful descent, reaching a distance of .05 units from zero in at most thirteen simulation steps. The remaining two actors did not descend successfully past a distance of .75 units from zero.

Next, stage one decision-making was simulated on first category manifolds, described by (1) and (2). On manifold (1), five out of ten actors performed nearly successful descent; on manifold (2), six actors stopped their descent at the local minimum near the origin, and one of the remaining actors got stuck somewhere between the local minimum and the "ultimate goal;" only the last three actors performed nearly successful descent.

The tables and figures corresponding to this stage of decision-making are marked 1a and 1b – corresponding to simulations on manifolds (5) and (2), respectively.

Table 1a

Actor	Initial	Final
1	0.85994	0
2	1.0019	0
3	1.244	1.0268
4	1.6681	0
5	1.8397	0
6	2.2684	1.1743
7	1.1941	0.0257
8	1.7034	0.2281
9	1.7473	0.6177

Table 1b

Actor	Initial	Final
1	5.6933	4.6265
2	8.5012	4.7801
3	9.6311	4.8252
4	9.15	5.5338
5	8.2583	6.1339
6	6.668	6.0644
7	4.3337	3.1830
8	8.578	6.0644
9	8.9303	6.0644

Figure1a

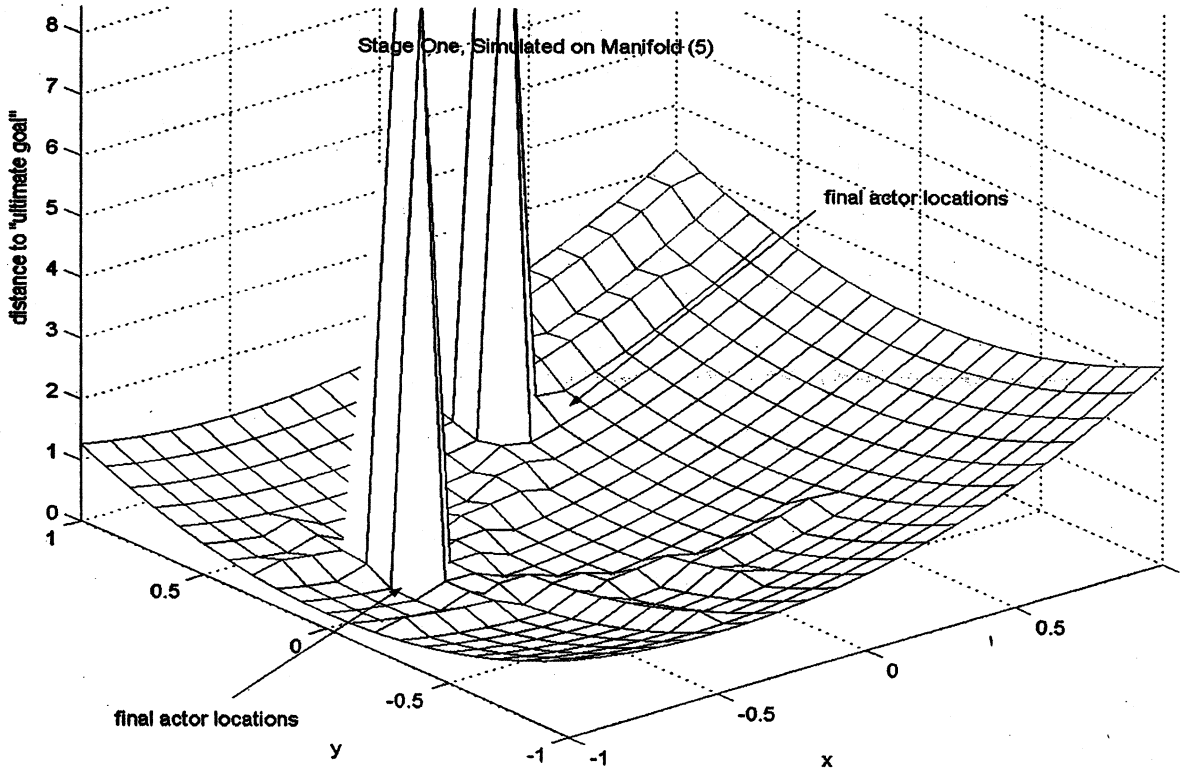
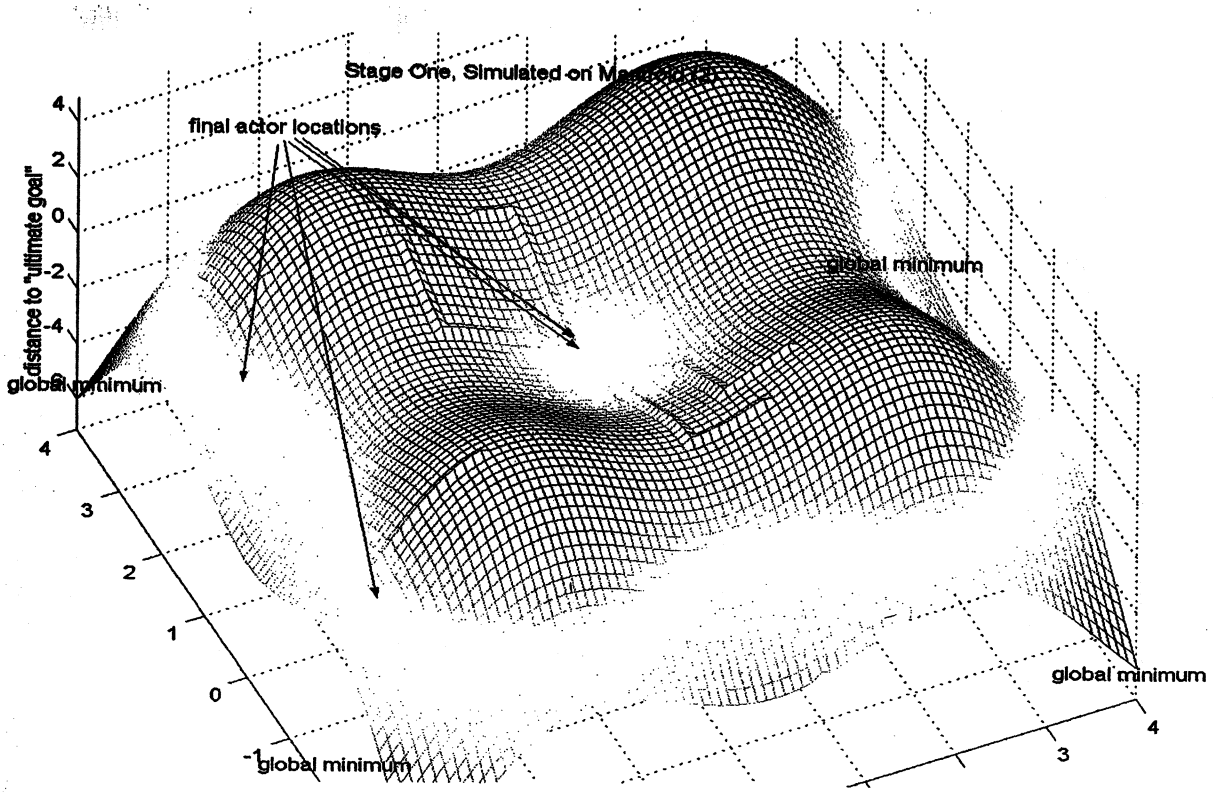


Figure1b



3.2 Stage Two

Stage two decision-making involved a system of neural nets that was trained on first category manifolds, described by (1) and (2), and simulated on the non-deterministic versions of those manifolds – (3) and (4), as well as on two deterministic manifolds:

(6) $z=x*\sin(\cos(2*x))+y*\cos(\sin(2*y))$

(7) $z=\cos(x/110*\log(x))+\sin(y/110*\log(y))$.

The number and initial locations of actors were kept constant throughout the simulations. Training was run for fifty steps; the simulations were run for varying lengths of time – from thirty to a hundred steps – depending on the point where the majority of the actors would cease descending and further simulation would achieve nothing new.

The network system trained well on manifold (1) – two of the eight actors performed completely successful descent, reaching a distance of 0 from the "ultimate goal," and three others made a progress of .6 units in the course of their descent. Two actors, however, did not descend at all, and got stuck at point close to their initial positions.

The network system trained far worse on manifold (2) (no successful or even semi-successful descents), and performed very poorly in all simulations. The only exception was the simulation on manifold (7) of the system, trained on manifold (2). In

the latter case, one actor, initially positioned close to the "ultimate goal," did perform nearly successful descent. The tables and figures corresponding to this stage of decision-making are marked 2a , 2b (for the two training manifolds) and 2c (for the simulation with one successful descent).

Table 2a

Actor	Initial	Final
1	2.3797	0
2	3.5316	0
3	2.9925	1.2869
4	2.3268	0.8139
5	0.74473	1.1095
6	2.7593	2.2734
7	3.5715	1.6681
8	1.3741	0.8542

Table 2b

Actor	Initial	Final
1	0..81853	2.2931
2	.95895	3.1583
3	1.2869	1.5602
4	1.6279	2.3268
5	1.7968	0.7447
6	2.2734	2.7593
7	1.7084	3.5657
8	*Error*[iv]	*Error*[v]

Table 2c

Actor	Initial	Final
1	2.3797	1.19972
2	3.5316	1,9972
3	2.9925	1.1004
4	2.3268	1.1969
5	0.74473	1.1969
6	2.7593	1.9099
7	3.5715	3.9808
8	1.3741	2.3412

Figure 2a

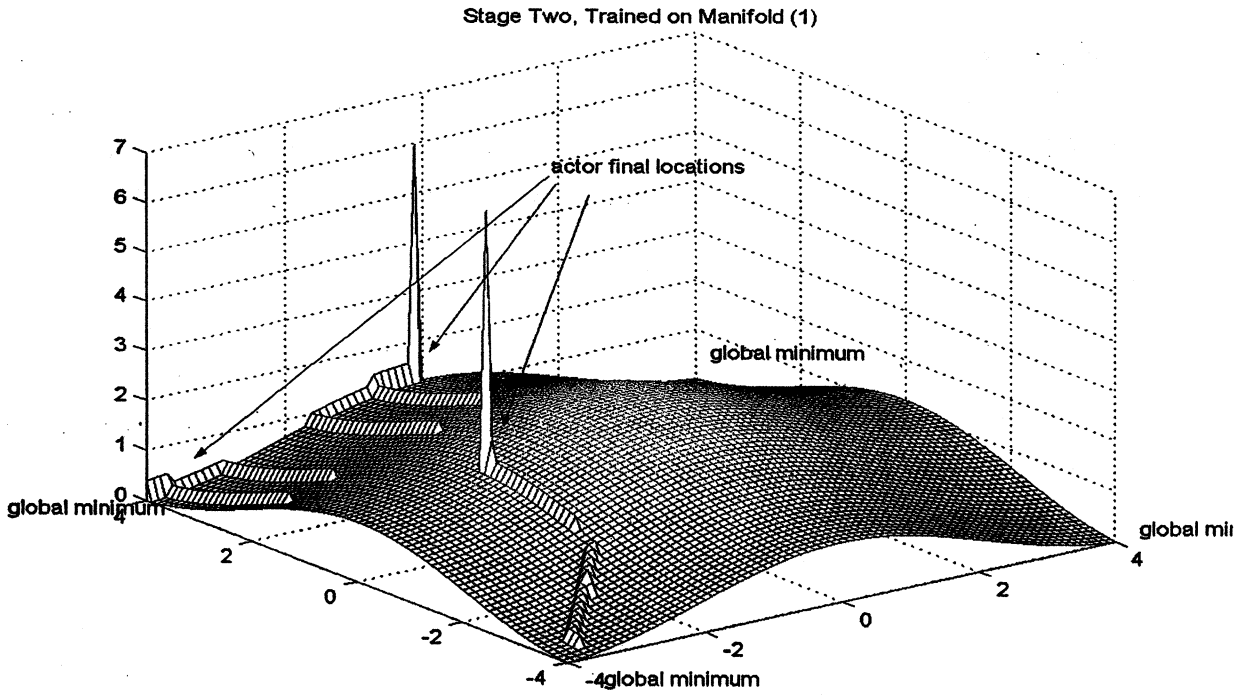


Figure 2b

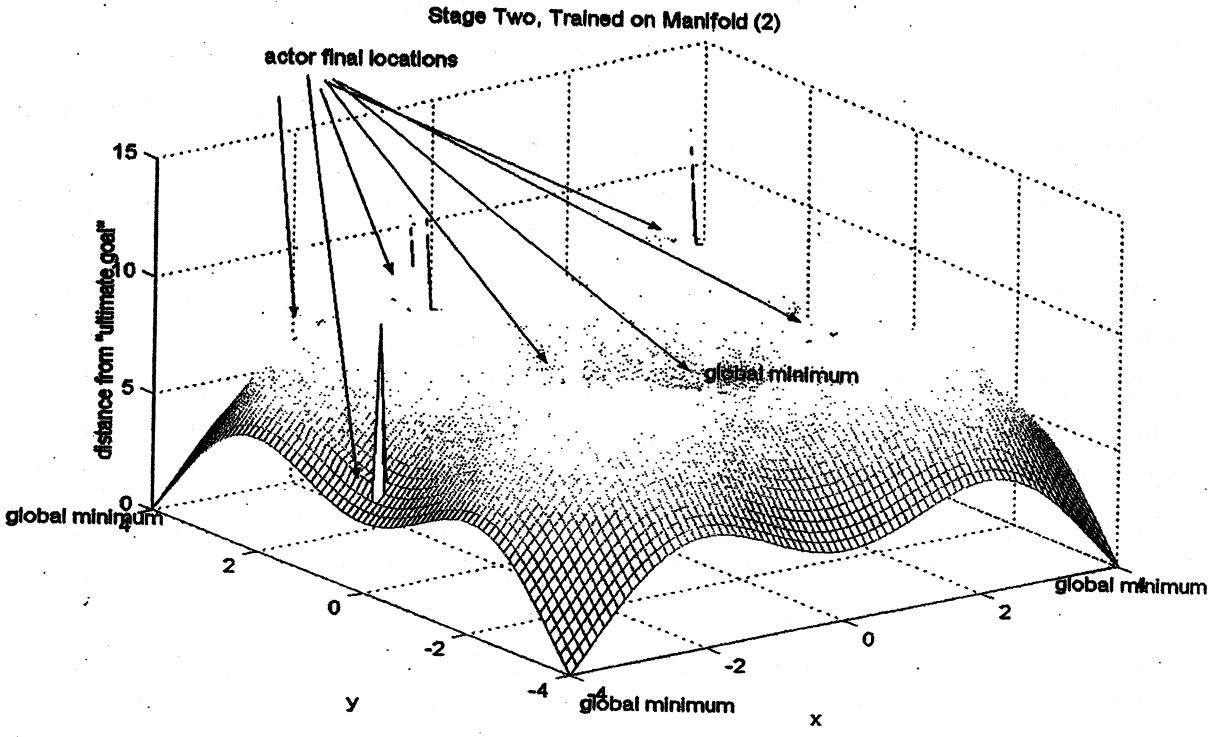
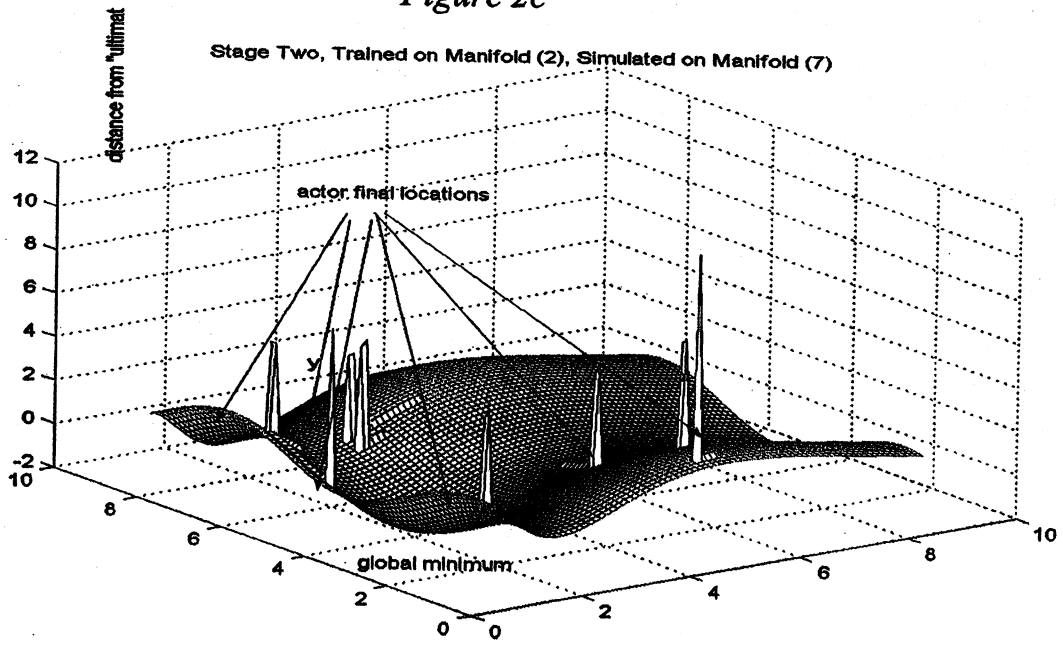


Figure 2c



3.3 Stage Three

Stage three decision-making was trained and simulated on the same manifolds as stage two decision-making, so that surface equations acted as a control for the last two stages of decision-making in the experiment. The initial locations of actors coincided with those from stage two, as did training and simulation times (fifty for training, varying from thirty to a hundred for simulation).One new actor was added to stage three simulations to test the effect of the number of competitors on individual progress; their locations were kept constant for all manifolds trained/simulated on.

Stage three decision-making performed significantly better than stage two, both in training and in simulation. Its training on manifold (1) resulted in five actors performing nearly successful descent (final distance from "ultimate goal" ranged from .6 to 1.15 units); its training on manifold (2) resulted in one completely successful descent and two nearly successful descents (final distance from "ultimate goal" ranging from 1.0 to 1.4 units; furthermore, as the appropriate figure shows, these actors are located only a few squares off the absolute minima of the manifold). The training on manifold (2) was also remarkable, because every actor avoided the local minimum near the center of the origin – even the actors that started near the local minimum ascended away from it and towards the global minima of the manifold. The nets, trained on manifold (1) performed with similar-to-training results in simulation on manifold (3), and worse on manifolds (4) and (6), but achieved one nearly successful descent on manifold (7), with a final distance from "ultimate goal" of 0.87. The nets, trained on manifold (2), resulted in one successful descent and four near-successful descents when simulated on manifold (3); in seven near-successful descents when simulated on manifold (4) (with one point just .08 above "ultimate goal"); and in another seven near-successful descents when simulated on manifold (7). The tables and figures corresponding to this stage of decision-making are marked 3a and 3b (for the two training manifolds), 3c, (for net, trained on manifold (1), simulated on manifold (7)) and 3d and 3e (for nets, trained on manifold (2), simulated on manifolds (4) and (7)).

During training on manifold (2), one of the actors descended successfully before the entire span of the simulation was over, and, due to a glitch in the program, caused the training to stop. For the purposes of further simulations, the networks were left as trained when the successful descent happened. One more simulation was made at the very end of the experiment: the final positions of the actors, minus the one that was located at "ultimate goal," were re-inserted into the networks and training was continued; the removal of the successful actor allowed another actor, previously stuck some distance from "ultimate goal," to descend successfully – but nothing further of interest was gained by this simulation or by subsequent simulations of this kind.

Table 3a

Actor	Initial	Final
1	0.81853	0.6027
2	0.95895	0.8284
3	1.2869	1.1134
4	1.6279	1.3706
5	1.7968	1.6662
6	2.2734	2.2602
7	1.1763	1.1763
8	1.6745	1.6745
9	1.7084	1.7084

Table 3b

Actor	Initial	Final
1	5.7799	1.4117
2	8.5462	3.0372
3	9.586	7.8331
4	9.0469	5.1508
5	8.3812	4.8059
6	6.7991	6.2499
7	4.2498	1.0668
8	8.5574	6.7594
9	8.8137	7.4340

Table 3c

Actor	Initial	Final
1	3.0717	1.5082
2	1.8987	1.0486
3	0.88942	0.6928
4	2.5406	1.4969
5	3.5291	2.6333
6	1.8724	1.3486
7	3.9798	2.0088
8	2.7587	1.8959
9	2.1681	0.8783

Table 3d

Actor	Initial	Final
1	5.6338	4.1537
2	8.5723	3.7781
3	9.6498	7.8833
4	9.2657	5.7574
5	8.4745	7.3939
6	6.5571	6.0809
7	4.4461	2.8979
8	8.6036	6.8007
9	8.9049	7.5058

Table 3e

Actor	Initial	Final
1	3.0717	1.7486
2	1.8987	0.7289
3	0.88942	0.1635
4	2.5406	1.4969
5	3.5291	2.3137
6	1.8724	1.4954
7	3.9808	2.9572
8	2.7587	1.3231
9	2.1681	1.8879

Figure 3a

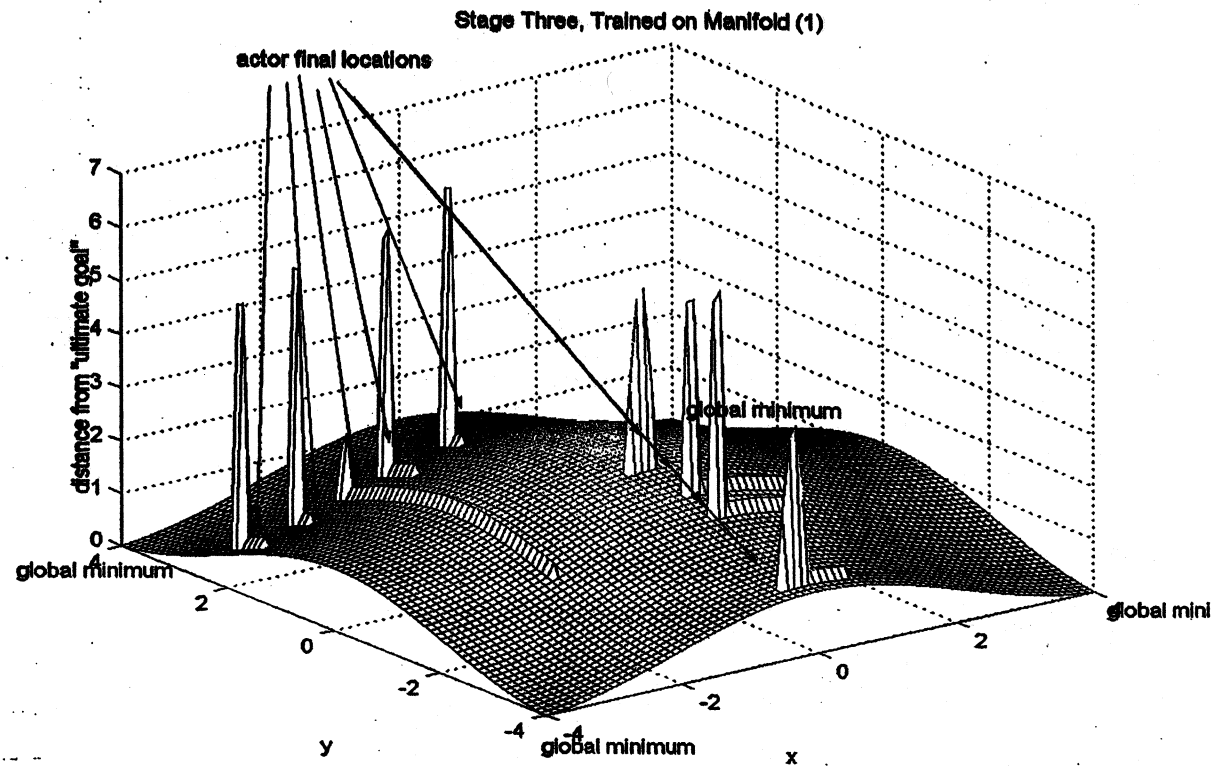


Figure 3b

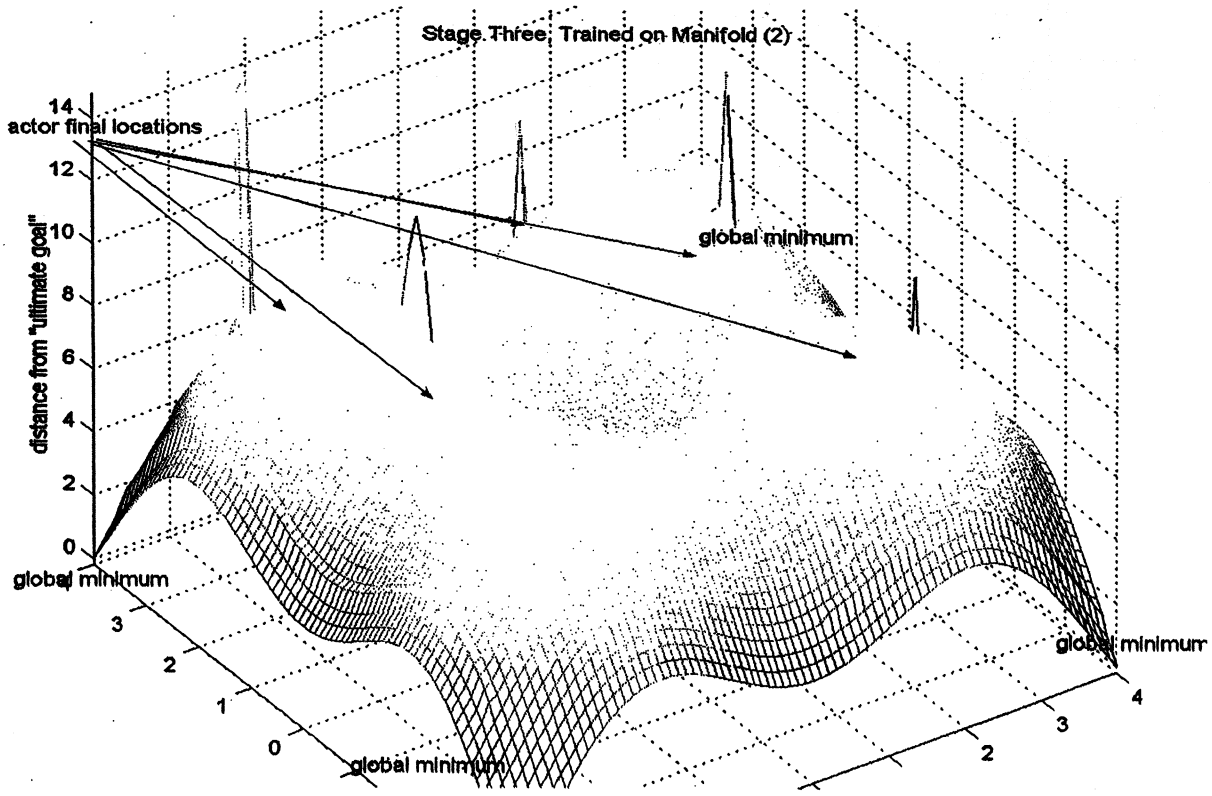


Figure 3c

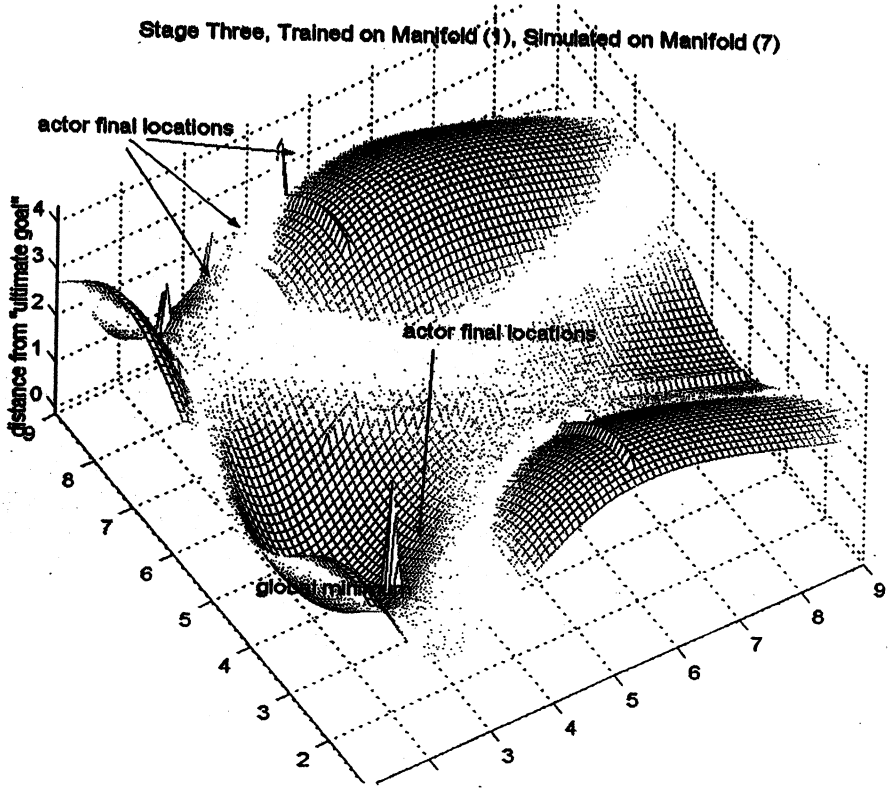


Figure 3d

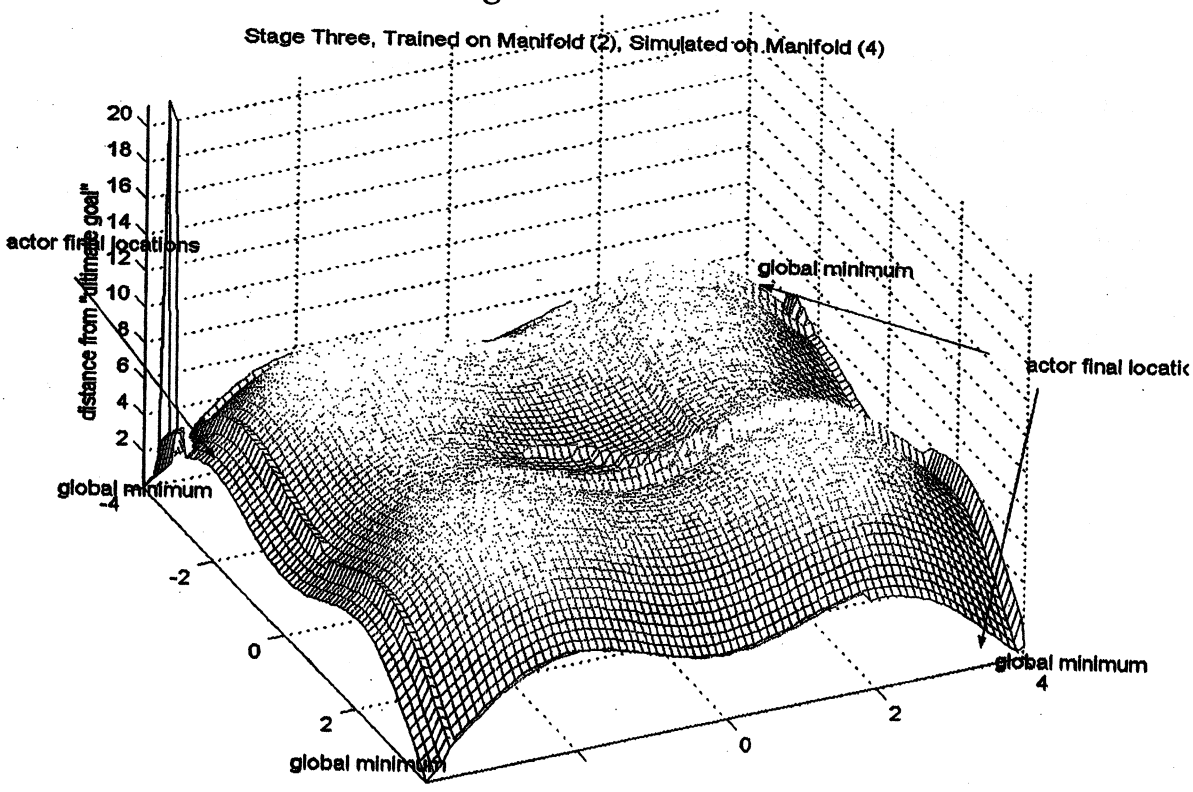
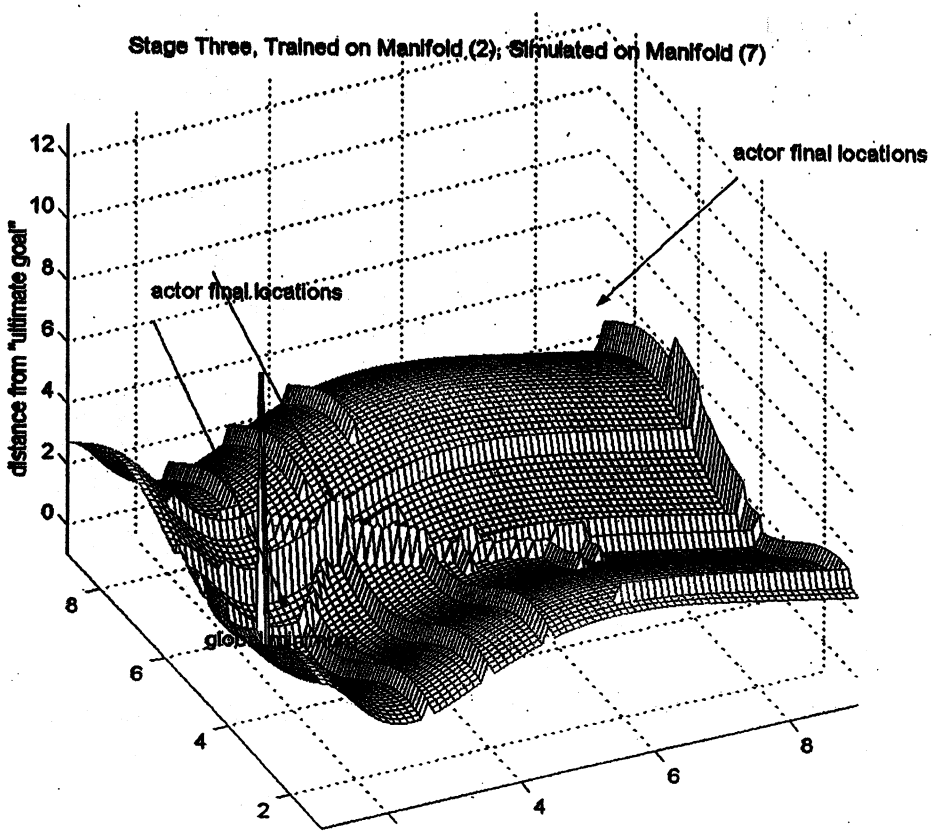


Figure 3e



1. CONCLUSIONS

There is a clear progression from stage one to stage three of actor decision-making. Stage one yields the best results for simple manifolds, like the ones defined by (5) and (1). In the case of even slightly more complex manifolds, however, stage one decision-making fails to locate the global minimum, as its simulation on the manifold, defined by (2), clearly shows: all actors descended towards the local minimum – representing a false solution to their problems (in reality, the furthest point away from their "ultimate goal" in manifold (2)) – with the exception of those that started out far away from and below it. For the sake of compactifying the data presented, additional simulations of stage one are not shown here, because the results of those simulations only confirm the conclusion drawn from stage one's performance on (2) – without the ability to learn and generalize, a decision-making algorithm will not be able to adapt to complex environments.

Stage two performs worse the worst of all three stages, possibly because it lack both the simplicity of stage one decision-making and the complex learning involved in stage three decision-making. Perhaps this failure is due to faults in the technical implementation of stage two; yet there may be another reason: individuals, whose behavior stage two can be said to model, attempt to succeed in competitive environments

without interaction with other individuals. Many real-world examples indicate that such attempts are doomed to failure; consider the student, for example, who attempts to go through a college in the USA without talking to his professors, or interacting with his peers, or anything of the sort. There is a small chance such a student might succeed, but his or her inability to receive valuable feedback about his or her progress effectively reduces this chance to zero. The same can be said of the army that attempts to secure a strategic position without interacting with friendly forces in the area, the civilian population, or the intelligence community: its endeavor is underprepared and will fail miserably if a more communicative force moves to prevent the army's attempts. Stage two, then, may be useful as a model despite its poor performance in the experiment, if only as an indicator of failure in the model where failure is expected in the real world.

Stage three's performance varies from poor on the simple manifolds to exceedingly good on the complex ones. Visually, the huge bulge in figure 3e indicates the collective movement of most actors towards an area very near the global minimum (*not* any of the local minima) in the course of the simulation. Similarly impressive is the progress shown in figures 3b and 3d: in training and in simulation, all actors avoided the local minimum of manifold (2), and left the minimum's vicinity even if they were nearby initially. This behavior indicates a high probability of learning by the model's stage three decision-making algorithms. The neural networks those algorithms are made up of interpolate data from all actors on the manifold, and thus are able to "realize" some actors are located in false minima without any external knowledge. Those actors are then encouraged to leave the false minima and head for the true, global minima, which is what happens in figure 3d (notice the large number of final actor locations near the four global minima of the manifold). Thus, despite its sub-par performance on simple manifolds, stage three decision-making seems to be the best model of competitive behavior developed in this experiment. It represents the behavior of individuals who attempt to succeed in a competitive environment not by simply heading for the goal, but by communicating with and aiding each other. Alone, these individuals get stuck and fail; together, they can all (or almost all) succeed, or nearly so.

Of interest is the successive training described at the end of the Results section. While clearly beyond the scope of the present experiment, this idea may be the next step in developing the models presented here; if the entrance and exit of actors into a competitive environment helps actors who stay in it, it may be worthwhile to apply the three-stage model complex to a constantly changing environment, with free actor entry and exit. On the practical level, such a model may be of use specifically in the economics of competitive markets with free entry and exit.

2. REFERENCES

1. M.D. Cohen, J.G. March, and J.P. Olsen(1972). A Garbage Can Model of Organizational Choice. *Administrative Science Quarterly*, v17 pp.1-25.

3. APPENDIX A

6.1 Data Pre-formatting

Stages two and three of actor decision-making involve a system of three neural networks, the first two of which accept cardinal compass direction gradient data for actors or whole sets of actors. The data is pre-formatted as follows: the first element of each of the two direction gradient pairs (north-south and east-west) is designated *a*, the second *b*. Four cases are singled out:

(8) $a<0, b<0$
(9) $a>=0, b>=0$
(10) $a<0, b>=0$
(11) $a>=0, b<0$

The weights of the first two networks were so arranged that the greater element passed on to them would be chosen as the output. In case (8), the absolute values of *a* and *b* were passed on to the network: this meant that the greater (by absolute value) gradient among *a* and *b*, corresponding to the direction of steeper descent, would be chosen. In case (9), the order of *a* and *b* was switched: this meant that the smaller gradient among *a* and *b*, corresponding to the direction of slowest ascent, would be chosen. In cases (10) and (11), the element >= 0 became 0, so the direction it corresponded to would not be chosen, and the actor would go down instead of up.

4. APPENDIX B

7.1 Tables and Figures

The figures are x-y-z graphs of actor progress on an individual manifolds, created in Matlab 6.0 using the *mesh* command. The "ultimate goal" areas are not drawn into any of the figures, but the global minimum/minima is/are marked to indicate the areas connecting the "ultimate goal" to the rest of the manifold. The points traversed by the actors are slightly elevated above the surface of the manifold; this visualization does not, regrettably, allow for differentiation between the paths of individual actors, but indicates the general trends of progress in the course of the simulation. The ends of actor paths are elevated higher than the rest of the paths (they look like "jagged peaks" on the surface of the manifold).

The tables give clearer indication of individual actor progress: they record the initial and final distances from "ultimate goal" for each actor. The first column lists the actors, the second the initial distances, and the third the final distances.

Three final tables give sample weight matrices for stage two and stage three decision-making network systems. The first two tables are from stage two decision-making, and give the north-south and east-west matrices when the system is trained on manifold (1). The last table is from stage three, and gives the east-west matrix when the system is trained on manifold (2).

[i] Rational number, randomly chosen from normal distribution on [0,1].
[ii] Rational number, randomly chosen from normal distribution on [0,1].
[iii] The neuron stress function, which basically multiplied the gradient of descent in the direction, currently chosen by an actor, by the current simulation step, did not have a significant effect on the results of any training or simulation, and thus was left out of the final write-up.
[iv] MATLAB yielded negative values for this row, which is theoretically impossible.
[v] MATLAB yielded negative values for this row, which is theoretically impossible.

A Simulation of Neurogenesis in a Neural Network

Michael Bell
Yale University Department of Computer Science
New Haven, CT 06520

Abstract

Neurogenesis is an observed phenomenon in which new neurons and synapses are formed in parts of the adult brain. In a neural network, we can simulate the death and rebirth of neurons by occasionally resetting the weights of synapses between a neuron and any neurons it is connected to. This throws away all of the old data that the neuron had accumulated over time and stored in the weights of connected synapses, and replaces it with that of a neuron in its initial state. This model should inherit some of the benefits of the young neural net - nodes that have not yet become saturated - while continuing to make use of the majority of the information that it has already learned. Here we examine several rates for neuron growth/death in a three-layered neural network. The network uses Hebbian learning to adjust its weights and classify elements of an input set. Once the elements of the set are classified consistently (that is, as the element is repeatedly exposed to the network, its classification no longer changes), the network is exposed to a new, similar data set. We investigate how the rate of neurogenesis affects how long it takes for this new data set to be classified consistently.

Keywords - neural networks, neurogenesis, Hebbian learning

1. INTRODUCTION

Over time, synaptic weights in a neural network become saturated and resistant to change. This resistance to change makes learning new data sets inefficient or impossible. Neurogenesis is an observed phenomenon in which new neurons and synapses are formed in parts of the adult brain. By introducing new, unsaturated synapses, neurogenesis allows the brain to inherit some of the benefits of a young neural network - nodes that have not yet become saturated - while continuing to make use of the majority of the information that it has already learned. We can simulate this by occasionally resetting the synaptic weights between a neuron and any neurons it is connected to, as if it were the death and rebirth of the neuron. This model should inherit some of the benefits of the young neural net - nodes that have not yet become

saturated - while continuing to make use of the majority of the information that it has already learned.

Our simulated neurogenesis model has its foundation in an observed biological phenomenon. Kornack and Rakic [1] have found signs of neurogenesis in the hippocampus of the adult macaque monkey, and estimate that at least one neuron per 24,000 existing ones have been generated in the past day. That is to say, 0.004 percent of neurons are newly generated each day. This result lends itself to the probabilistic model of neurogenesis that we implement for our simulation; during each iteration of the network, neurogenesis occurs with some probability p (such as 0.00004).

2. PROBLEM SETUP

The basic structure for the neural network that we test our hypothesis on is that of a a three-layered network where any two neurons in adjacent layers are connected by a synapse. At some time t, the neurons in the first layer are presented with input values of -1 or 1. The value n_t for a given neuron n in the remaining two layers is calculated as the sum of the values of the neurons in the preceding layer, weighted according to the synapses between the neurons and n. If the value of the weighted sum calculated at n is above zero, then $n_t = 1$, otherwise the $n_t = -1$.

The values of the neurons in the third layer, the output layer, are based on the input values and can be thought of as the network's encoding for that particular input. When the network consistently gives the same encoding for a given input, we will consider it to have "learned" that input. The network accomplishes this through basic Hebbian learning. After an input has been presented to the network, and the values of each of the neurons in the network has been determined, the weights of all synapses are adjusted. When both neurons on either side of a synapse are active (both have a value of 1) or both are inactive (have a value of -1), we have a change in the weight w of the synapse corresponding to $\delta w = \eta$ (where η is the learning rate); when the two neurons don't have the same value, we weaken the strength of the synapse between them by $\delta w = -\eta$.

For the implementation of neurogenesis in the network, we take a simple model for neuron death and birth. Rather than construct a network which allows variable numbers of neurons, we simulate the death and simultaneous rebirth of a neuron by resetting the weights of the synapses to and from the given neuron.

We use a probability p for resetting nodes in this manner, and test p as it varies over certain values. Since we have no other way of measuring time intervals, we apply this p-probability death at each iteration of the network; after every input presented.

3. WORK

The network inputs are strings of 35 numbers, 1 or -1, to represent white and black pixels respectively on a 5 by 7 pixel-grid. When given a pixel-grid as input, the network will calculate values for the neurons in the output layer which represent some encoding of this pixel-grid. We can then measure how quickly (if at all) the network "learns" the pixel-grid inputs when consistently presented with a set of grids.

We proceed to expose the network to grids that represent the English characters a, b and c for some number of iterations. During each iteration, the network is exposed to each of the inputs in turn.

We stop after some fixed number of iterations, and verify that for each input, the network produces a consistent output over several iterations. The network has now "learned" the entire initial input set. The input set is then changed to be that of Greek characters. This new input set is then fed into the network for a number of iterations, and we examine how the value of p (our probability of Neurogenesis) affects how quickly and accurately the new set is learned.

4. RESULTS

Initially, the weights for the network were set up to simulate locality in the network space. A "nearness" value was assigned to each of the pairs, and weights were calculated using a Gaussian distribution based on how close pairs of nodes were. This didn't work particularly well. After as few as ten iterations, the nodes had reached a repeated output where all inputs were classified the same way. The Gaussian wouldn't work for this experiment.

Instead, the weights for the network are assigned random values ranging from -1 to 1. This seems to work much more accurately. After between 70 and 80 iterations, the network settles down on a repeated output patterns with a relatively high degree of accuracy.

At $p = 0$, we have a model with a zero probability of node death, or a model without any neurogenesis. This model gave a relatively good degree of accuracy for the English letters, and approximately equal accuracy for the Greek letters. It was expected that the Greek letters would fare worse than the English ones, and I suspect that there is a flaw in the underlying design which leads to this result.

For $p = 0.00004$, the value observed in macaque monkey hippocampus, we had results that were basically identical to those for $p = 0$. With only slightly

more than fifty neurons in our network, only a single neuron in 480 iterations was expected to be reset.

For $p = 0.001$ and $p = 0.005$, we had high accuracy in all of the test cases for the Greek letters. Approximately the same fairly good accuracy was observed for the English characters.

At higher values of theta ($p = 0.01$), the coded classifications for the net would not stabilize on a consistent output. Even for very large numbers of iterations (10000+), the network would still not give consistent output for the English characters.

We can see that for the theta-values within the range ($0.001 \leq p \leq 0.005$), the Greek characters are more effectively learned. However, the model seems flawed in the initial learning of the English characters. I would expect some model which learns the English characters more reliably is necessary to give a more definitive result about the usefulness of neurogenesis.

BIBLIOGRAPHY

1. D.R. Kornack and P. Rakic. Continuation of Neurogenesis in the Hippocampus of the Adult Macaque Monkey, *Proc. Natl. Acad. Sci. USA*, 96:5768-5773, 1999.

2. S. Haykin. *Neural Networks: A Comprehensive Foundation (2nd Edition)*. Prentice Hall, 1998.

Distinguishing Prosody in Speech with a Liquid State Machine

Anthony Di Franco
Yale University, Computer Science Department
New Haven, CT 06520

Abstract
The liquid state machine architecture is central to a proposed non-Turing theory of real-time computation, and its implementations give performance comparable or superior to non-realtime methods in applications where the results of a computation on time-varying inputs are immediately necessary. These characteristics among others make it a promising candidate for use as an online speech classifier. I apply the liquid state machine to determine the prosody class (one of praise, prohibition, soothing, suggesting attention, and neutral) of speech directed at a robot so that the robot might modify its behavior accordingly. In a software implementation, the classifier shows performance likely to be on par with the best methods available in recognizing the prosody class of an utterance as early as possible during the utterance. Implementations suitable for online use and requiring only modest computational resources are briefly discussed.
Keywords – prosody, feature extraction, speech processing, liquid state machines, neural networks, artificial intelligence.

1. INTRODUCTION

1.1. Background

As early as the first week of life, humans can distinguish speech by its prosodic content, that is, classify utterances according to the general intent of the speaker. Since this occurs without any kind of knowledge of language, more general features of the sound itself are thought to be the basis of this distinction. In her thesis, Breazeal (2000) uses a combination of twelve features of the spoken waveform, seven statistical functions of pitch and five of energy, to make these distinctions. She achieves 70-90% accuracy in distinguishing whether a given utterance is in either of two given prosody classes, using a generally different subset of the features to make each such pairwise distinction.

We note that Breazeal's method directly demonstrates the role of many different mathematical features of the waveform of an utterance in determining its prosody class. In 2002, Maass et. al. introduced the *liquid state machine* (Maass, 2002), an artificial neural network architecture modelled on (highly recurrent) cortical microcircuits, and placed it in a theoretical framework for general computation in real time. The general features of the liquid state machine are discussed and diagrammed below; for a detailed

discussion, (Maass, 2002) should be consulted. Among its most remarkable and appealing features is the firm separation it develops between a so-called *computational liquid* (or simply *liquid* hereafter), whose state unambiguously distinguishes time-series of inputs, and the *readout*, a map from the state of the computational liquid to the desired output. In practice, Maass and coworkers find that readouts can be trained to simultaneously compute a wide variety of arbitrarily chosen functions of input without any kind of change to the liquid.

Theoretically speaking, the liquid state machine transforms a time-varying input into a spatial firing-rate pattern. It can be rigorously shown (Maass, 2002) that this pattern distinguishes between practically any distinct inputs, and thus implicitly computes all possible functions of the input. By performing pattern recognition on the state of the liquid, one can in principle learn any function of the input so long as the function has vanishing dependence on inputs remote in time.

Most relevant to the task at hand, in (Maass, 2003) excellent results are presented for the task of distinguishing the spoken digits zero through nine from their waveforms. Maass et. al. show that the liquid state machine slightly outperforms all the networks designed to distinguish the spoken digits for a well-publicized competition (Hopfield, 2001). The liquid state machine solution used approximately 30 times fewer neurons than the network custom-designed by Hopfield and Brody for the task; moreover, it was not customized in any way for the task. The liquid state machine also performed competitively using only the 10 readout neurons of its 145 total neurons. It could cope equally well with more than one input spike per input neuron per utterance, while the limit to one such spike was a limitation in the design of the custom network. It was also trained for the more general task of distinguishing the digits at any time during the utterance, rather than just at the end, or 450 ms after the end as was required by the custom-designed architecture. This also required no customization to the liquid. Instead, the readouts were trained at intervals during the utterance instead of only at the end.

Because Breazeal's work demonstrates the role of many basic features of the waveform in the determination of prosodic classes, without suggesting that any particular classification scheme is definitive, the ability of the liquid state machine to implicitly distinguish according to any feature is ideal. Moreover, the elaborate design work, experimentation, and parameter adjustments required in Breazeal's ad-hoc classifier have no analog for a liquid state machine classifier. That is, given a training set and objective function, familiar and relatively simple methods to associate the state vector of the liquid to the desired output in the sense of supervised learning, such as linear regression or support vector machine fitting, obtain good results in practice. In effect, the liquid subsumes much of the complexity of the computation in a general way, and the remaining task – to

distinguish between liquid states and associate the desired outputs with them – is computationally and conceptually straightforward.

1.1. Problem Statement

Given a training set of recordings of utterances with varying speakers and linguistic content, which are pre-sorted by human listeners into distinct prosodic classes, the task is to use the set of training examples to develop an automatic classifier, which is to be able to correctly distinguish new examples.

We adopt the approach of training a liquid state machine, and compare its results with those of Breazeal's on the prosody task, and Maass's results on the digit classification task. Criteria are classification accuracy and suitability of possible implementations for online use in a robot.

2. NETWORK ARCHITECTURE

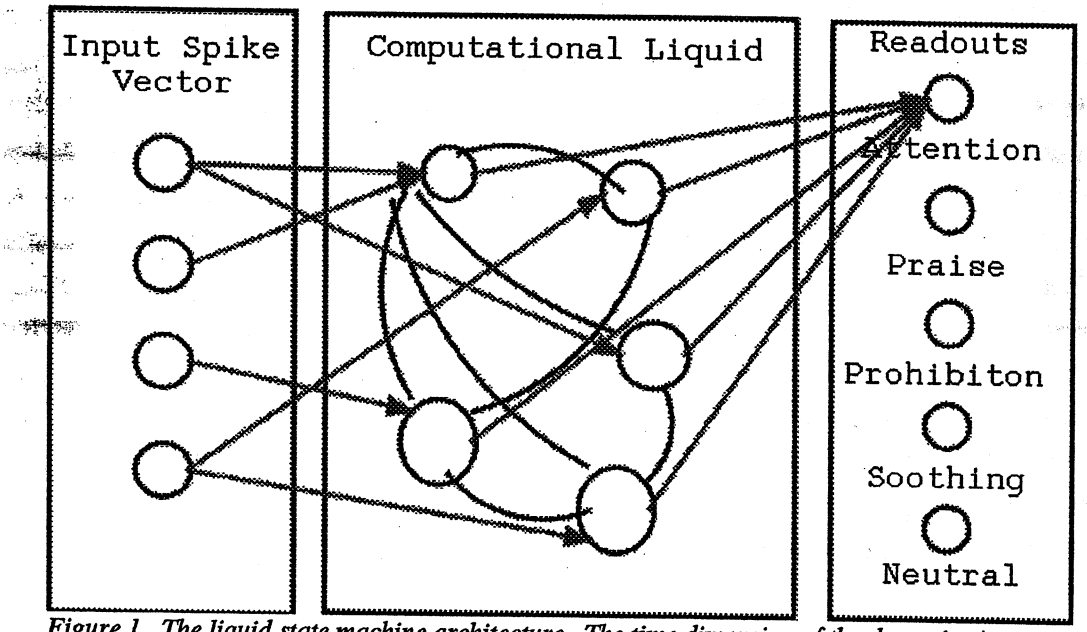


Figure 1. The liquid state machine architecture. The time dimension of the dynamics is not depicted. For each stimulus waveform, the preprocessing stage generates a set of spike times, one per input neuron. At the time so determined, the simulation generates a spike on the proper input neuron. The input neurons are randomly connected to neurons in the computational liquid, and the neurons in the computational liquid are randomly, recurrently interconnected by spiking synapses. The neurons in the computational liquid are fully connected to each of the readouts (only one set of connections is shown for clarity). A low-pass filter is applied to the spiking activity of the computational liquid to make it compatible with readouts that operate on continuous-valued domains. See text for details.

The network was constructed using a Matlab package freely available at the Neural Microcircuits Website (http://www.lsm.tugraz.at/), one specifically designed to support experiments of this type with liquid state machines. The simulation code used was based directly on the code developed using that package for the digit classification task in

(Maass, 2003). It must be noted that the package is fairly new, and as such the package's circuit simulation and learning features are largely undocumented. Moreover, the digit classification application itself was completely undocumented and coded in an ad-hoc style, and so I was unable to modify it to perform the prosody recognition task at the end of the utterance, which is most directly relevant to the task at hand. As such, the results presented below, while appearing consistent with good performance, beg further verification, perhaps with a future version of the package accompanied by adequate documentation. Currently, no other viable alternatives exist for carrying out such a simulation in full, and writing one from scratch is a significant undertaking.

The architecture used begins with a preprocessing phase to produce a set of spike times, distinctive with respect to different waveforms, that can be fed into the liquid (since it is composed of spiking neurons). Waveform files corresponding to utterances are encoded by the pre-processing phase into vectors of 40 time points corresponding to the earliest of 3 types of events in certain frequency bands, omitting some (type, band) classes that are relatively invariant in speech in general. The events are onset, peak, and offset of the energy of the waveform, and the frequency bands covered the range from 200 Hz to 5 kHz, where all the significant energy of typical speech lies.

Each of the 40 event times is then used to generate a single spike at that time on one of 40 distinct inputs to the liquid state machine. These inputs are randomly connected to the computational liquid. The 'liquid' of the liquid state machine consists of a pool of 135 spiking neurons connected randomly and recurrently among themselves, whose interconnection statistics follow data from rat cortex as in (Maass, 2003)[1]. The weights from input neurons to the liquid were also randomly determined to make the inputs distinguishable. To make the spiking activity of the pool compatible with a continuous-domain readout, a low-pass filter[2] was applied to the activity of each of the neurons in the liquid before sending their activities to the readouts. Readouts based on both linear regression on the state vector and support vector machine classification of the state vector were used.

A set of 5 readouts was used, one for each of the prosody classes. (These were one of praise, prohibition (weak scolding), soothing, suggesting attention, and neutral. Scolding was left out due to having insufficient examples.) The readouts were fully connected to the neurons in the liquid so as to have access to the complete state, and were trained to read one if the utterance was in the corresponding class, and zero otherwise, at the end of

1Parameters were identical to those used in (Maass, 2003), and were based on data from rat cortex cited there.

2The low-pass filter, besides being necessary to allow the use of linear regression and support vector machine readout, was meant to correspond with the equivalent low-pass filtering effect of transmission across a cell membrane of definite capacitance. A biologically realistic time constant of 30 ms was used.

the utterance. Training was carried out on a randomly selected subset of 300 of the 500 files (100 of each class); the remaining 200 were used to test performance.

3. RESULTS

Prosody Class	*LR Training Set Error*	*LR Test Set Error*	*LR Overall Error*	*SVM Training Set Error*	*SVM Test Set Error*	*SVM Overall Error*
Attention	0.284	0.759	0.474	0.087	0.938	0.427
Approval	0.322	2.619	1.241	0.087	1.298	0.571
Praise	0.379	1.464	0.813	0.142	1.889	0.841
Prohibition	0.340	1.650	0.864	0.155	1.015	0.499
Soothing	0.424	1.610	0.898	0.098	1.039	0.474
Overall	0.350	1.620	0.858	0.114	1.236	0.562

The error score is defined as in (Maass, 2003) as number of false positives (N_{fp}) divided by number of correct positives (N_{cp}) plus number of false negatives (N_{fn}) divided by number of correct negatives (N_{cn}), that is, $\frac{N_{fp}}{N_{cp}} + \frac{N_{fn}}{N_{cn}}$. The figures in the overall row are the means of the corresponding columns, and the figures in the overall columns are the weighted means of the corresponding rows (weighted for 300 training examples vs. 200 test examples out of 500). A score of 2 would be expected for unbiased random guessing, and a score of 0 would indicate perfect performance.

4. DISCUSSION

The results obtained were only for the 'anytime' speech recognition task described in (Maass, 2003). Paraphrased here, in the 'anytime' recognition task, a readout for a given class is trained to fire at each of a set of times spaced evenly between the beginning and end of the utterance, if this utterance is in the class it is meant to recognize, and as little as possible otherwise. This is opposed to the standard task, where there is only one set of readouts that is trained to make the distinction only at the end of the utterance, which is more relevant to the task at hand since our robot need not react in mid-utterance. On the 'anytime' task, an overall test set score of about 1.2 was obtained using support vector machine readouts. For comparison, a score of 1.4 in (Maass, 2003) on the 'anytime' digit-classification task was found to correspond with a score of 0.14 in the standard task, which is equivalent to over 90% classification accuracy, and is competitive

with Breazeal's classifier. Because the code used was undocumented, largely uncommented, large, and written in an ad-hoc style, and the original author unreachable, I was unable to successfully modify it to perform the standard task.

If we assume that a similar relationship between 'anytime' performance and standard performance exists for our prosody task, then we may expect to be able to construct a similar classifier designed to work at the end of an utterance with an error rate competitive with Breazeal's classifier, and likely superior. The main distinction to be drawn between them in the context of online use in a robot is then ease of implementation, economy of computational resources used, and online performance.

A software implementation of Breazeal's classifier requires the recording of a complete utterance before feature extraction can begin, and uses a complex multi-stage algorithm to classify based on different subsets of the determined features. It requires extreme care in design to achieve real-time performance, and is computationally expensive in any case. Constructing the feature-extraction filters in analog hardware would introduce significant additional costs in design effort and power consumption, while providing better for fast online performance, while using dedicated digital DSP hardware would alleviate some of the design cost while somewhat decreasing speed. However, the design remains dependent on having the features of the entire waveform determined before classification can begin, which places a lower bound on the online performance of the algorithm, that is, it must always provide its answer at some time after the end of the utterance.

On the other hand, the liquid state machine classifier gives its decision exactly at the end of the utterance in the worst case. As for effective implementations, though software processing is admittedly quite slow, spiking neural networks have simple, well-known, and very fast realizations in analog hardware, which is complemented by the fact that the liquid state machine architecture calls for almost no design effort – merely random connections among neurons and between pools of neurons following a particular distribution, so that the detailed layout could be determined automatically. Moreover, the theory of the liquid state machine is compatible with implementations using not only spiking neural networks, but also many other devices which are more easily implemented, such as systems of tapped delay lines. For discussion, see (Maass, 2003). Also, it is easy to envision a linear readout trivially implemented in hardware after being trained in software, or a software support vector machine could run on the output of the liquid hardware while introducing a delay on par with that of Breazeal's classifier.

5. PRELIMINARY CONCLUSION AND FUTURE WORK

Because the liquid state machine is a new architecture with only a new, largely undocumented software implementation available, only tentative results could be obtained

with existing experimental tools. However, it is also an exciting and intriguing architecture, both in theory and application, and in keeping with that, the results obtained were encouraging, and strongly suggested the possibility of a classifier combining good online performance with simple, direct, and efficient implementation.

It remains to conclusively determine the performance of the liquid state machine for the standard recognition task, and consider in detail the many options for its implementation.

ACKNOWLEDGEMENTS

I would like to thank Prashant Joshi of Wolfgang Maass' reseach group for providing in timely fashion the simulation code used in (Maass, 2003), and Thomas Natschläger, the original author or the code.

REFERENCES

1. Breazeal, Cynthia (2000). *Social Machines: Expressive Social Exchange Between Humans and Robots.* Unpublished doctoral dissertation, Massachusetts Institute of Technology.
2. Hopfield, J.J. & Brody, Carlos D. (2001). What is a moment? Transient synchrony as a collective mechanism for spatiotemporal integration. *Proceedings of the National Academy of Sciences,* v. 98(3), pp. 1282-1287.
3. Maass, W., Natschläger, T., & Markram, H. (2003). Computational models for generic cortical microcircuits. In J. Feng, editor, *Computational Neuroscience: A Comprehensive Approach,* ch. 18. CRC-Press. To appear.
4. Maass, W., Natschläger, T., & Markram, H. (2002). Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, v. 14(11). pp. 2531-2560.

A Neural Network Implementation of the Rescorla-Wagner Model

Byron Igoe
Yale University, Department of Computer Science
New Haven, CT 06520

Abstract
The Rescorla-Wagner model of learning in operant conditioning is the current paradigm. By constructing a neural-network simulation of a lab rat in an operant chamber, we can test the validity of this, and other models. It has already been hypothesized, by Rescorla himself, that the model is incomplete. Using the simulation to test the validity of new models is quicker, easier, and more reliable than using real rats.

Keywords – operant conditioning, Rescorla-Wagner, lab rat, reinforcement learning, neural networks.

1. INTRODUCTION

The type of learning described by the Rescorla-Wagner model (1972) is conditioning. When two events are paired proximally in time, they become associated with each other in the brain. For example, we learn to expect thunder after we see lightning. The first of two events is the conditioned stimulus (CS). The second of the two is either a conditioned response (CR) or an unconditioned response (UR). UR usually refers to behavior that is instinctive, such as food-seeking or pain-avoidance. The Rescorla-Wagner model attempts to describe the rate at which the association between stimuli and responses takes place. Additionally, it accurately predicts certain properties of learning, such as blocking and conditioned inhibition.

Learning has long been studied by psychologists using rats. To study conditioning, they place these rats inside an operant chamber. The chamber has two stimuli, a light and a speaker that emits a tone. The chamber also has a bar that the rat can press. The first training that must occur is associating a bar press (CR) with receipt of food (UR). Thereafter, rats can be trained to correctly learn any of a number of complicated rules.

2. NEURAL NETWORK MODEL: LLABRAT

Any neural network model has inputs, an output, and a learning rule. The neural-network implementation of the Rescorla-Wagner model is called LLABRAT (the first L is for Linux, the operating system on which it was developed). The inputs to LLABRAT are the stimuli present in the particular trial in the operant chamber. The output is the contingent bar press. The learning rule used by the neural net is adapted from the Rescorla-Wagner model of learning.

The number of repetitions, or trials, is logarithmically proportional to the change in associative strength of the relevant stimuli. In LLABRAT, you choose the number of trials to run. The first step in a trial is to generate random stimuli. The "rat" will press

the bar contingent upon the strength of the associations between the stimulus, and receipt of food in the past. That is, the rat will learn that pressing the bar after the presentation of certain stimuli is futile.

If the rat presses the bar, LLABRAT decides whether to provide the rat with food. Food is the reinforcement that stimulates learning. If the rat is given food, then the strengths of the present stimuli are increased. If the rat is not given food, then the strengths of the present stimuli are decreased. The increase is an asymptotic one that approaches a maximum. The decrease is proportional to the strength of the input. This change in strength is described by the following equation:

$$\Delta V = \alpha(\lambda - \Sigma V)$$

Here ΔV is the change in associative strength of a stimulus, α is the salience of that stimulus (Salience is pre-determined. For example, rats respond more to light than sound due to their evolution.), λ is either 1 or 0 (1 signifying an increase, and 0 a decrease), and ΣV is the sum of the strengths of all inputs present in the trial. Salience is a measure of how important an input is, relative to the others.

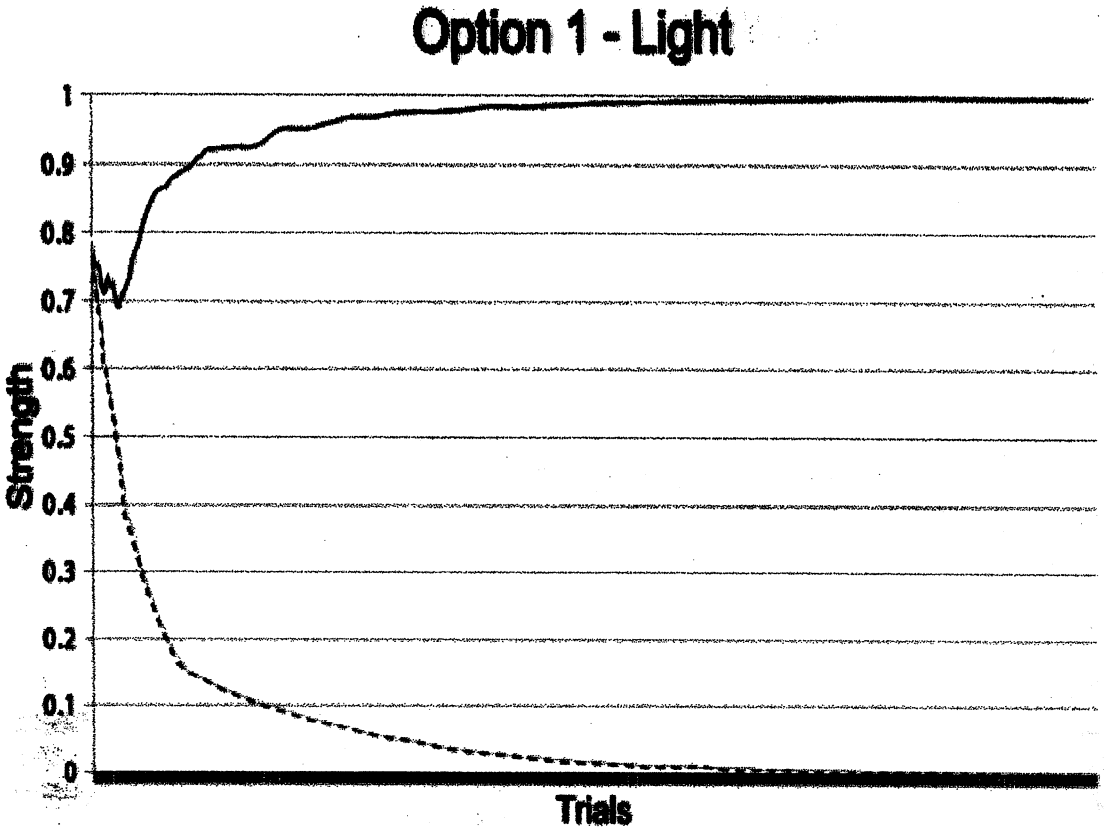
There are many learning scenarios built into LLABRAT that you can choose among. The range of "tricks" shows the power of the Rescorla-Wagner model. The rat can be trained to respond just to one stimulus, ignoring the other. The rat can learn to respond only when both stimuli are present. Most interestingly, the rat can learn to press the bar when one stimulus is present in the absence of the other (conditioned inhibition).

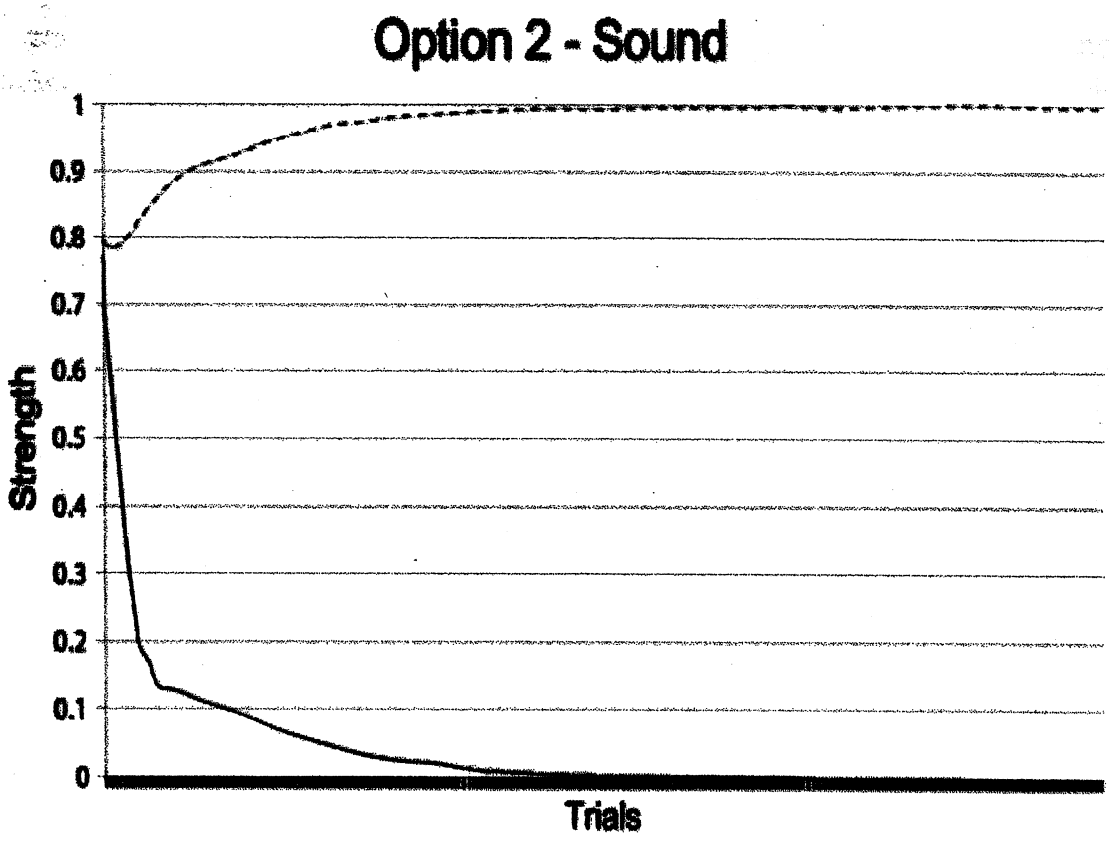
Unfortunately, much like the Perceptron, the Rescorla-Wagner model cannot explain the exclusive-or operation. Rats, however, can learn XOR. Additionally, Rescorla's recent work (2000) points to another fault with the current system. The change in associative strengths of stimuli with different salience depends on their status as either an exciter or inhibitor.

Knowing these shortcomings of the Rescorla-Wagner model, we can use LLABRAT to help develop a more complete equation. It is easier to program tentative learning rules into LLABRAT and run hundreds of trials, than to go through the process of training hundreds of real rats. Another way in which the simulation is better is the repeatability of results. If a certain sequence of stimulus presentations causes a problem, you can re-simulate that exact sequence many times as you adjust the learning rule to handle it.

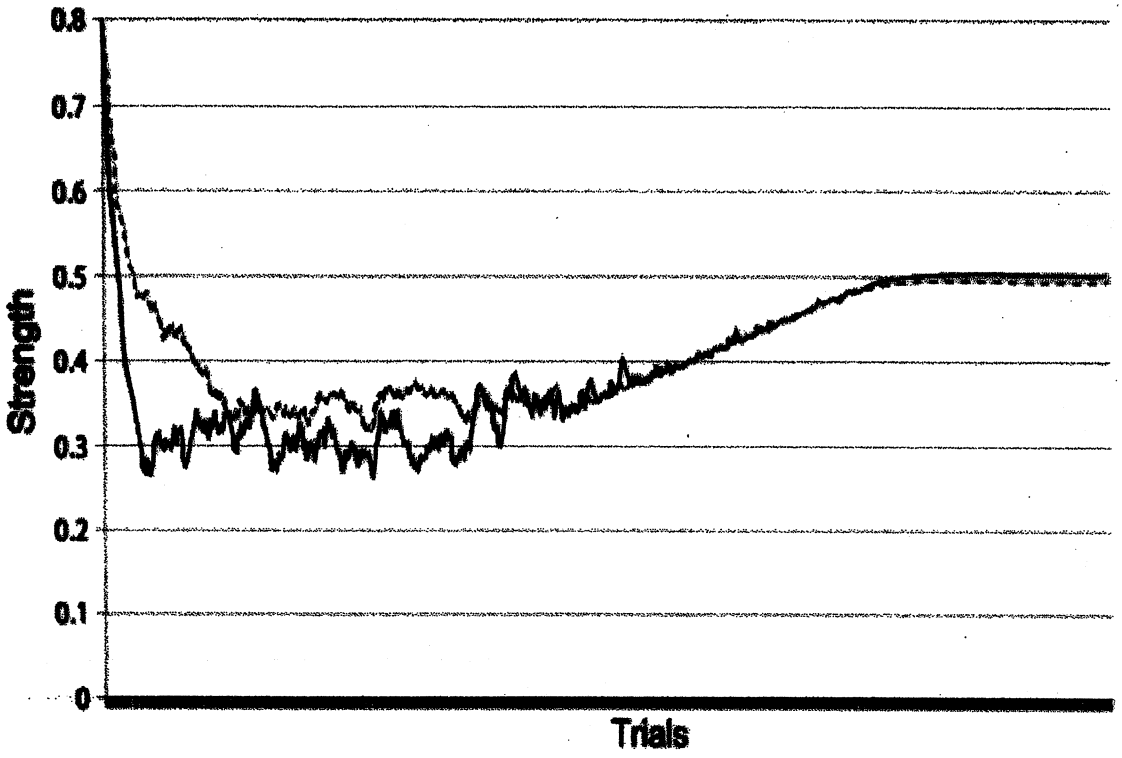
3. RESULTS

The graphs that follow show associative strength of the two stimuli as they change over hundreds of trials. The solid line represents the associative strength of the Light stimulus, and the dashed line represents the associative strength of the Sound stimulus. In Options 1 and 2, one stimulus is ignored (the associative strength drops to 0). In Option 3, neither stimulus is strong enough to cause a response when presented alone. When they are presented together, however, they add up, causing the rat to bar press. In Options 4 and 5, one stimulus is driven negative, i.e. to be an inhibitor. When presented together, the inhibitor will subtract from the strength of the exciter, suppressing a response on the part of the rat.

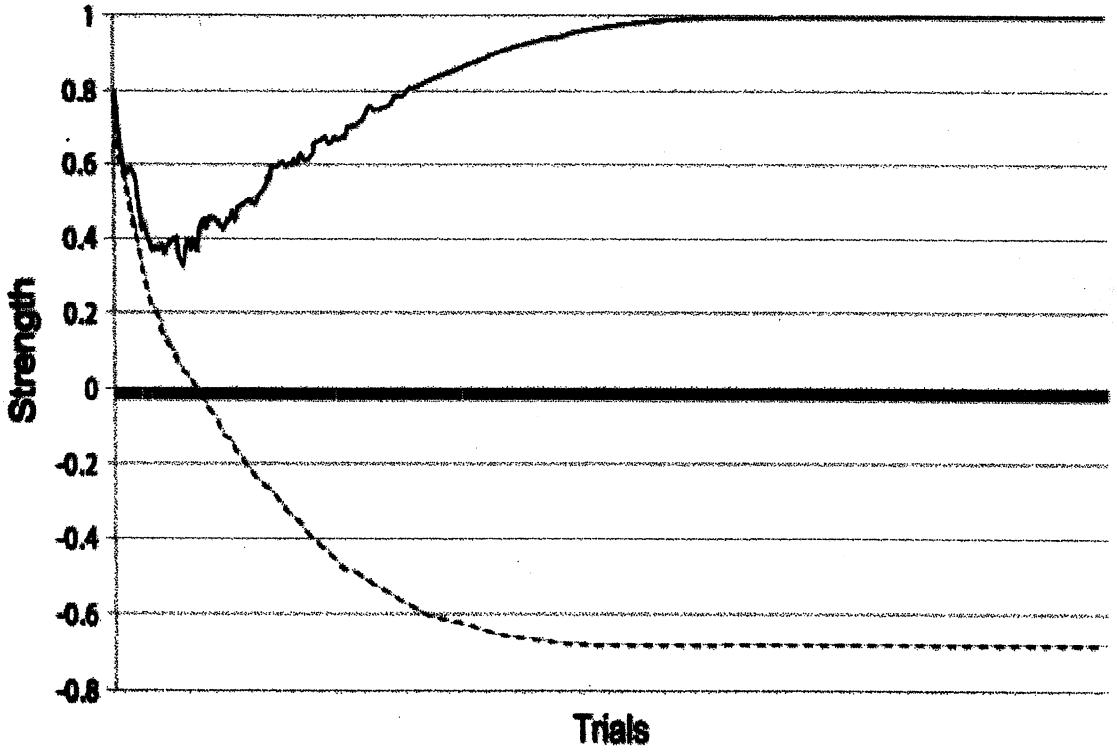

Option 1 - Light
Strength
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Trials

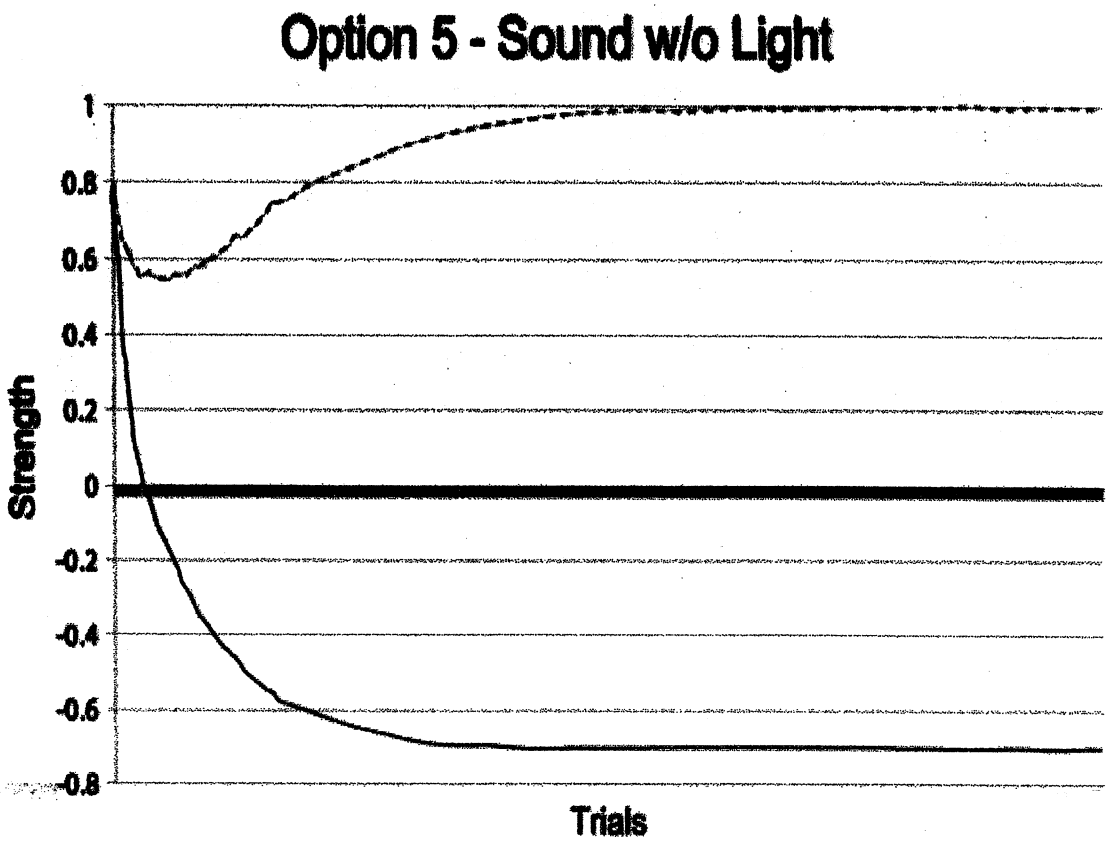

Option 2 - Sound
Strength
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Trials

Option 3 - Light and Sound



Option 4 - Light w/o Sound





4. DISCUSSION

Many rats have been trained to do the five discrimination tasks in LLABRAT by students in introductory conditioning and learning courses. The way rats will behave over many trials of each of these tasks is predictable within a margin of error. The Rescorla-Wagner model was created to attempt to predict other behavior before real rats were trained on those new tasks.

The learning rule currently used in LLABRAT successfully "trains the rat" on all tasks except for those of exclusive-or and the anomaly discovered by Rescorla (2000). The Rescorla-Wagner model does not account for XOR or certain properties of joint stimuli with different associative strengths. The model must be modified to account for these new features, but it is a good start. A possible change would include higher order terms.

5. CONCLUSION

The best way to test hypotheses of learning rules is by using an artificial neural network model like LLABRAT. Once the model causes the simulation to accurately imitate a real rat in all dimensions that we've studied, we need to develop more sophisticated tests. Training an actual rat takes much more time, and introduces many more possibilities for error.

REFERENCES

1. Rescorla, R. A. (2000). Associative changes of excitors and inhibitors differ when they are conditioned in compound. *Journal of Experimental Psychology: Animal Behavior Processes*, 26, 428-438.
2. Rescorla, R. A. & Wagner, A. R. (1972). A theory of Pavlovian conditioning: Variations in the effectiveness of reinforcement and nonreinforcement. *Classical Conditioning II: Current Theory and Research*, 64-99. New York: Appleton-Century-Crofts.
3. Wagner, A. R. & Rescorla, R. A. (1972). Inhibition in Pavlovian Conditioning: Application of a theory. *Inhibition and Learning*, 301-336. London: Academic Press.

Lexical Memory: Identification of Verb Forms

José R. Rivera
Department of Computer Science, Yale University
New Haven, CT 06520

Abstract

Our understanding of human language has led to various attempts to reproduce aspects of human linguistic competence. Most of these models, however, are implemented using traditional computational systems. The human brain is a dynamic and complexly parallel system. Any work, then, that would model a human cognitive function would benefit from a similarly parallel system. Thus a neural network architecture seems an ideal choice for implementing such a model.

The human linguistic system seems to implement a lexical memory for the retrieval of semantic information, e.g., meaning, associated with a particular word. Any system purporting to interpret language must be able to reach this semantic data from any of the various morphological forms in which a particular word can manifest itself. Because the system can extrapolate what the various forms of a word that it has never encountered might be, it is conjectured that the lexical memory references an abstraction of the differential parts of a word when searching for data.

The following work implements a Syntactic-Lexical Memory System using neural networks that, given a language, models this behavior.

Keywords – neural networks, lexical memory, self-organizing map, feed-forward network, back-propagation, Hopfield network

1. INTRODUCTION

1.1 Linguistic Considerations

The human linguistic system supports phonological, syntactic, and semantic processes (Levelt, et al. 1999). Lexical access research has shown that phonological processing must occur prior to syntactic processing, which must, in turn, occur prior to semantic

processing (Levelt, et al. 1999; Canseco-Gonzalez, et al. unpubl.). The semantic processes are believed to interact with a lexical memory that stores the semantic information associated with a particular word (O'Grady et al. 1997). Syllabification of the English pronunciation of the phrase *escort us*, e.g., yields *e-scor-tus*[1]; thus, before any syntactic or semantic manipulation of the phrase occurs, the phonological process must yield *escort* and *us* to the syntactic (Levelt, et al. 1999); furthermore, some syntactic process must compute the representation expected by the semantic processes to fetch a word's semantic data for any given form, e.g., *escorted* (Canseco-Gonzalez, et al. 1998).

In authoring their lexicons, Eblaite scholars of the 3rd millennium and Akkadian scholars of the 2nd, referenced verbs by their infinitives (Hallo 2004). As Eblaite and Akkadian are among the earliest languages[2], it seems reasonable to select the infinitive as the basis for lexical organization of verbs in this work. We recognize that the actual lexical organization within the brain may not use such a representation, though barring further insight into the representation we present it as a possibility.

1.2 Requirements of the Syntactic-Lexical Memory System

The proposed system means to identify the lexical entry corresponding to a given verb form. Retrievable lexical entries will exist in an implemented memory (in emulation of the lexical memory described in §1.1). Since lexical access studies suggest that the human brain retrieves the lexical data of verbs differently than that of nouns (Caramazza and Hillis 1991; Damasio and Tranel 1993; Silveri et al. 2003), there exists a precedent to treat both forms separately, which justifies the system's specialization on verb retrieval.[3]

This system will not attempt to perform any processes prior to the syntactic (e.g., phonological processes). As a result, the verb forms presented to this system will correspond to the representation of the word once it is ready for syntactic manipulation. Since the system only implements syntactic and lexical processes, it is hereafter referred to as the syntactic-lexical memory system (*S-LM*).

The *S-LM* must function as a universal system, i.e., just as native speakers of English or of Arabic can successfully retrieve the infinitive of a given verb form for their respective languages, so should the *S-LM* behave for its native language[4].

1.3 Language Choice

To satisfy, by inference, the imposed universality constraint (§1.2), the language chosen for this instance of the *S-LM* must present sufficient morphological alterations to the infinitive of a given verb that none of the set of forms considered for a given verb are

---

[1] This example is taken from Levelt, et al. (1999).
[2] Sumerian and Chinese are the earliest documented written languages dating to the early 3rd millennium (Insler 2004).
[3] Verbs generally have more distinct forms than nouns.
[4] The native language of the system represents the language chosen for its instantiation.

trivially[5] related to the infinitive. The language must also provide enough verb forms to demonstrate the validity of the system over the many verb forms that different languages may present.

Akkadian[6] seems a suitable candidate for this exercise. Not only does each Akkadian verb have approximately 370 verbal forms (there are nominal and adjectival forms for each verb as well), the morphological alterations to the infinitive of the verb are significant. The G-Stem[7] Preterite form of parāsum is presented as an example:

3rd Sing C:	iprus	3rd Plur M:	iprusū
2nd Sing M:	taprus	3rd Plur F:	iprusā
2nd Sing F:	taprusī	2nd Plur C:	taprusā
1st Sing C:	aprus	1st Plur C:	niprus

In addition to this morphological complexity, Akkadian presents morphological alterations that result from phonological processes. As per the discussion in §1.2 these alterations fall outside the scope of this work.

As mentioned above, we will consider only those forms which are not trivially related to the infinitive. Consequently, we chose to focus on the Present, Preterite, and Perfect tenses which account for 192 forms of the Akkadian verb. Of these 192 forms, twelve 1st person singular forms are indistinguishable from their corresponding 3rd person singular form; this brings the total to 180 distinct forms.

2. SYSTEM DESIGN

2.1 Architecture

The *S-LM* is implemented using three neural networks: a self-organizing map, responsible for performing syntactic classification, a feed-forward network, which performs syntactic manipulations (discussed in §2.4), and a Hopfield network acting as the lexical memory.

[5] Triviality, in this context, refers to verbs for which the greater part of the infinitive is a substring of the form in question; many English verb forms are all trivially related to their infinitive in that the infinitive is a substring of any such forms.

[6] Akkadian is a Semitic language that was spoken in the Middle East from around 2500 BCE to 0 CE.

[7] Akkadian has 4 main stems (G, D, Š, N) and 2 additional derived stems (-t, -tn) for each of those, except the N for which only the –tn seems to have existed (Caplice 1988). These stems all have the present, preterite, and perfect tense, among others.

A diagram of the Syntactic-Lexical Memory System follows:

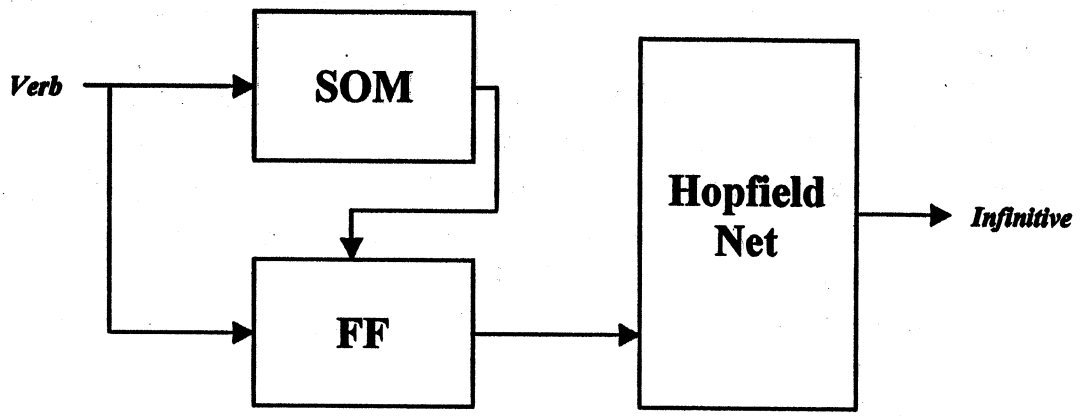


Figure 2-1: *Syntactic-Lexical Memory* System Architecture

2.2 Inputs

Normalized Akkadian[8] has 32 possible characters:

ʾ a ā â b d e ē ê g ḫ i ī î k l m n p q r s ṣ š t ṭ u ū û w j z

Accordingly, the input sets selected assign a numeric representation to each letter in a word that corresponds to its ordinality in the above list.

Each letter is represented by a 6-bit vector (the extra bit is needed to represent the *null* value discussed below) where the bits of the vector are set according to the binary representation of the ordinality for the letter being represented. The letter *a* is thus represented: 000001.

Concatenation of the appropriate vector representations for letters in a word yields the vector representation of that word.

iprus: [001011 010010 010100 010000 001011][9]

For the network to function, all the inputs must be the same length; thus, a *null* value was added to the list of available characters. These *null* values are inserted into the representation of the verb forms as appropriate. As the 2nd Person Plural (Masculine and Feminine) Štn-Stem Present form, is the longest Akkadian verb form that this instantiation of the *S-LM* will handle (it has 12 letters), and each letter is represented by a 6-bit sequence, the resulting input is a 72-bit vector.

[8] Normalized Akkadian refers to a Romanized representation of cuneiform Akkadian text.
[9] The spacing is used only to illustrate the letter boundaries and does not appear in the implementation

The *null* value used was 100000 as the Hamming distance from this point to the representation of the ', the least used letter in Akkadian, is the shortest[10]. The representation of iprus is provided as an example:

iprus: [100000 001011 010010 010100 010000 001011 100000 ... 100000]

2.3 Syntactic Classification

We divided the syntactic processing into two portions: syntactic classification and syntactic transformation. A self-organizing map (SOM) implements the syntactic classifier. This network receives the exogenous input.

The SOM defines one neuron per tense in each stem (derived or not). Since a developed human linguistic system, cannot, without training, process syntactic information of unknown languages, this does not compromise the universality of the system.

Defining a neuron for each tense highlights three features of any Akkadian verb form: the stem class (G, D, Š, and N), the stem (root stem, -n, -tn), and the tense (Present, Preterite, Perfect). Corresponding to these three features a 3-dimensional hexagonal distribution of neurons was used to instantiate the SOM. A hexagonal distribution was chosen to take advantage of the regularity of a traditional Kohonen map while exploiting the sparser topography that the hexagonal distribution yields. We hoped that the topography would help the system make more accurate classifications.

The network was composed of 36 neurons (4 main stems by 3 stems per main stem by 3 tenses[11]). It was trained on all 192 verb forms for 20 verbs, and was tested on a total of 40 verbs totaling 7680 words.

The output from the network, the identifier of a neuron in the map, corresponds to the network's classification of the inputted verb form. As one neuron was defined for each tense in each stem, we hypothesized that the classification would yield one neuron per tense, regardless of person or number. This hypothesis does not consider the effect of the additional neurons in the network, though we hoped they would serve to refine classification within the various tenses (person and number).

2.4 Syntactic Transformation

Once a word has been classified, the same word and the output from the classifier are fed to the feed-forward network. This network provides the appropriate syntactic manipulations that transform the input word to its infinitival form (or the deterministic components of the infinitive) for retrieval from our lexical memory.

---

[10] The Hamming distance from ' [000000] to *null* [100000] is one bit. As none of the letter representations defined use the highest order bit, the ' presents the shortest Hamming distance to the *null* value.

[11] Unfortunately, this architecture defines a number of neurons that, given the hypothesized results, would not be used.

Knowledge of the language in question is necessary for this process, i.e., humans are only able to identify verb forms for languages that they know, implying a body of knowledge that is acquired during language learning. This knowledge is represented in the feed-forward network's training. Thus the exemplars against which this network is trained must be chosen such that they deterministically identify the verb. As the deterministic components for Akkadian words are its consonants, these are used as exemplars. Because there is only one lexical entry for all the forms of a verb (see §2.5), the target of any form is a representation of its infinitive, regardless of the output from the syntactic classifier. The target of any form of parāsum, e.g., is p_r_s__.

The initial design of the network is illustrated in Figure 2-2:

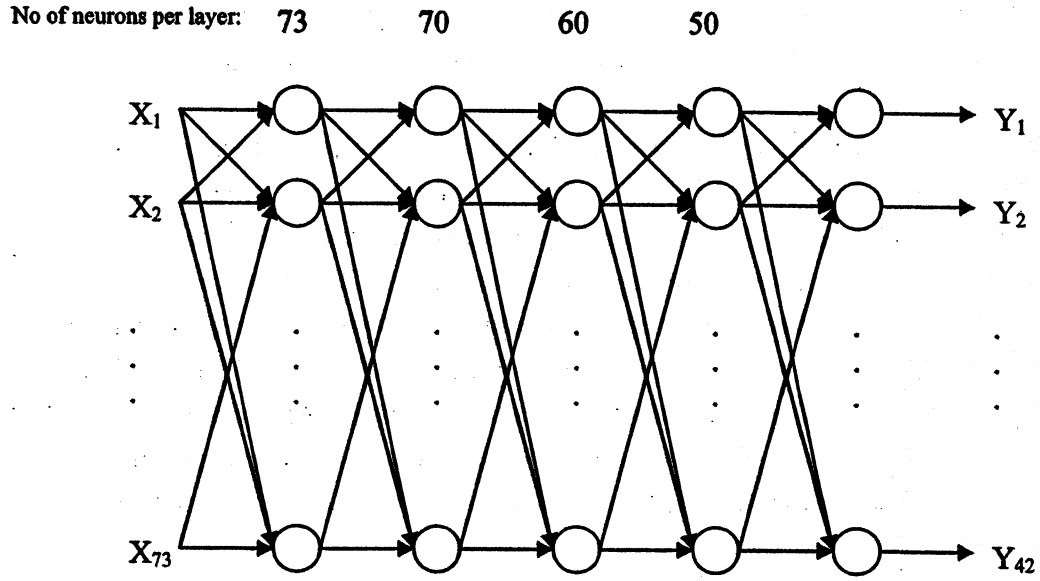


Figure 2-2: Design of feed-forward network used to implement the syntactic transformer.

This feed-forward network takes 73 inputs: the 72 bits representing the verb form in question, and the identifying integer output from the syntactic classifier. As the example of parāsum illustrates, the output of the feed-forward network is a 7 letter representation, i.e., a 42-bit vector, similar to the infinitive. The network was conceptualized as a five layer feed-forward network. Since staging the processing through at least five layers would provide more computational power than otherwise, we believed that the system would perform more accurately than with less.

2.5 Lexical Memory

A Hopfield network serves as the lexical memory for the *S-LM*. As this exercise focuses on the retrieval of a lexical entry given a particular verb form, the memory only contains

the infinitive of each verb. We assume tha, once the lexical entry can be retrieved any data stored with it is also available, thus the addition of such data to the memory is extraneous for the purposes of this implementation.

The spurious states that Hopfield networks can produce are actually a desired feature for the memory, because these states might include accurate generalizations of infinitival verb forms that are not stored in the memory, such as an Akkadian non-word[12]. Just as humans can identify a nonexistent verb by its form, so might the *SL-M* produce similar generalizations.

As introduced in §1.1, this Lexical Memory stores a 42-bit representation of the infinitive forms of those verbs that it has learned. Given such a representation, the Hopfield network should yield the same 42-bit representation. It should also correct any corrupt versions of the infinitive. Thus the input that the Lexical Memory receives from the Syntactic Transformer is such a corrupt value, e.g., p_r_s__. This input set was chosen to help simplify the work of the Syntactic Transformer. Rather than producing an exact replication of the infinitive, the feed-forward network can disregard specific values where the *null* values are inserted.

3. RESULTS

3.1 Syntactic Classification

Reliable classification of various verb forms was possible. In this analysis reliability refers to the extent to which a particular verb form was identified by one particular classifier. It does not claim that the classification was unique to that form; in fact, the number of neurons (36) cannot uniquely classify all the unique verb forms (180). From the data compiled, the mode was selected as the neuron to which a particular form was mapped.

A listing of modes for each verb form can be found in §A.2; a similar listing of modes mapped to the percent reliability of each classification follows the mode listing in §B.1. Further reliability data can be found in §B.2. As Figure B-5 (b) presents, reliable classification occurred 84.91% of the time. §C.1 illustrates the percentage distribution of classification among the various verb forms. Any further analysis uses the mode as the referential point for each verb form.

Contrary to the hypothesized results, the Syntactic Classifier did not produce a mapping that would identify tense uniquely[13]. The mapping did, however, distinguish dependably among person.

---

[12] An Akkadian non-word refers to a word that, given the structure of the lexicon, is possible, but nonexistent, e.g. the English non-word frub.

[13] It did uniquely identify the G-Stem Preterite Tense.

The anomalous classification of the G-Stem Preterite tense (see §A.2), poses some analytical problems. Because this anomaly does not generate any useful analysis, we have omitted it from the rest of this discussion and from any calculations therein unless otherwise specified. Since each tense has eight verb forms, the total number of verb forms thus considered is 184. Moreover, since the 1st Person Singular form in all considered tenses of the D and Š-Stem are identical to that of the 3rd Person Singular of their respective stems, they will be considered, not as 1st Person forms, but 3rd Person forms.

In classifying the verb forms by person, the network produced some collisions. When such was the case, the recurrence of the classification in other forms of the same person was identified, and the person with the least number of forms identified by that particular classification was taken to be in error. Figure D-1 shows the collisions that occurred. As the data recorded in §A.2 demonstrates, all of the errors, except for two distinct collisions with all three persons that have yet to be discussed, were 1st Person misclassifications. The error of the collisions just mentioned was attributed to the 2nd Person and 1st Person as per the criterion outlined above.

With these considerations in mind, the network classified 84.55% of the verbs forms correctly by person. This low percentage results from the poor classification of 1st Person forms (29.41% correct). 3rd Person classification was correct 98.77% of the time, and 2nd Person classification: 98.55% (see Figure D-2 (b)).

Revisiting the G-Stem Preterite classification, we can conclude that, given the length of its forms (the Preterite forms of this particular stem has the shortest words that the *S-LM* encountered), the classifier gave more weight to the *null* values of the words in this tense than to the letters. Indeed, all of the words for this tense, except the form 2nd Person Plural form which only had five, had at least six *null* values (half of their length).

3.2 Syntactic Transformation

Preliminary testing on a single verb within a single tense of a single stem yielded promising results, but when the range of values the network needed to handle increased memory limitations barred progress.

The initial training set consisted of 20 verbs in all the forms tested. This totaled 3840 words, which, given their 72-bit representation, yielded a 34.56 KB (276,480-bit) training set. The memory limitations imposed by the network simulator proved too stringent for the size of the training set and the network.

To attempt to overcome the memory limitations we implemented 2 measures. First, we reduced the number of verbs from 20 to 10, producing a 17.28 KB (138,240-bit) training set. Second, we simplified the networks architecture moving from a 5 layer feed-forward network to a 2 layer network. Unfortunately neither was successful at surmounting the memory limitations.

Since further attenuating the number of verbs would produce too few exemplars for each tense from which the network could make accurate generalizations, further work on this stage was halted.

3.3 Lexical Memory

As expected, the Hopfield network was able to retrieve any values with which it had been trained; it also retrieved all of the verbs it had learned when given an input identical to what the Syntactic Transformer would output. None of the spurious states found corresponded to an infinitive that we explicitly requested but was not in the memory.

4. CONCLUSIONS AND BEYOND

4.1 Syntactic Classification

Given the classifier's demonstrated ability to distinguish verbs by number, further work towards a complete classification layer seems warranted. Important considerations for such work include close attention to the input definition, possibly redefining some of the letter representations. Codes guaranteeing a specific Hamming distance, making collisions less probable, may yield more accurate unique classifications of verb forms.

4.2 Syntactic Transformation

Compression of the input vectors, possibly using LZW, or another such algorithm, should sufficiently decrease the size of the input vectors to make work on this stage feasible. The compression most likely to help would involve only whole letters (either alone, or in a group) since compressing sequences of bits that represent portions of letters, could yield two representations of the differential parts for a particular verb form that differ too drastically for the syntactic classifier to identify reliably or for the transformer to draw accurate generalizations. As the previous statement suggests, the compressed values should be tested with the syntactic classifier in hopes of achieving better overall performance.

4.3 Lexical Memory

The lexical memory yielded adequate results, dependably performing the primary task assigned to it. Manipulations of the input sets as described in §4.1 and §4.2, should be tested on this portion of the system as well, as they might yield the generalizations hypothesized at the on-set of this experiment.

5. REFERENCES

Caplice, Richard (1988). Introduction to Akkadian, 3rd rev. ed. Rome: Biblical Institute Press.

Caramazza, Alfonso, and Argye E. Hillis (1991). *Lexical organization of nouns and verbs in the brain.* Nature, v. 349, 788-790.

Canseco-Gonzalez, et al. (1998). *Processing of grammatical information in Jabberwocky sentences.* Unpublished

Damasio, Antonio R., and Daniel Tranel (1993). *Nouns and verbs are retrieved with differently distributed neural systems.* Proceedings of the National Academy of Science, USA, v. 90, pp 4957-4960.

Haykin, Simon (1999). Neural Networks: A Comprehensive Foundation. Upper Saddle River: Prentice Hall.

Hallo, William W. (2004). E-mail to the author. 17 January.

Insler, Stanley (2004). Historical Linguistics Introductory Lecture. Yale University, New Haven. 12 January.

O'Grady, William, et al. (1997). *Syntax: The analysis of Sentence Structure.* Contemporary Linguistics: An Introduction, ch. 5, pp 163-221

Silveri, Maria C., et al (2003). *Grammatical class effects in brain-damaged patients: Functional locus of noun and verb deficit.* Brain and Language, v. 85, pp. 49-66.

6. ACKNOWLEDGEMENTS

Many thanks to Professor Willard Miranker for his supervision of this project and guidance throughout, to Professor William W. Hallo, for his support, Assyriological expertise and tutelage, and to Elizabeth Payne for further help with Akkadian.

The font used to notate Akkadian normalizations, CuniTTGoe, was designed by Dominique Charpin.

APPENDIX A

A.1 Verb Forms: Parāsum

The verb parāsum is provided as an illustration of the various verb forms used.

G

Present Tense		Preterite Tense		Perfect Tense	
iparras	iparrasū	iprus	iprusū	iptaras	iptarsū
taparras	iparrasā	taprus	iprusā	taptaras	iptarsā
taparrasī	taparrasā	taprusī	taprusā	taptarsī	taptarsā
aparras	niparras	aprus	niprus	aptaras	niptaras

Gt

Present Tense		Preterite Tense		Perfect Tense	
iptarras	iptarrasū	-	-	iptatras	iptatrasū
taptarras	iptarrasā	-	-	taptatras	iptatrasā
taptarrasī	taptarrasā	-	-	taptatrasī	taptatrasā
aptarras	niptarras	-	-	aptatras	niptatras

Gtn

Present Tense		Preterite Tense		Perfect Tense	
iptanarras	iptanarrasū	-	-	iptatanras	iptatanrasū
taptanarras	iptanarrasā	-	-	taptatanras	iptatanrasā
taptanarrasī	taptanarrasā	-	-	taptatanrasī	taptatanrasā
aptanarras	niptanarras	-	-	aptatanras	niptatanras

D

Present Tense		Preterite Tense		Perfect Tense	
uparras	uparrasū	uparris	uparrisū	uptarris	uptarrisū
tuparras	uparrasā	tuparris	uparrisā	tuptarris	uptarrisā
tuparrasī	tuparraṣā	tuparrisī	tuparrisā	tuptarrisī	tuptarrisā
uparras	nuparras	uparris	nuparris	uptarris	nuptarris

Dt

Present Tense		Preterite Tense		Perfect Tense	
uptarras	uptarrasū	-	-	uptatarris	uptatarrisū
tuptarras	uptarrasā	-	-	tuptatarris	uptatarrisā
tuptarrasī	tuptarrasā	-	-	tuptatarrisī	tuptatarrisā
uptarras	nuptarras	-	-	uptatarris	nuptatarris

Dtn

Present Tense		Preterite Tense		Perfect Tense	
uptanarras	uptanarrasū	-	-	-	-
tuptanarras	uptanarrasā	-	-	-	-
tuptanarrasī	tuptanarrasā	-	-	-	-
uptanarras	nuptanarras	-	-	-	-

Figure A-1: The G, Gt, Gtn, D, Dt, and Dtn stems for parāsum. The dashes (-) stand in for forms that are identical to one previously listed, e.g., the G-Stem Perfect and Gt-Stem Preterite forms are identical.

Š

Present Tense		Preterite Tense		Perfect Tense	
ušapras	ušaprasū	ušapris	ušaprisū	uštapris	uštaprisū
tušapras	ušaprasā	tušapris	ušaprisā	tuštapris	uštaprisā
tušaprasī	tušaprasā	tušaprisī	tušaprisā	tuštaprisī	tuštaprisā
ušapras	nušapras	ušapris	nušapris	uštapris	nuštapris

Št

Present Tense		Preterite Tense		Perfect Tense	
uštapras	uštaprasū	-	-	uštatapris	uštataprisū
tuštapras	uštaprasā	-	-	tuštatapris	uštataprisā
tuštaprasī	tuštaprasā	-	-	tuštataprisī	tuštataprisā
uštapras	nuštapras	-	-	uštatapris	nuštatapris

Štn

Present Tense		Preterite Tense		Perfect Tense	
uštanapras	uštanaprasū	-	-	-	-
tuštanapras	uštanaprasā	-	-	-	-
tuštanaprasī	tuštanaprasā	-	-	-	-
uštanapras	nuštanapras	-	-	-	-

N

Present Tense		Preterite Tense		Perfect Tense	
inparras	inparrasū	inparis	inparsū	intapras	intaprsū
tanparras	inparrasā	tanparis	inparsā	tantapras	intaprsā
tanparrasī	tanparrasā	tanparsī	tanparsā	tantaprasī	tantaprasā
anparras	ninparras	anparis	ninparis	antapras	nintapras

Ntn

Present Tense		Preterite Tense		Perfect Tense	
ittanapras	ittanaprasū	-	-	ittatapras	ittataprasū
tattanapras	ittanaprasā	-	-	tattatapras	ittataprasā
tattanaprasī	tattanaprasā	-	-	tattataprasī	tattataprasā
attanapras	nittanapras	-	-	attatapras	nittatapras

Figure A-2: The Š, Št, Štn, N and Ntn stems for parāsum. The dashes (-) stand in for forms that are identical to one previously listed, e.g., the G-Stem Perfect and Gt-Stem Preterite forms are identical.

A.2 Mode for Verb Classification

The mode of the classification data was adopted as the identifier for a particular verb form.

G

Present Tense		Preterite Tense		Perfect Tense	
23	36	18	18	6	3
11	36	18	18	14	3
11	11	18	18	14	14
11	11	18	18	6	6

Gt

Present Tense		Preterite Tense		Perfect Tense	
4	4	-	-	4	4
27	4	-	-	27	4
27	27	-	-	27	27
3	15	-	-	3	15

Gtn

Present Tense		Preterite Tense		Perfect Tense	
1	1	-	-	5	9
25	1	-	-	25	9
25	25	-	-	25	25
13	29	-	-	13	29

D

Present Tense		Preterite Tense		Perfect Tense	
24	36	24	36	16	16
12	36	12	36	28	16
12	12	12	12	28	28
24	12	24	12	16	28

Dt

Present Tense		Preterite Tense		Perfect Tense	
32	16	-	-	21	9
28	16	-	-	33	21
28	28	-	-	33	33
32	28	-	-	21	33

Dtn

Present Tense		Preterite Tense		Perfect Tense	
1	1	-	-	-	-
34	1	-	-	-	-
34	34	-	-	-	-
1	34	-	-	-	-

Figure A-3: The G, Gt, Gtn, D, Dt, and Dtn stems for parāsum. The dashes (-) stand in for forms that are identical to one previously listed, e.g., the G-Stem Perfect and Gt-Stem Preterite forms are identical.

Š

Present Tense		Preterite Tense		Perfect Tense	
24	36	24	36	32	16
12	36	12	36	28	16
12	12	12	12	28	28
24	12	24	12	32	28

Št

Present Tense		Preterite Tense		Perfect Tense	
32	16	-	-	21	21
28	16	-	-	33	21
28	28	-	-	33	33
32	28	-	-	21	33

Štn

Present Tense		Preterite Tense		Perfect Tense	
1	1	-	-	-	-
34	1	-	-	-	-
34	34	-	-	-	-
1	34	-	-	-	-

N

Present Tense		Preterite Tense		Perfect Tense	
3	4	6	6	3	3
14	4	6	6	27	3
27	27	14	14	27	27
3	3	6	6	3	3

Ntn

Present Tense		Preterite Tense		Perfect Tense	
5	1	-	-	5	9
25	1	-	-	25	9
25	25	-	-	25	25
13	13	-	-	13	13

Figure A-4: The Š, Št, Štn, N and Ntn stems for parāsum. The dashes (-) stand in for forms that are identical to one previously listed, e.g., the G-Stem Perfect and Gt-Stem Preterite forms are identical.

APPENDIX B

B.1 Percent Reliability of a Classification

Reliability refers to the extent to which a particular verb form could be identified by one particular classifier. It does not claim that the classification was unique to that form. Uniqueness and reliability are distinct concepts.

G-Stem Mode to Reliability

Present	3-S-C	23	70
	2-S-M	11	97.5
	2-S-F	11	100
	1-S-C	11	60
	3-P-M	36	100
	3-P-F	36	57.5
	2-P-C	11	100
	1-P-C	11	97.5

Preterite	3-S-C	18	100
	2-S-M	18	100
	2-S-F	18	100
	1-S-C	18	100
	3-P-M	18	100
	3-P-F	18	100
	2-P-C	18	100
	1-P-C	18	100

Perfect	3-S-C	6	75
	2-S-M	14	55
	2-S-F	14	62.5
	1-S-C	6	90
	3-P-M	3	70
	3-P-F	3	52.5
	2-P-C	14	67.5
	1-P-C	6	75

Gt-Stem Mode to Reliability

Present	3-S-C	1	62.5
	2-S-M	25	52.5
	2-S-F	25	70
	1-S-C	13	97.5
	3-P-M	1	52.5
	3-P-F	1	50
	2-P-C	25	67.5
	1-P-C	29	50

Perfect	3-S-C	5	50
	2-S-M	25	67.5
	2-S-F	25	77.5
	1-S-C	13	52.5
	3-P-M	9	100
	3-P-F	9	95
	2-P-C	25	77.5
	1-P-C	29	70

Gtn-Stem Mode to Reliability

Present	3-S-C	1	62.5
	2-S-M	25	52.5
	2-S-F	25	70
	1-S-C	13	97.5
	3-P-M	1	52.5
	3-P-F	1	50
	2-P-C	25	67.5
	1-P-C	29	50

Perfect	3-S-C	5	50
	2-S-M	25	67.5
	2-S-F	25	77.5
	1-S-C	13	52.5
	3-P-M	9	100
	3-P-F	9	95
	2-P-C	25	77.5
	1-P-C	29	70

Figure B-1: The G, Gt, and Gtn stems reliability percentage for each verb form classification, by tense. The three columns represent the verb form, mode, and percent reliability respectively.

D-Stem Mode to Reliability

Present	3-S-C	24	100
	2-S-M	12	97.5
	2-S-F	12	100
	1-S-C	24	100
	3-P-M	36	100
	3-P-F	36	90
	2-P-C	12	100
	1-P-C	12	95

Preterite	3-S-C	24	100
	2-S-M	12	100
	2-S-F	12	100
	1-S-C	24	100
	3-P-M	36	100
	3-P-F	36	92.5
	2-P-C	12	100
	1-P-C	12	97.5

Perfect	3-S-C	16	42.5
	2-S-M	28	97.5
	2-S-F	28	100
	1-S-C	16	42.5
	3-P-M	16	87.5
	3-P-F	16	80
	2-P-C	28	100
	1-P-C	28	77.5

Dt-Stem Mode to Reliability

Present	3-S-C	32	32.5
	2-S-M	28	92.5
	2-S-F	28	100
	1-S-C	32	32.5
	3-P-M	16	77.5
	3-P-F	16	65
	2-P-C	28	100
	1-P-C	28	75

Perfect	3-S-C	21	95
	2-S-M	33	100
	2-S-F	33	100
	1-S-C	21	95
	3-P-M	9	72.5
	3-P-F	21	62.5
	2-P-C	33	100
	1-P-C	33	100

Dtn-Stem Mode to Reliability

Present	3-S-C	1	100
	2-S-M	34	92.5
	2-S-F	34	92.5
	1-S-C	1	100
	3-P-M	1	100
	3-P-F	1	100
	2-P-C	34	97.5
	1-P-C	34	100

Figure B-2: The D, Dt, and Dtn stems reliability percentage for each verb form classification, by tense. The three columns represent the verb form, mode, and percent reliability respectively.

Š-Stem Mode to Reliability

Present	3-S-C	24	97.5
	2-S-M	12	100
	2-S-F	12	100
	1-S-C	24	97.5
	3-P-M	36	100
	3-P-F	36	80
	2-P-C	12	100
	1-P-C	12	100

Preterite	3-S-C	24	100
	2-S-M	12	100
	2-S-F	12	100
	1-S-C	24	100
	3-P-M	36	100
	3-P-F	36	85
	2-P-C	12	100
	1-P-C	12	100

Perfect	3-S-C	32	45
	2-S-M	28	100
	2-S-F	28	100
	1-S-C	32	45
	3-P-M	16	100
	3-P-F	16	90
	2-P-C	28	100
	1-P-C	28	77.5

Št-Stem Mode to Reliability

Present	3-S-C	32	40
	2-S-M	28	100
	2-S-F	28	100
	1-S-C	32	40
	3-P-M	16	100
	3-P-F	16	90
	2-P-C	28	100
	1-P-C	28	77.5

Perfect	3-S-C	21	100
	2-S-M	33	100
	2-S-F	33	100
	1-S-C	21	100
	3-P-M	21	87.5
	3-P-F	21	100
	2-P-C	33	100
	1-P-C	33	100

Štn-Stem Mode to Reliability

Present	3-S-C	1	100
	2-S-M	34	100
	2-S-F	34	100
	1-S-C	1	100
	3-P-M	1	100
	3-P-F	1	100
	2-P-C	34	100
	1-P-C	34	100

Figure B-3: The Š, Št, and Štn stems reliability percentage for each verb form classification, by tense. The three columns represent the verb form, mode, and percent reliability respectively.

N-Stem Mode to Reliability

Present	3-S-C	3	82.5
	2-S-M	14	45
	2-S-F	27	85
	1-S-C	3	82.5
	3-P-M	4	95
	3-P-F	4	80
	2-P-C	27	75
	1-P-C	3	62.5

Preterite	3-S-C	6	100
	2-S-M	6	77.5
	2-S-F	14	50
	1-S-C	6	100
	3-P-M	6	62.5
	3-P-F	6	87.5
	2-P-C	14	65
	1-P-C	6	95

Perfect	3-S-C	3	77.5
	2-S-M	27	90
	2-S-F	27	95
	1-S-C	3	92.5
	3-P-M	3	80
	3-P-F	3	77.5
	2-P-C	27	90
	1-P-C	3	60

Ntn-Stem Mode to Reliability

Present	3-S-C	5	42.5
	2-S-M	25	97.5
	2-S-F	25	100
	1-S-C	13	95
	3-P-M	1	52.5
	3-P-F	1	47.5
	2-P-C	25	100
	1-P-C	13	82.5

Perfect	3-S-C	5	52.5
	2-S-M	25	97.5
	2-S-F	25	100
	1-S-C	13	60
	3-P-M	9	100
	3-P-F	9	85
	2-P-C	25	100
	1-P-C	13	52.5

Figure B-4: The N and Ntn stems reliability percentage for each verb form classification, by tense. The three columns represent the verb form, mode, and percent reliability respectively.

B.2 Percentage Reliability

	% Reliability		
Stem	Present	Preterite	Perfect
G	85.31	100.00	68.44
Gt	79.69		79.38
Gtn	62.81		73.75
D	97.81	98.75	78.44
Dt	71.88		90.63
Dtn	97.81		
Š	96.88	98.13	82.19
Št	80.94		98.44
Štn	100.00		
N	75.94	79.69	82.81
Ntn	77.19		80.94
Total	84.20	94.14	81.67

(a) Percentage Reliability by Tense

	% Reliability
G	84.58
Gt	79.53
Gtn	68.28
D	91.67
Dt	81.25
Dtn	97.81
S	92.40
St	89.69
Stn	100.00
N	79.48
Ntn	79.06
Total	84.91

(b) Percentage Reliability by Stem

Figure B-5: **(a)** Shows the percentage reliability by tense of each verb form. **(b)** Shows the percentage reliability by stem of each verb form.

APPENDIX C

C.1 Classification Reliability

The charts below illustrate the percentage reliability of each category a particular verb form had been assigned.

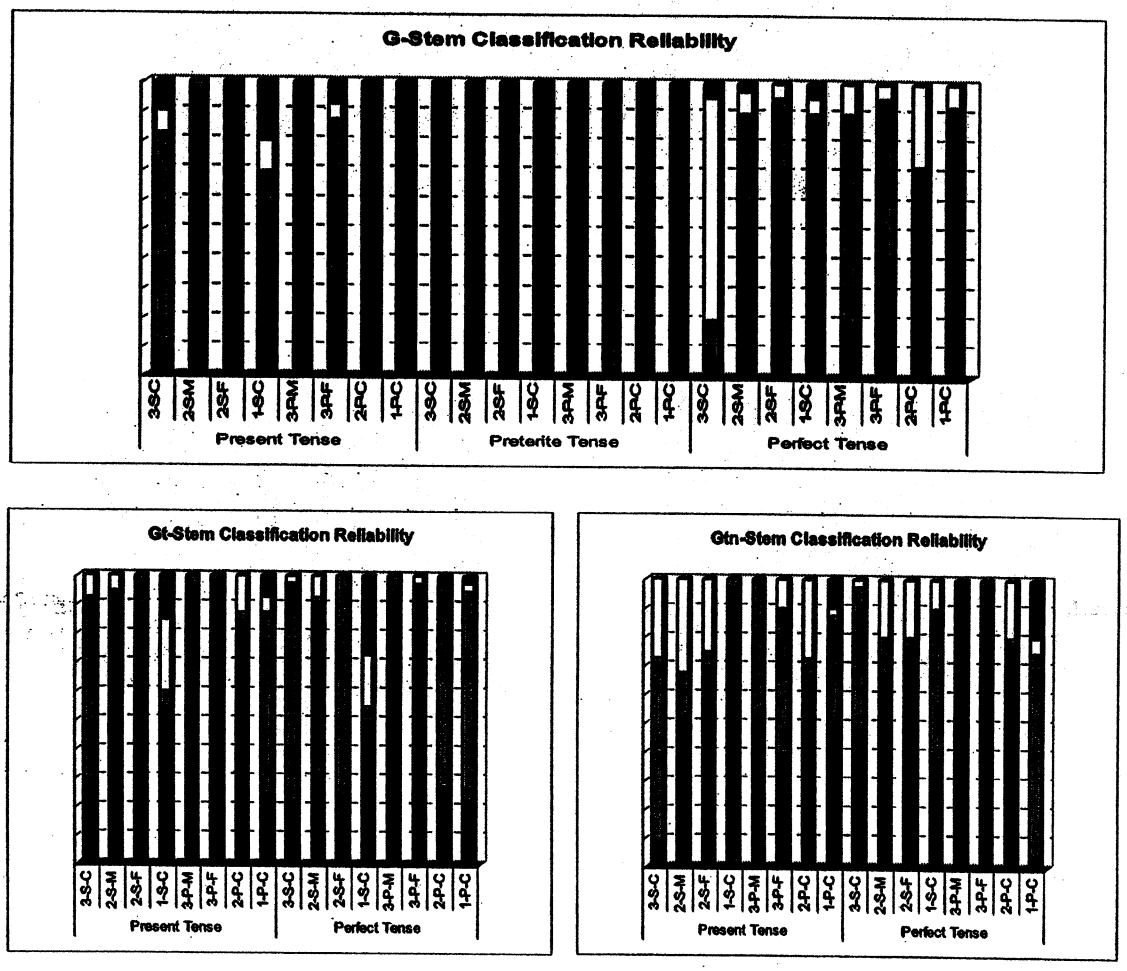


Figure C-1: The G, Gt, and Gtn stems reliability percentage for each classification each verb form was assigned.

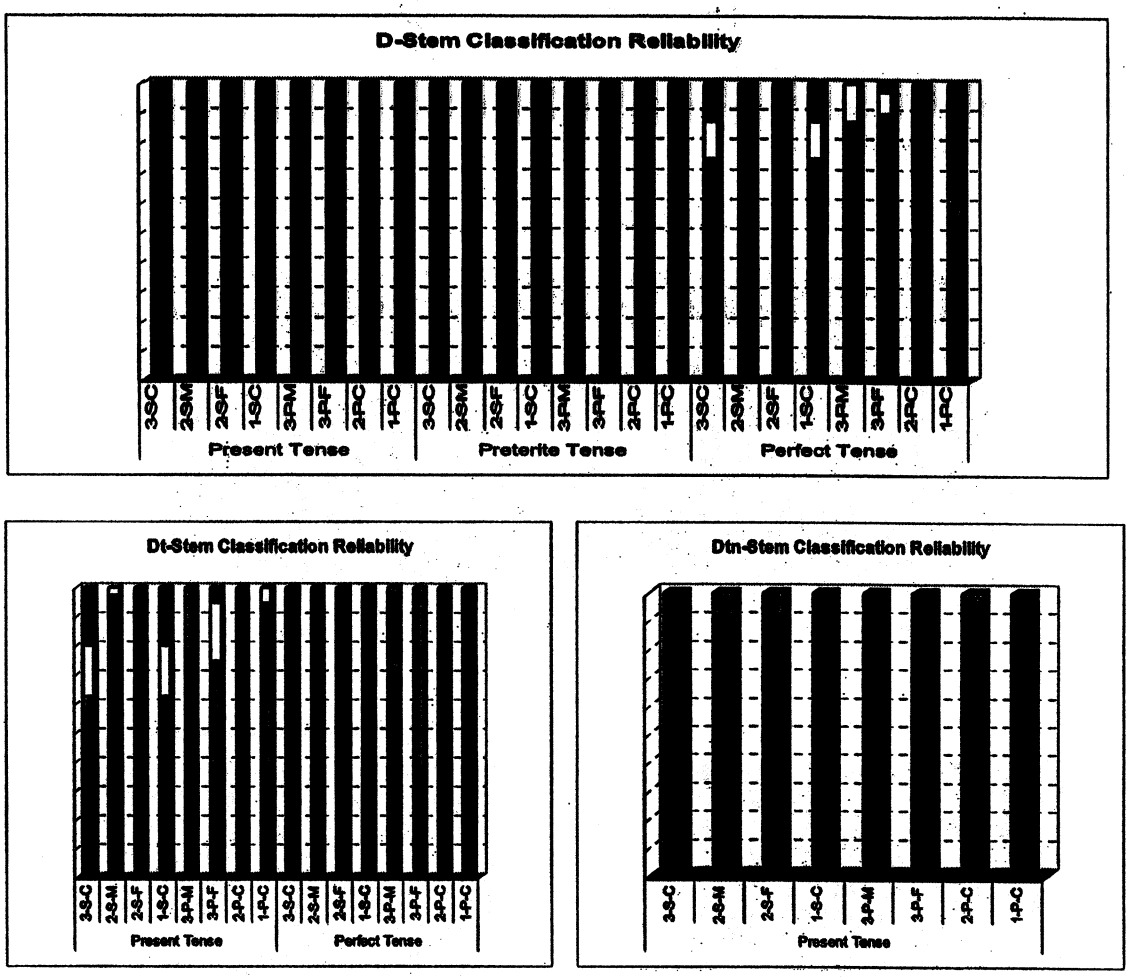


Figure C-2: The D, Dt, and Dtn stems reliability percentage for each classification each verb form was assigned.

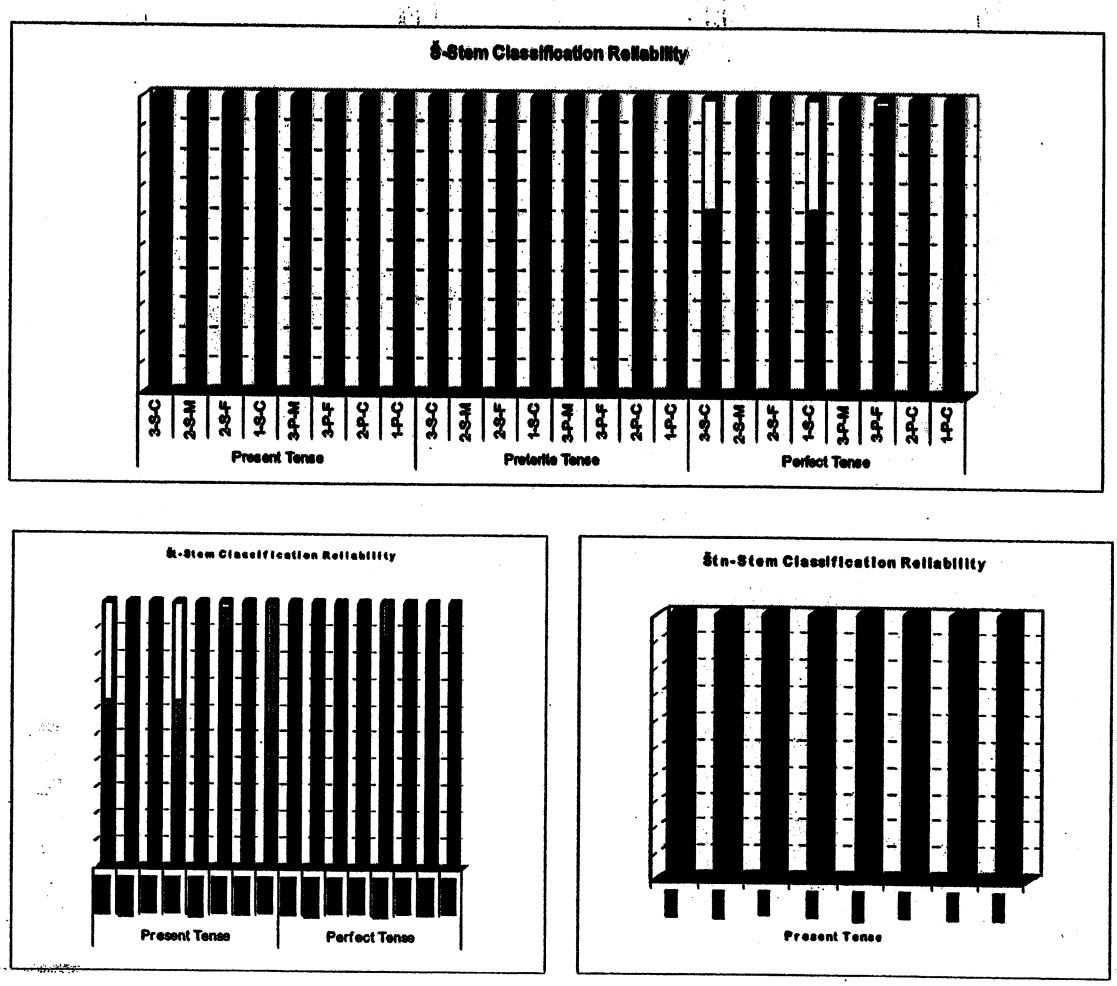
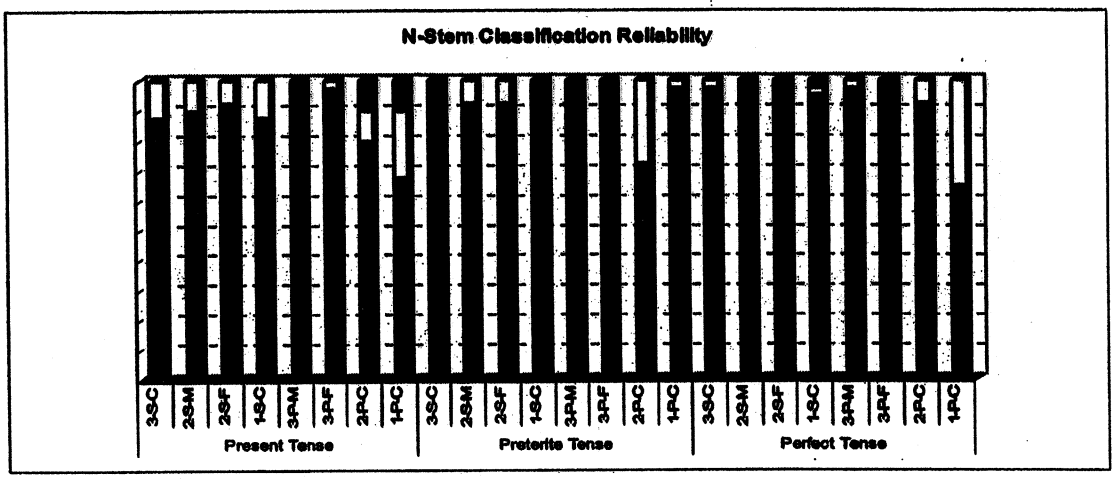


Figure C-3: The Š, Št, and Štn stems reliability percentage for each classification each verb form was assigned.



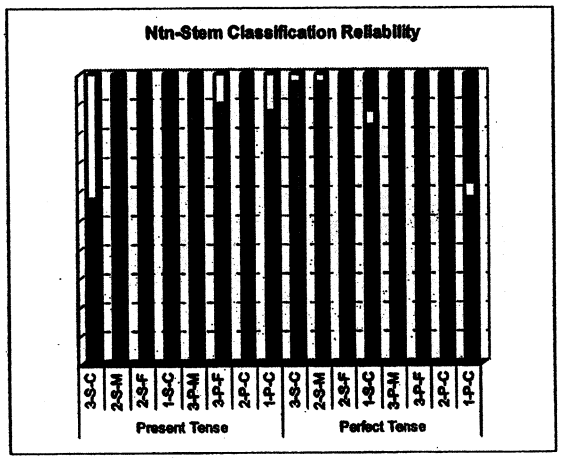


Figure C-4: The N and Ntn stems reliability percentage for each classification each verb form was assigned.

APPENDIX D

D.1 Classification of Person

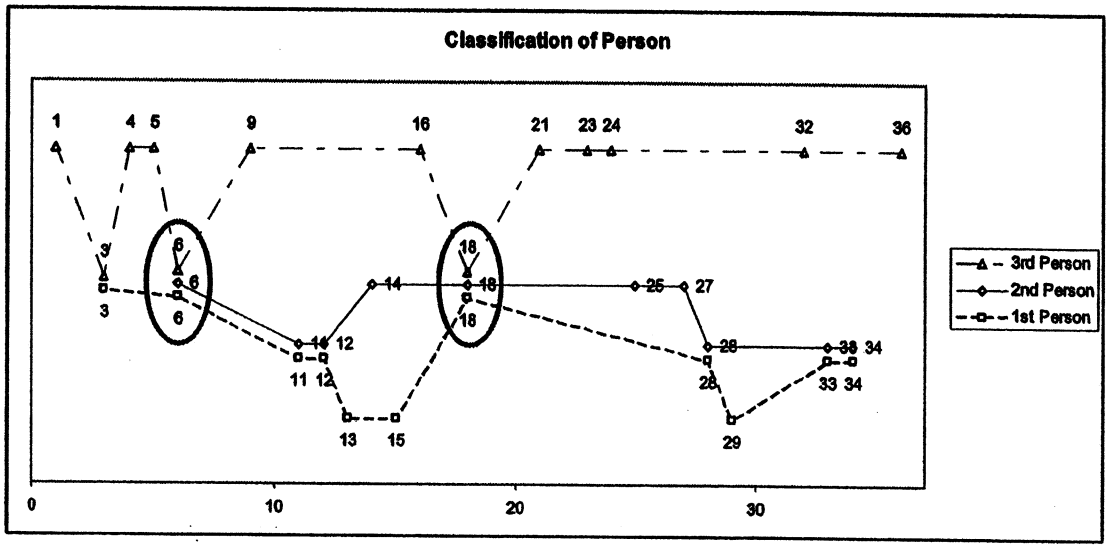


Figure D-1: Shows the various collisions produced by the Syntactic Classifier by person. Collisions involving the 3rd and 2nd Person are circled to illustrate the dependable classification of these two persons. The proximity of collisions in the chart is meant only to highlight them. Given frequency data for the classification of the various verb forms showed that the collisions were primarily the result of a misclassification of the 1st Person.

D.2 Person Classification Data

	Correct	Total	% Correct
3rd	80	81	98.77
2nd	68	69	98.55
1st	10	34	29.41
All	158	184	85.87

(a) Percentages given Considerations

	Correct	Total	% Correct
3rd	71	84	84.52
2nd	55	72	76.39
1st	10	36	27.78
All	136	192	70.83

(b) Percentages prior to Considerations

Figure D-2: **(a)** Gives the percentages of correct number classifications given considerations for unanalizable G-Stem Preterite anomaly, and error assignment. **(b)** Gives the percentages of correct number classifications prior to the considerations above.

Kohonen Maps for Automated Microarray Gridding

Thomas E Royce
Yale University, Department of Computational Biology and Bioinformatics
New Haven, CT 06520

Abstract

An unsupervised algorithm for microarray gridding is developed utilizing Kohonen self-organizing maps. The method identifies grids correctly in images with well-formed spot arrangements as well as in images containing missing or faint spots, occasional spot drift, and global rotations. The proposed algorithm works well for finding grids that lack sub-grid divisions.

Keywords – microarray, automation, gridding, Kohonen maps, self-organization

1. INTRODUCTION

Microarray technology allows for the simultaneous measurement of tens of thousands of genes' messenger RNA (mRNA) expression levels and is rapidly becoming the experimental platform of choice for the field of functional genomics (Brown and Botstein 1999). A microarray experiment is a multi-step process utilizing techniques from microbiology and molecular genetics (Nguyen, Arpat et al. 2002) as well as from the computational sciences (Leung and Cavalieri 2003). The computational protocol of microarray gridding, which, in general, is the identification of a grid of spots within an experimentally obtained scanned image, is considered here.

Scanned microarray images typically display thousands of spots, often organized in regular rectangular blocks (Figure 1). Each spot consists of several to hundreds of bright pixels within the image. The goal of microarray gridding is to find a mapping f to a potentially irregular grid of spots within the scanned image from a perfect grid of identical dimensionality. Each vertex within the perfect grid carries with it a spot identifier, such as a gene name, so that by identifying f, each spot in the scanned image becomes associated with its identifier. Once f is found, brightness measurements can

subsequently be taken for each identifier in the perfect grid. These brightness measurements are the principle aim of a microarray experiment.

Irregularities such as grid rotations and spot drifting (Figure 2) make the automatic gridding of these images a non-trivial task. Efforts toward automation are hampered further by the fact that some (or many) spots may be missing and that their brightness can vary by up to four degrees of magnitude from spot to spot. Yielding to these difficulties, microarray gridding usually requires an experimentalist to define f manually using any one of several software packages in a 'point and click' fashion in which the vertices of the perfect grid are 'dragged' onto the spots in the scanned image. This procedure can take hours to days depending on the software used and the number of vertices to be found. An unsupervised method of microarray gridding addressing these issues is introduced here employing a modification of Kohonen self-organizing maps (SOM) (Kohonen 1997). The method is able to automatically find grids in images containing a sparse arrangement of spots, having non-perfect spot placement, and/or having global rotations. The only inputs required of the algorithm are the microarray image files themselves, and the expected number of rows and columns of spots to be found therein.

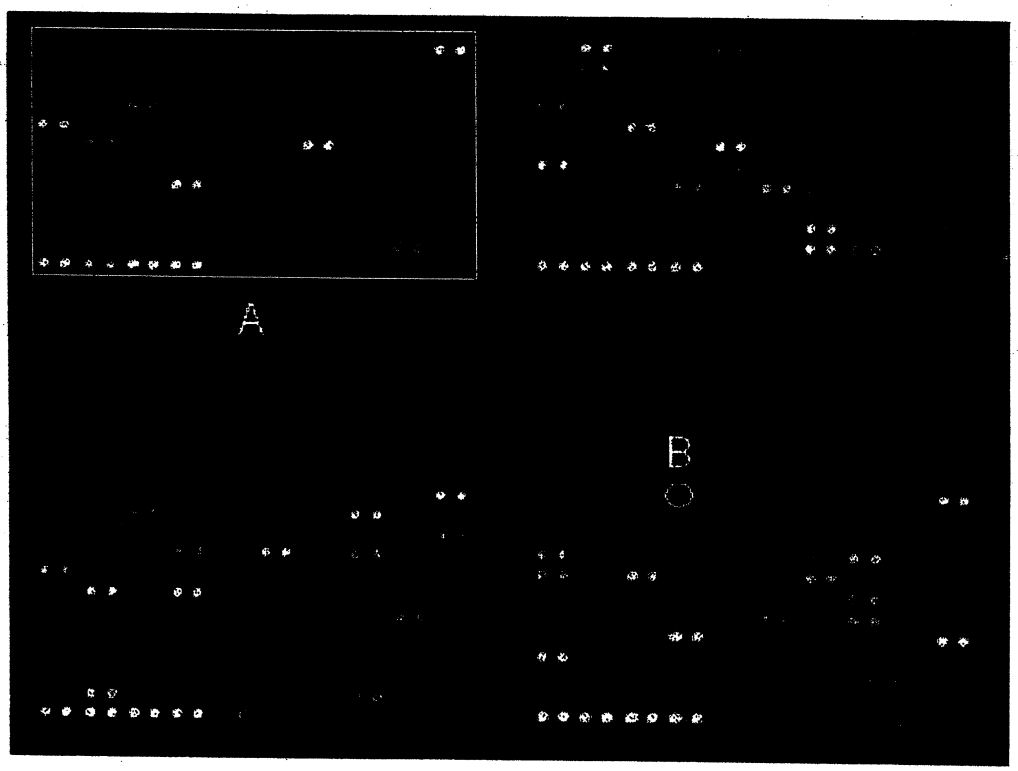


Figure 1: scanned microarray image showing block structure (A) and an example spot (B)

In Section 1.1, a brief introduction to the microarray technology is given. Section 1.2 outlines the computational aspects of the microarray experiment. This introduction concludes with an account of previous work on the microarray gridding problem in section 1.3. The SOM algorithm is developed in Section 2 followed by empirical results in Section 3 and a brief discussion in Section 4.

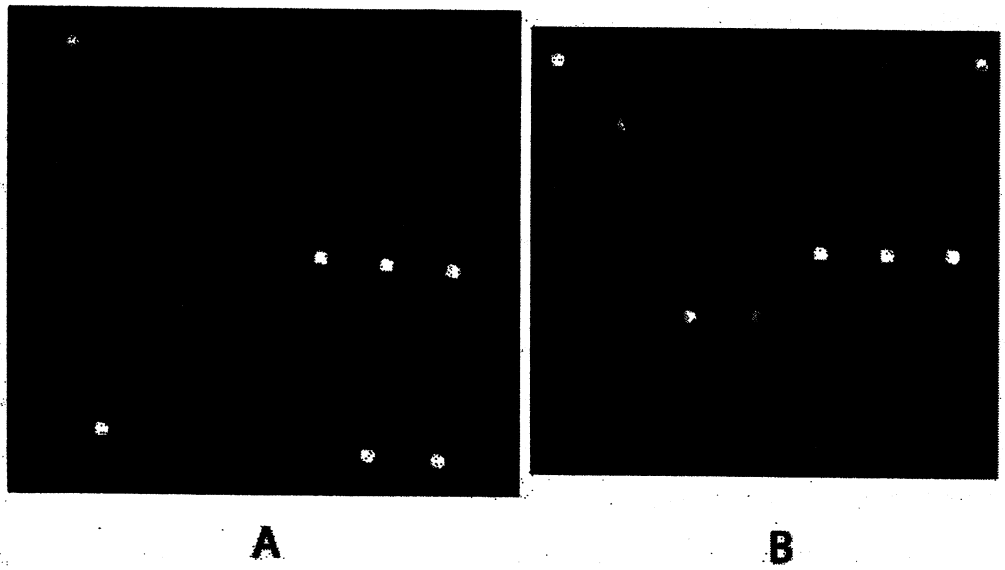


Figure 2: rotated block (A) and drifting spots (B)

1.1 The microarray experiment

The central dogma of molecular genetics is that the creation of a new protein molecule (the molecular machines of the cell) is reliant upon the presence within the cell of its corresponding mRNA molecule (Alberts 1994). Following a set of rules known as the genetic code, the cell is able to process any given mRNA molecule and produce the corresponding functional protein. The mRNA molecule, in turn, is a transcribed copy of a specific piece of DNA within the organism's genome. This specific piece of DNA is known as a gene and, therefore, it is usually extrapolated that each gene within the genome codes for a particular protein.

Since the relative populations of proteins within a cell, along with their relative abundance levels, determines the cell's function (i.e. whether it is a nerve cell, blood cell,

tumor cell, etc.) it would be extremely useful for biologists to be able to monitor these populations of proteins under various environmental and physiological conditions. For instance, identifying proteins that are present at high levels in tumor cells relative to their healthy counterparts could lead to the discovery of useful biomarkers – molecules that can provide evidence of cancerous tissue in seemingly healthy patients at early stages of the disease (Dhanasekaran, Barrette, et al. 2001).

Unfortunately, defining technologies for quantitating the entire population of proteins present within a cell is a difficult task. Nascent technologies exist with this aim (Issaq, Veenstra et al. 2002; Washburn, Ulaszek et al. 2002; Yan, Devenish et al. 2002; Ghaemmaghami, Huh et al. 2003; Peng, Elias et al. 2003), but the task remains an open challenge of great interest to the biotechnology community. Microarrays, while they do not quantify relative protein abundances directly, do provide insight into this problem. Microarrays are able to quantitate the cell's population of mRNAs both for their presence and for their relative expression levels. Since mRNA molecules are the precursors to proteins, as discussed above, global analyses of mRNA populations within the cell can provide useful insights into the molecular biology of cells in different tissues and under different environmental conditions.

A microarray consists of a solid surface, typically a glass microscope slide, with thousands of genes immobilized on the surface in an ordered fashion (Figure 1). The microarray is utilized as follows (for a good review of the experimental protocol, see (Cheung, Morley et al. 1999). First, a population of mRNA molecules is extracted from a biological sample. Next, the molecules are labeled with a fluorescent dye. Then, this sample is washed over the microarray. mRNA molecules, or derivatives thereof, will hybridize, or *stick* (via Watson-Crick complementarity), to their respective gene on the microarray. Therefore, interpreting levels of fluorescence for each gene on the microarray serves as measurement of the abundance of each mRNA species within the population. A common variation to this protocol is to obtain two different biological samples, label them with different fluorescent dyes, and allow them to hybridize to the microarray at the same time. In this way, it is possible to identify differences in mRNA abundance levels for each gene between two biological samples (Schena, Shalon et al. 1996).

1.2 Computational aspects

Following hybridization, the microarray is scanned for fluorescent emissions using a specialized scanner. For each biological sample hybridized, an independent scan is performed which results in a 16-bit grayscale Tagged Image File Format (TIFF) image (Figure 1). So, the common experiment in which two different biological samples are measured on a single microarray simultaneously, results in two 16-bit grayscale images. Once scanned images are obtained, a grid needs to be laid on top of them with each vertex in the grid being aligned with bright regions (spots) of the image that correspond to the genes on the microarray. As noted previously, the process of applying a grid to the TIFF images can be an arduous task for the experimentalist as spots often appear in a slightly warped, or rotated arrangement and spot spacing is not always entirely uniform across the microarray. These imperfections, arising from the physical microarray production process, have dampened attempts at automatic gridding. Therefore, researchers have largely had to resort to applying the grids manually, a process that can take hours or even days.

Each vertex of the grid has an assigned gene identifier, and in this way, intensities can be extracted from each spot (gene) on the microarray once the grid is applied. These intensities are used as measurements of mRNA abundance and serve as the raw data for downstream statistical analyses.

1.3 History

Despite the obvious utility of developing an automated microarray gridding system, few attempts to do so have been made. One approach builds histograms of horizontal and vertical intensities from which a grid is deduced (Jain, Tokuyasu et al. 2002). Peaks of the histogram correspond to the rows and columns of spots on the array. This system works well for well-formed grids but its performance worsens as spot arrangements are skewed or rotated and post-processing steps are needed to aid in these corrections. Varying illumination across the image can also cause the histogram-finding algorithm to falter.

An approach which relies on graph algorithms for gridding has also been proposed (Jung and Cho 2002). First, the algorithm thresholds the image into foreground and background pixels and then applies a variant on the k-nearest neighbor algorithm to identify blocks of spots. Then operating on each block individually, a similar graph algorithm is applied to align the grid to spots in the image. This algorithm does a good job and detecting and accounting for grid rotation and skewedness. Requirements of minimum block spacing and the requirement of 'anchor' spots, which are known to exist a priori in the corner of each block, limit the method.

Techniques from the discipline of mathematical morphology (Angulo and Serra 2003) have also been applied to this problem but warping and rotational flaws limit the approach. This technique is mainly of theoretical interest.

An approach to solving the gridding problem is presented that utilizes Kohonen self-organizing maps (SOM). The SOM approach is able to work for microarray images containing both skewed regions and rotational flaws. The approach also remains robust to images that are missing a large proportion of expected spots.

2. KOHONEN SELF-ORGANIZING MAPS FOR MICROARRAY GRIDDING

The algorithm presented requires three inputs at its onset: (1) a set of microarray TIFF images (2-D arrays of pixels and their intensities) defines the input space in which to search for the grid, (2) the number of expected rows of spots to identify, and (3) the expected number of spot columns to identify. With these inputs, the microarray gridding process is then automated. The algorithm is developed in the following sections and demonstrated with sample images.

2.1 Pre-processing

Hereafter for clarity, we confine our attention to a microarray experiment utilizing two biological samples, and thus two input 16-bit TIFF images. The process is easily generalized to experiments using either just one or more than two samples (images).

First, the two input 16-bit TIFF images G and R are combined for gridding purposes. The combining process follows from (Yang, Buckley et al. 2002) but will be included here for completeness. First, the square root of each 16-bit pixel's intensity in each

image is computed. Next, the median of the square-root transformed intensities is calculated for each image. These medians will be represented as med_G and med_R for the images labeled G and R, respectively.

An initial linear combination is taken of the two images' square-root intensities at each pixel i, the result being such that the two images are normalized to their medians and summed. This is done to ensure that neither image dominates the combined image and thus the downstream gridding process. For instance, consider in the extreme case, two microarrays displaying non-intersecting sets of spots in their respective images. That is, spots showing up in one image to not appear in the other image and vice-versa. If one image has much higher pixel intensities than the other, then the gridding process would be biased to identifying spots in the brighter image and not in the dimmer one.

For any given pixel normalized in this fashion, a summed value greater than 2^8-1 is then set equal to 2^8-1. This creates a single, combined 8-bit TIFF image. This linear combination and thresholding is summarized in Equation 1 where C_i is the value obtained for pixel i in the new combined image and R_i and G_i represent the square-root transformed intensities for pixel i in images R and G, respectively.

$$(1)\quad C_i = \min\left\{2^8 - 1, G_i + R_i\left(\frac{med_G}{med_R}\right)\right\}$$

Using the combined 8-bit TIFF (set of all intensities, C_i) obtained from Equation 1, foreground pixels are then separated from the background pixels based on their relative intensities. To do this, Otsu's histogram thresholding algorithm is used (Otsu 1979). The thresholded foreground pixels' (x,y)-coordinates are recorded and subsequently used to guide the SOM algorithm.

2.2 Initialization of the SOM

An SOM consists of computational units (hereafter referred to as neurons) arranged in a grid (Figure 3). For the application at hand, this grid is a rectangle where interior neurons have four neighbors, edge neurons have three neighbors and corner neurons have two neighbors. Besides storing information identifying its neighbors, a neuron also stores

two additional values representing x- and y-coordinates in the input (image) space. These two values are traditionally termed 'weights,' and this term will be used hereafter. It is convenient to represent these weights as a single two-element column vector $m_i(t)$ where i indicates that the ith neuron in the grid is being considered. This vector is commonly referred to as the 'weight vector' of neuron i. Note also that each weight vector carries with it a clock variable t. This time-dependence will be specified later.

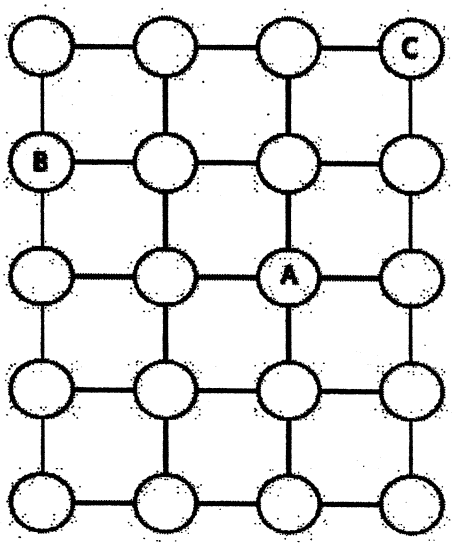


Figure 3: SOM with representative interior (A), border (B), and corner (C) neurons

A priori, the expected number of spot columns and rows in the microarray image is assumed known. A grid of neurons with an identical number of spots, rows and columns is initialized to model the microarray as follows. Each neuron sets its weights such that if all neurons' weight vectors were plotted as nodes in two-dimensional space and topological connections between neurons were plotted as arcs between the neurons, the resulting graph would resemble a regular lattice, as in Figure 3. At onset, this grid is positioned at the center of the input image space spanning the middle 10% of the image.

2.3 Self-organization

The goal of self-organization is to adjust the weights (x- and y-coordinates in input space) of the neurons so that the net evolves into an accurate representation of where spots lie in the original microarray. In the resulting net, the neurons are centered at each

spot in the image (or where the spot should be if it is missing from the image). This net is achieved through an iterative process described as follows.

First, a foreground pixel obtained from Otsu's algorithm is chosen at random. Note that each foreground pixel carries with it the (x,y) coordinate representing its location within the microarray image. Let such an input pixel be denoted by this coordinate as the two-element column vector $X(t)$ where t again indicates a clock variable.

Next, all neurons in the grid compare their x-coordinate with the x-coordinate of $X(t)$. The goal here is to identify the column J within the grid that is closest to $X(t)$ in the horizontal direction and is formalized in Equations 2 and 3 where $X_x(t)$ indicates the x-coordinate of the input pixel and $m_{ix}(t)$ denotes the x-coordinate of neuron i.

$$(2)\quad j = \arg\min_i \left\{ \left| X_x(t) - m_{ix}(t) \right| \right\}$$

$$(3)\quad J = \left\{ i \mid m_{ix} = m_{jx} \right\}$$

Ties of minimum distance that are potentially obtained from Equation 2 are resolved randomly.

Once J is found, the element within J that is closest to $X(t)$ in the vertical direction is then found. This neuron is labeled α, the latter specified in Equation 4. $X_y(t)$ and $m_{iy}(t)$ denotes the y-coordinate of the input vector and the ith neuron, respectively.

$$(4)\quad \alpha = \arg\min_i \left\{ \left| X_y(t) - m_{iy}(t) \in J \right| \right\}$$

Next, all of the neurons in the column J have their x-coordinates updated according to Equation 5.

$$(5)\quad m_{ix}(t+1) = m_{ix}(t) + \eta(t) h_{\alpha i}(t) [X_x(t) - m_{ix}(t)]$$

Equation 5 introduces the functions h and η that decrease the degree to which m_{ix} is updated. Both of these functions depend upon the iteration step t and are described in Equations 6-11. Intuitively, the function h causes those neurons most proximal to α to be

moved closer to $X(t)$ than those less proximal to α. The function η is a 'forgetting function,' which decreases in value at each iteration of the algorithm. Together, these two functions aid in the convergence and eventual fine-tuning of the grid in that, as time passes, a smaller 'neighborhood' of neurons adjusts their x-coordinates towards $X(t)$. Moreover, these adjustments trend smaller due to the forgetting function. In Equations 6 and 7, *ITER* is the number of iterations that the algorithm runs and, in Equation 8, γ indicates the number or rows in the grid. In Equation 10, α_y denotes the y-coordinate of α.

(6) $\tau_{column} = \dfrac{ITER}{\log \gamma}$

(7) $\tau_{forget} = \dfrac{-ITER}{\log 0.001}$

(8) $\sigma_{column} = \gamma \exp^{\frac{-t}{\tau_{column}}}$

(9) $\eta(t) = \exp^{\frac{-t}{\tau_{forget}}}$

(10) $d_i = |\alpha_y - m_{iy}|$

(11) $h_{\alpha i}(t) = \exp^{\frac{-d_i^2}{2\sigma_{column}^2}}$

Following the weight adjustments called for Equation 5, an analogous operation is performed whereby those neurons residing in the row closest to the input $X(t)$ (in the vertical sense) have their y-coordinates updated in a manner following Equations 5-11. The only differences are that γ is set to the number of columns in the grid and the neurons in the closest row have their y-coordinates updated rather than their x-coordinates. Figure 4 illustrates what a given horizontal update of the grid in Figure 3 might look like.

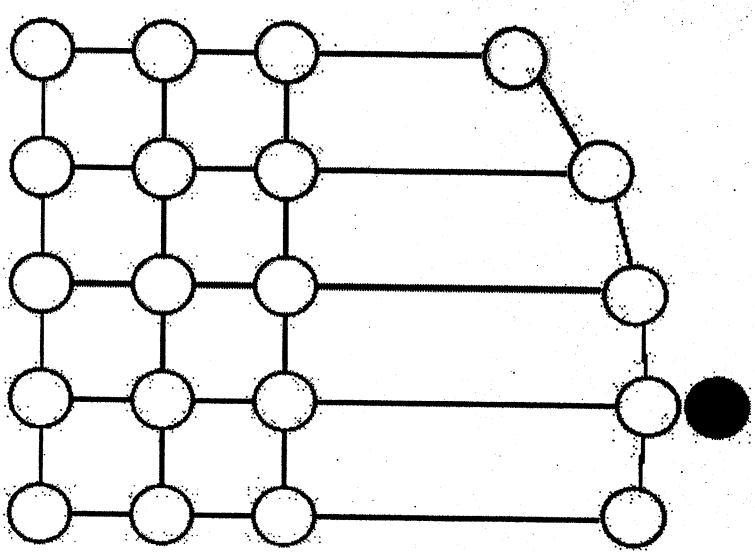


Figure 4: horizontal update of Figure 3 to input (black circle)

3. RESULTS

A formal proof demonstrating the validity and theoretical limitations of the SOM approach to microarray gridding is not pursued here. Rather, the results of applying this method to five simple test cases are provided. The first test case is that of a simple seven-by-seven block. Figure 5 illustrates the SOMs success in correctly identifying the microarray's grid. Figure 6 demonstrates that this approach continues to work well on a larger twenty-by-sixteen spot microarray image. Note also that the number of missing spots and increased noise in the thresholded image of Figure 6 appears to have a negligible effect on correctly positioning the grid.

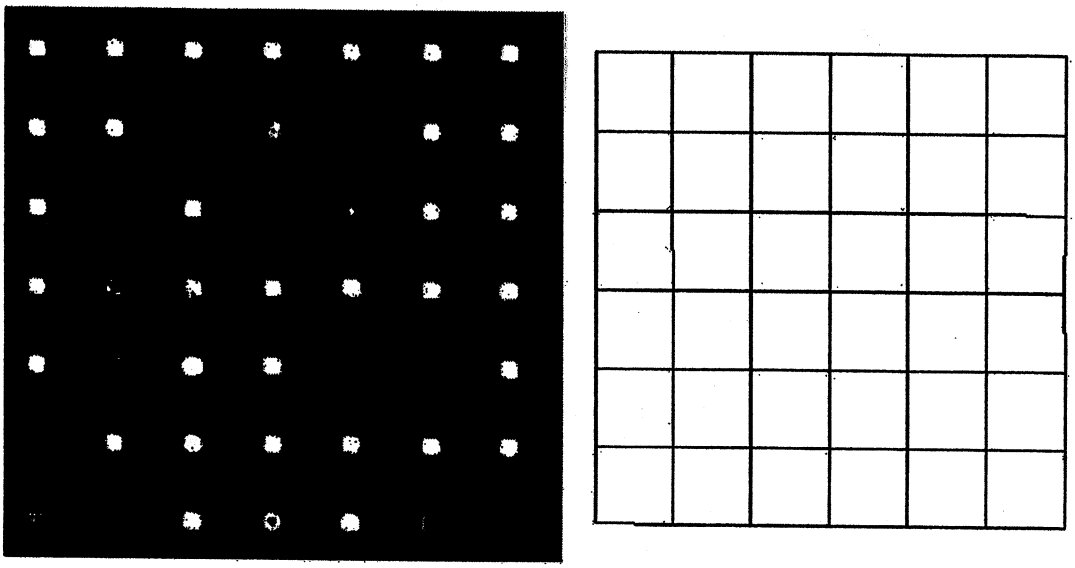


Figure 5: thresholded image (A) and its computed SOM (B)

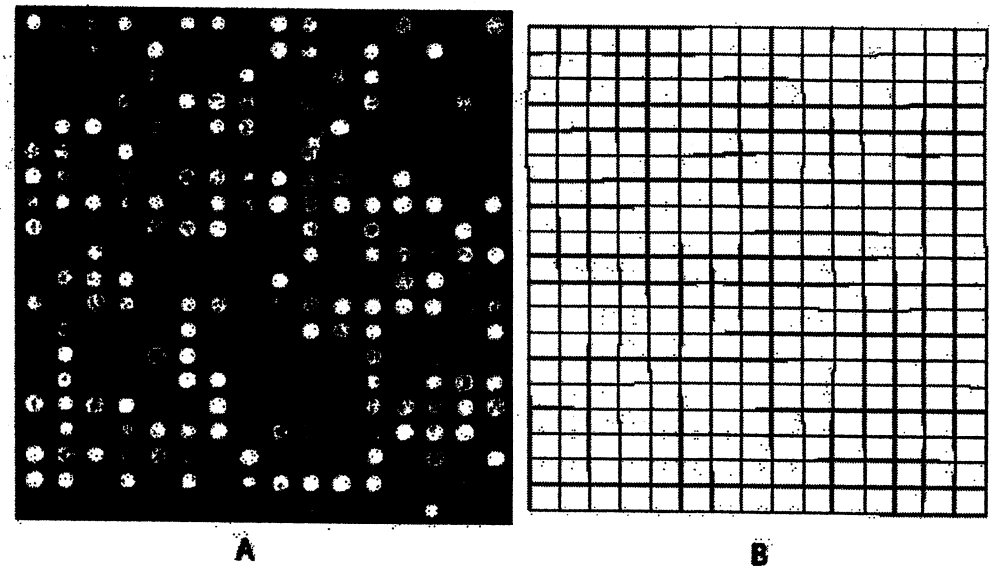


Figure 6: larger thresholded image (A) and its computed SOM (B)

A slightly more challenging test is seen in Figure 7. Here, a seven-by-seven block was created by moving the upper-right spots of Figure 5 slightly to the right. Again, a valid grid was identified.

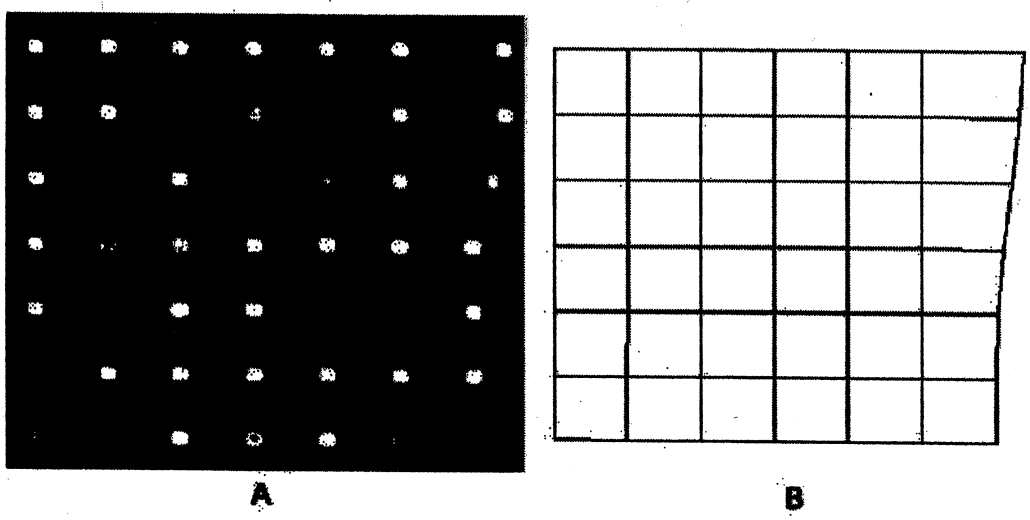


Figure 7: thresholded image with drifting spots (A) and its computed SOM (B)

Another problem often encountered in microarray images is that they can often times be globally rotated. In Figure 8, such an image is shown along with the SOM that still correctly identifies the underlying spots.

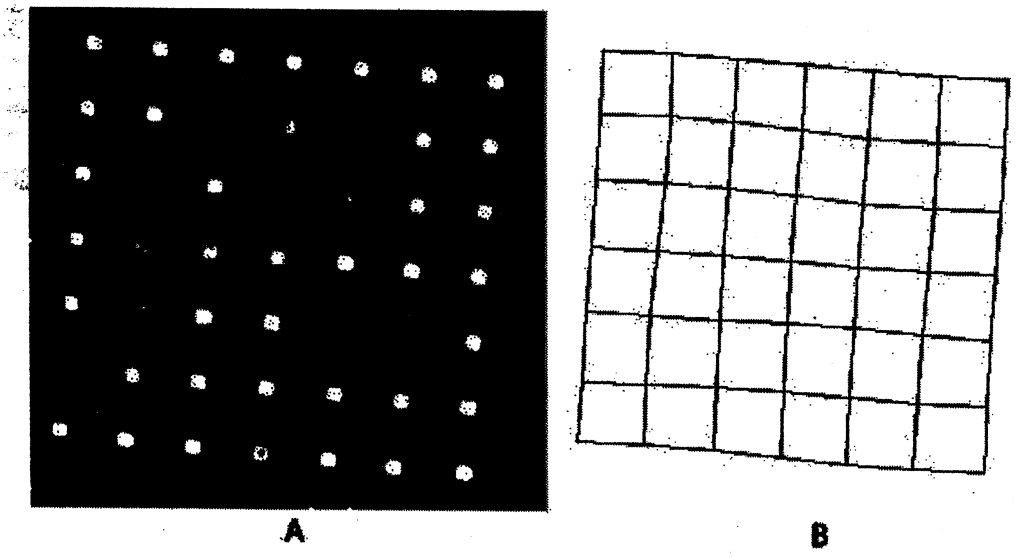


Figure 8: thresholded image with rotation (A) and its computed SOM (B)

When the SOM algorithm is applied to an image with the spots arranged in distinct blocks, the results are not so good, however. It happens that there are many instances in which a single row or column in the SOM grid spans multiple rows or columns in the microarray image (data not shown). This is likely due to the presence of noisy pixels

occurring between two blocks. Its possible that image processing techniques for reducing noise in the thresholded image may abate this problem.

4. DISCUSSION

Researchers in the biological sciences are increasingly becoming interested in the development of high-throughput experiments. The microarray technology is certainly an enabling tool with this objective as several thousands of genes can be studied simultaneously with a single microarray. A time consuming step in the microarray protocol is labeling (assigning gene identifiers to) the spots visualized in a microarray image. This 'gridding' procedure is typically done manually because irregularities commonly found in the scanned image are obstacles to automated processing. The algorithm presented here is a step towards such automation. Images containing spots of fairly uniform spacing are handled readily, even if they contain spot drifting or global rotations. The algorithm is not yet suited for typical microarray images since these are normally encountered as having several distinct blocks of spots rather that a single large block encompassing all of the image's spots. We may expect the development of algorithms that can first identify the block regions so that the described SOM algorithm may be applied on a block-by-block basis. It is further conceivable that a similar SOM algorithm is up to this task of block segmentation as well.

5. REFERENCES

Alberts, B. (1994). Molecular biology of the cell. New York, Garland Pub.

Angulo, J. and J. Serra (2003). "Automatic analysis of DNA microarray images using mathematical morphology." Bioinformatics **19**(5): 553-62.

Brown, P. O. and D. Botstein (1999). "Exploring the new world of the genome with DNA microarrays." Nat Genet **21**(1 Suppl): 33-7.

Cheung, V. G., M. Morley, et al. (1999). "Making and reading microarrays." Nat Genet **21**(1 Suppl): 15-9.

Dhanasekaran, S. M,, Barrette, T. R. , et al (2001). "Delineation of prognostic biomarkers in prostate cancer." Nature **412**(6849): 822-6.

Ghaemmaghami, S., W. K. Huh, et al. (2003). "Global analysis of protein expression in yeast." Nature **425**(6959): 737-41.

Issaq, H. J., T. D. Veenstra, et al. (2002). "The SELDI-TOF MS approach to proteomics: protein profiling and biomarker identification." Biochem Biophys Res Commun **292**(3): 587-92.

Jain, A. N., T. A. Tokuyasu, et al. (2002). "Fully automatic quantification of microarray image data." Genome Res **12**(2): 325-32.

Jung, H. Y. and H. G. Cho (2002). "An automatic block and spot indexing with k-nearest neighbors graph for microarray image analysis." Bioinformatics **18 Suppl 2**: S141-51.

Kohonen, T. (1997). Self-organizing maps. Berlin ; New York, Springer.

Leung, Y. F. and D. Cavalieri (2003). "Fundamentals of cDNA microarray data analysis." Trends Genet **19**(11): 649-59.

Nguyen, D. V., A. B. Arpat, et al. (2002). "DNA microarray experiments: biological and technological aspects." Biometrics **58**(4): 701-17.

Otsu, N. (1979). "Threshold Selection Method from Gray-Level Histograms." Ieee Transactions on Systems Man and Cybernetics **9**(1): 62-66.

Peng, J., J. E. Elias, et al. (2003). "Evaluation of multidimensional chromatography coupled with tandem mass spectrometry (LC/LC-MS/MS) for large-scale protein analysis: the yeast proteome." J Proteome Res **2**(1): 43-50.

Schena, M., D. Shalon, et al. (1996). "Parallel human genome analysis: microarray-based expression monitoring of 1000 genes." Proc Natl Acad Sci U S A **93**(20): 10614-9.

Washburn, M. P., R. Ulaszek, et al. (2002). "Analysis of quantitative proteomic data generated via multidimensional protein identification technology." Anal Chem **74**(7): 1650-7.

Yan, J. X., A. T. Devenish, et al. (2002). "Fluorescence two-dimensional difference gel electrophoresis and mass spectrometry based proteomic analysis of Escherichia coli." Proteomics **2**(12): 1682-98.

Yang, Y. H., M. J. Buckley, et al. (2002). "Comparison of methods for image analysis on cDNA microarray data." Journal of Computational and Graphical Statistics **11**(1): 108-136.

A Multi-Stage Technique for Determining Head Orientation from Monocular Images using Neural Networks

Frederick Shic
Yale University, Social Robotics Group
New Haven, CT 06510

Abstract
A technique for determining the three-dimensional head orientation of a human subject given a monocular digital image is described. The first phase of this technique extracts skin tone from an image. The second phase uses the original image and the previously detected skin tonal areas to determine head locations. Finally, the third phase combines all previous information with a generalized feature detector to extract head orientation. Each step relies heavily on solutions guided by feed-forward back propagation neural networks.
Keywords – head orientation, head gaze, face tracking, skin detector, flesh detection, neural networks.

1. INTRODUCTION

The head orientation of a human subject can be used to determine the subject's focus of attention, gauge his level of interest, or even predict his future actions. In conjunction with a robotic system capable of interacting with humans, such information could help in the construction of intuitive man-machine interfaces.

Previous work for determining head orientation from images includes feature detection combined with geometric proof (Gee & Cipolla, 1994; Horprasert et al., 1996), probabilistic texture modeling and matching (Pappu & Beardsley, 1998, Wu & Toyama), and coarse image to orientation solutions using neural networks (Zhao et al., 2002; Stiefelhagen et al., 2001). Each of these techniques performs well under a variety of situations, but no one technique is suitable for all applications. We describe a technique for determining head orientation in a situation in which:

1. The camera environment is fairly constrained in terms of both background and lighting (e.g. the environment of a stationary robot).
2. The entire task, beginning with locating a face in an image and ending with an estimate for head orientation, must be accomplished.
3. Training data is minimal and coarse.
4. The system runs in real-time on the computational equivalent of a standard 2002 desktop computer (computational overhead is minimal).

In addition, we desire an approach that is both efficient and simple. We would like to build a complete (constraint 2) system in which both the setup time is short (constraint 3) and operation is fast (constraint 4). We impose constraints on the environment

(constraint 1) in order to try to reduce the complexity of the total task. The first step in the head orientation system (HOS) is to determine which image pixels in an image correspond to skin. Skin detection allows us to reduce the dimensionality of the search over image space, allowing us to quickly zoom in on candidates which may be human faces. From these potential human face locations, we select and isolate locations which actually correspond to faces, allowing us to focus specifically on the general area of a human head. Finally, from the general location of the verified faces, we automatically locate salient features and reduce the orientation of these features into an orientation of the entire head. Each of these steps is accomplished by training to a sub-task with neural networks for two primary reasons:

1. Neural networks can effectively skip several preprocessing steps, such as color space transformation and lighting normalization, which are normally used in head detection/head recognition systems, as many preprocessing steps are simple transforms.
2. Once trained, neural networks of reasonable size usually run extremely quickly.

We will begin, in section 2, with a discussion of the difficulties of detecting skin tones from images and present a method that seems to adequately address these difficulties. We continue, in section 3, with the presentation of a method for determining which sections of an image correspond to faces when given several candidate skin locations. Finally, we conclude, in section 4, with the presentation of a technique for roughly determining head orientation, given cropped images of faces.

2. SKIN DETECTION

Before we can determine the head orientation of faces appearing in the viewing frustrum of a camera, we have to locate and categorize some initial targets. A good method for reducing the search space of the visual stream is to begin by using skin color as a feature for localization. Using skin color as an initial feature has several advantages:

(1) Pixel-to-pixel reductions to skin are typically fast
(2) Accurate skin detection leads to a dramatically reduced search space
(3) The distribution of colors in skin, as determined by examining patches of skin, for people with pigment in their skin (i.e. non-albino humans), has been shown to fall within a very narrow range of color space, in comparison to the color distribution of background features such as walls, chairs, and carpets.

Skin color detection also has several drawbacks:

(1) There is no universality of agreement on the choice of a proper color space in which to perform skin detection. That is, there is no general agreement on whether, for example, the HSV color space is superior to the YCbCr color space for the purposes of discriminating skin color.

(2) Learning methods for skin detection depend on the breadth of the training data. For instance, it is likely that a skin detector trained solely under bright lightning would perform poorly when presented with images of faces illuminated dimly.
(3) Pixel-level methods for identifying skin do not seem sufficient to segregate faces and typically more advanced methods such as edge detection are necessary.

In order to weigh the importance of these drawbacks and advantages, we first begin by examining the properties of skin tones in various color spaces, and try to build an effective Gaussian model for skin (Section 2.1). We follow this analysis by building a system that maps pixel colors directly to the classification of skin versus non-skin, using neural networks (Section 2.2). Finally, we extend the neural-network model by augmenting the knowledge available at each pixel by adding information concerning the neighborhood around each pixel (Section 2.3).

2.1. Skin detection by color histogram analysis and Gaussian modeling

We first begin by investigating the color spaces that would be appropriate for skin characterization on a camera system embedded on Nico, a robot currently in development in the Yale Social Robotics group. We acquire 25 frames from Nico's wide field-of-view camera at resolution of 320x240 pixels and import the images into Matlab and investigate manually selected regions of skin (shown in Figure 1). Each pixel in the selected region contains numerous pixels, with each pixel represented by a color vector (for instance, red value, green value, and blue value for the traditional RGB color space). For each axis of the color vector, we can count the number of pixels that fall within a certain numeric range (for instance we can count 30 pixels that have a green intensity value of 90 to 110 on an RGB scale) in order to build a color distribution map for the selected region- the color histogram. If we make the assumption that the areas selected manually correspond to representative distributions of skin color, then by examining these color histograms for multiple sites we can draw some conclusions regarding the color properties of skin. We examine several traditional color spaces in order to determine if any one color space is superior to the others for the purposes of discriminating skin coloration: Red-Green-Blue (RGB), Intensity-Chrominance (YCbCr), and Hue-Saturation-Value (HSV) (Figure 2). That is we see if any of these color spaces has a compact representation when viewed as a color histogram. The histogram of an axis of a color space is considered to be a compact representation when it is not uniformly or randomly distributed across all possible values of that color space axis (for instance we can see that the color space representation for chrominance intensity 1 and chrominance intensity 2 occupy a very small area of the full color range in Figure 2, indicating that skin has a compact representation for those two axis in YCbCr, whereas the red, green, and blue axis of the RGB histogram are more uniformly distributed across the possible intensity values). The minimum value for all color spaces is 0, whereas the maximum value is typically either 1 (HSV color space) or 256 (RGB and YCbCr color spaces).

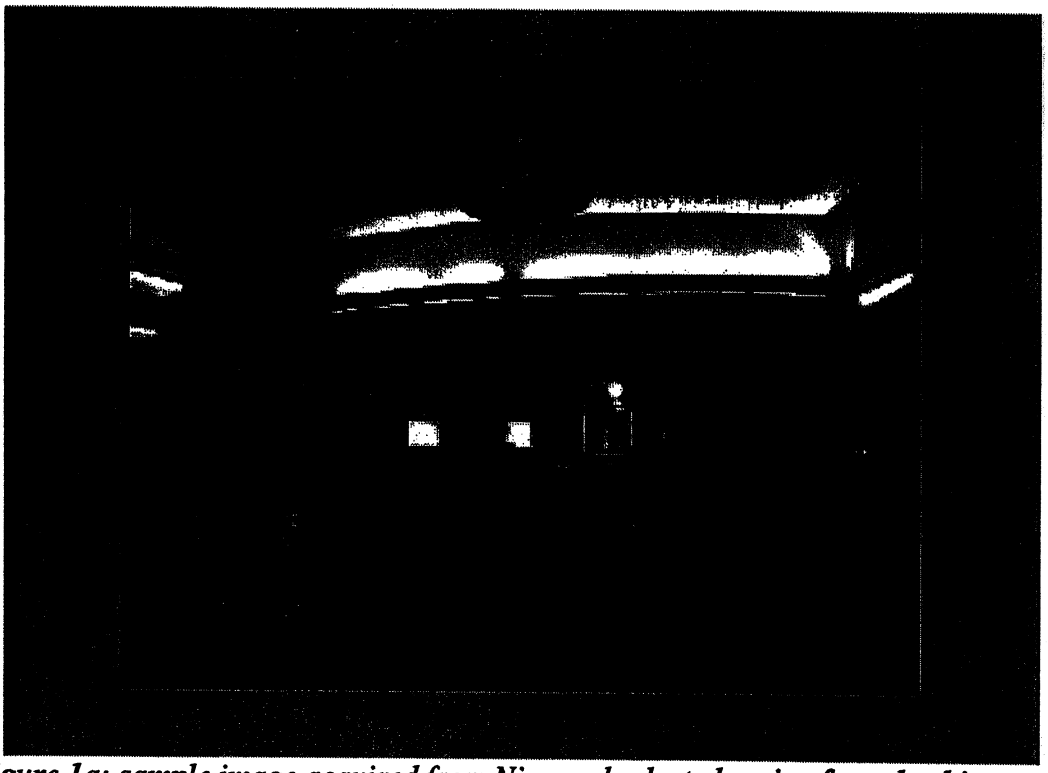


Figure 1a: sample image acquired from Nico and selected region for color histogram analysis (box on the face of the rightmost figure)

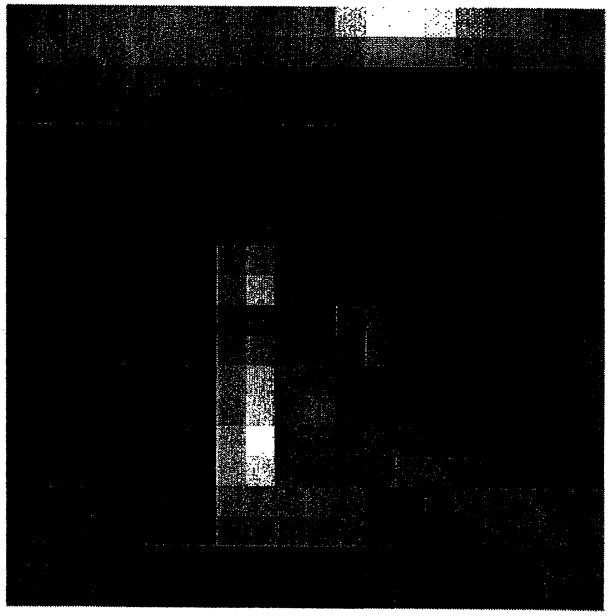


Figure 1b: zoom of region selected in 1a

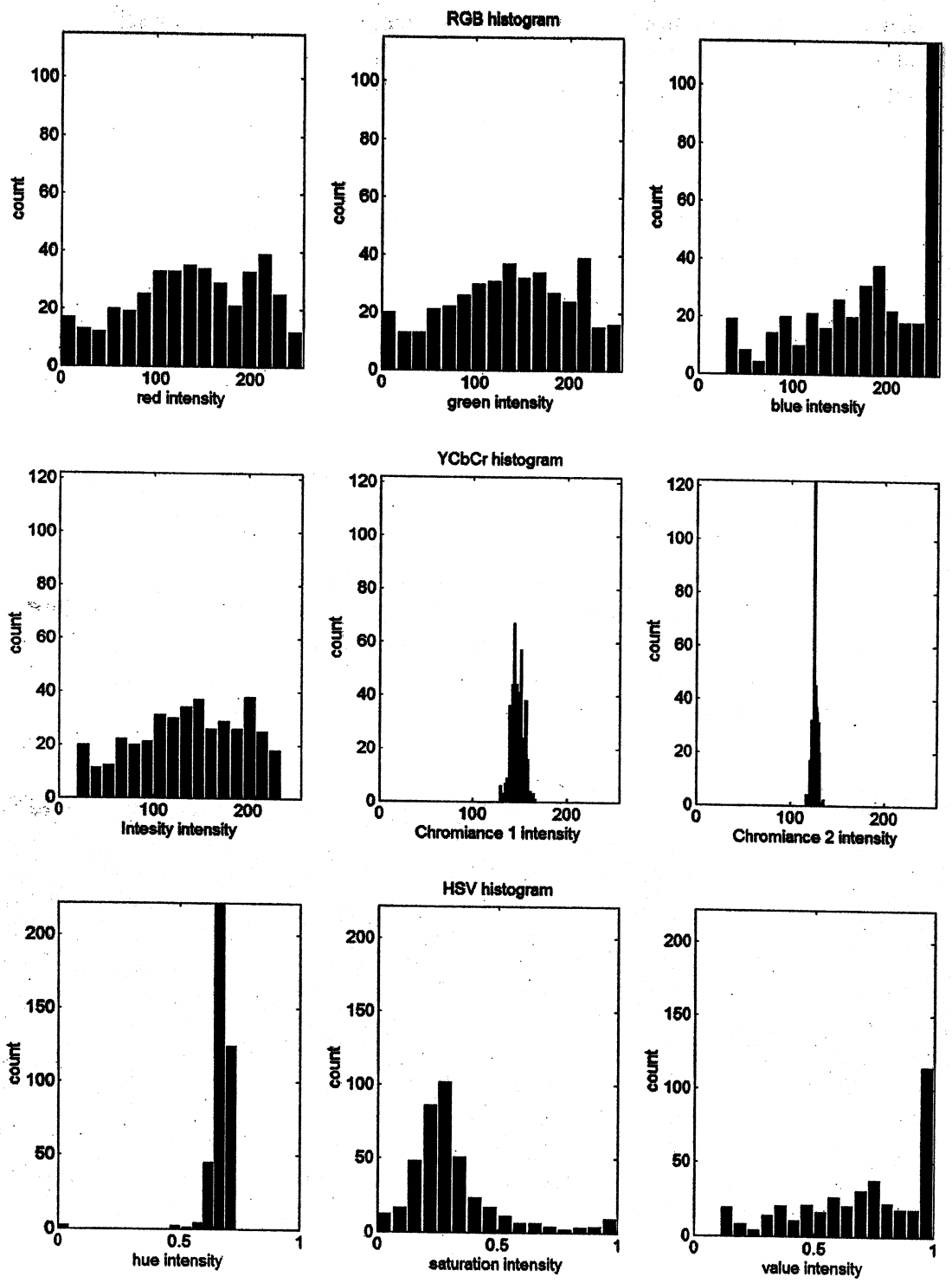


Figure 2: RGB (top), YCbCr (middle) and HSV (bottom) color histograms for the selected region in Figure 1.

As evident by inspection of Figure 2, the distribution over all the axes of RGB, unlike YCbCr and HSV, was uniform through its possible intensity values. For this reason we believed that the RGB color space would be insufficient for building a skin discrimination system that could segregate skin from non-skin and did not investigate it further.

YCbCr and HSV both demonstrated clear peaks in non-intensity based color space axes. For this reason, the axis corresponding to pure grayscale intensity was removed, and the remaining two axes of both YCbCr and HSV were used to build Gaussian probability models of the form:

$$p(x_1, x_2) = \frac{1}{\sigma_a \sigma_b 2\pi} e^{\frac{-(x_1-\mu_a)^2}{2\sigma_a^2} + \frac{-(x_2-\mu_b)^2}{2\sigma_b^2}} \quad (1)$$

where $p(x_1,x_2)$ gives the probability that a particular pixel with color space coordinates x_1 and x_2 corresponds to skin. μ_a and μ_b are the expected means of skin color in each color axis, and σ_a and σ_b are the corresponding standard deviations for the distribution.

By investigating the color distributions of a number of skin locations in images, we derived a good estimate for the parameters of the Gaussian skin probability model for both the YCbCr color space (μ_a=145, σ_a=7.5 for axis 1, and μ_b=127, σ_b=3.5 for axis 2) and for the HSV color space (μ_a=0.675, σ_a=0.0875 for hue, and μ_b=0.25, σ_b=1.0 for saturation). Applying these parameters yielded skin probability maps of the form shown in Figure 3.

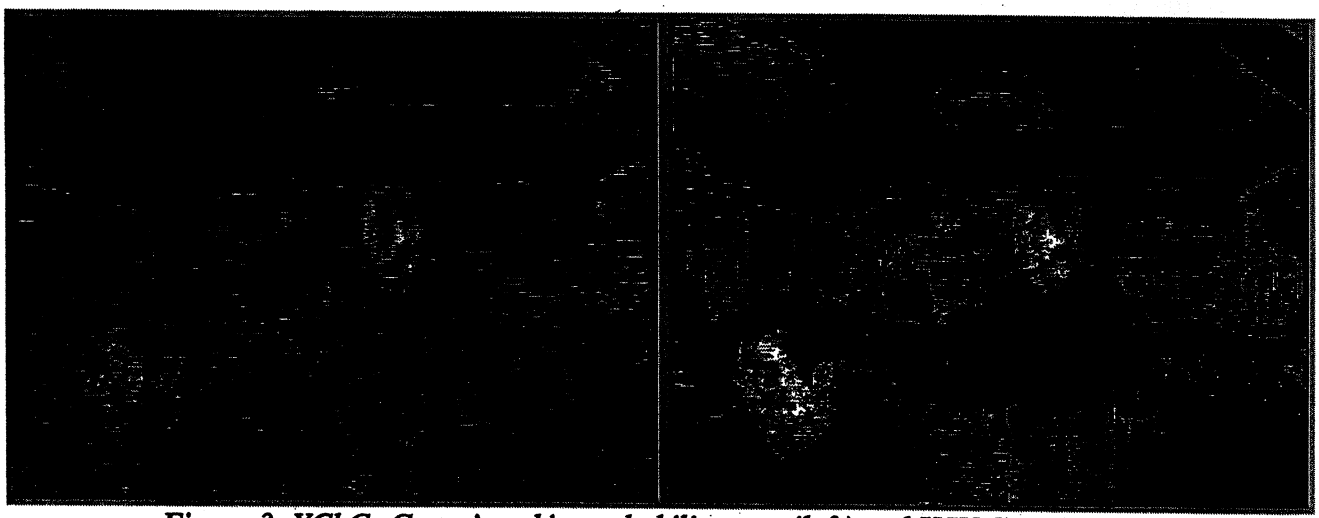


Figure 3: YCbCr Gaussian skin probability map (left) and HSV Gaussian skin probability map (right)

Note that the HSV skin probability map in Figure 3 yields much sharper skin information, but that the noise is higher than that of the YCbCr skin probability map. Also notice that HSV receives false positives for the khaki pants worn by the rightmost figure in Figure 3, an effect that is not noticeable in YCbCr. For these reasons we

combine the two color schemes by multiplying their corresponding probability maps and normalizing:

$$p(\mathbf{x}) = \sqrt{p_{YCbCr}(\mathbf{x}_{YcbCr})\, p_{HSV}(\mathbf{x}_{HSV})} \tag{2}$$

This yields the combined skin probability map shown in Figure 4:

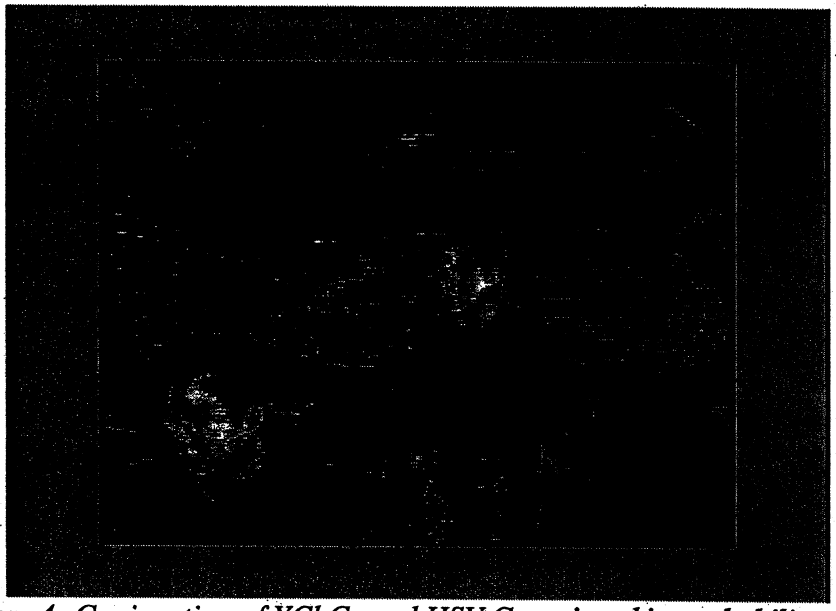


Figure 4: Conjunction of YCbCr and HSV Gaussian skin probability maps

This skin probability map does not work well when the intensity becomes saturated. For instance if we apply (2) to the image shown in Figure 1, we obtain Figure 5:

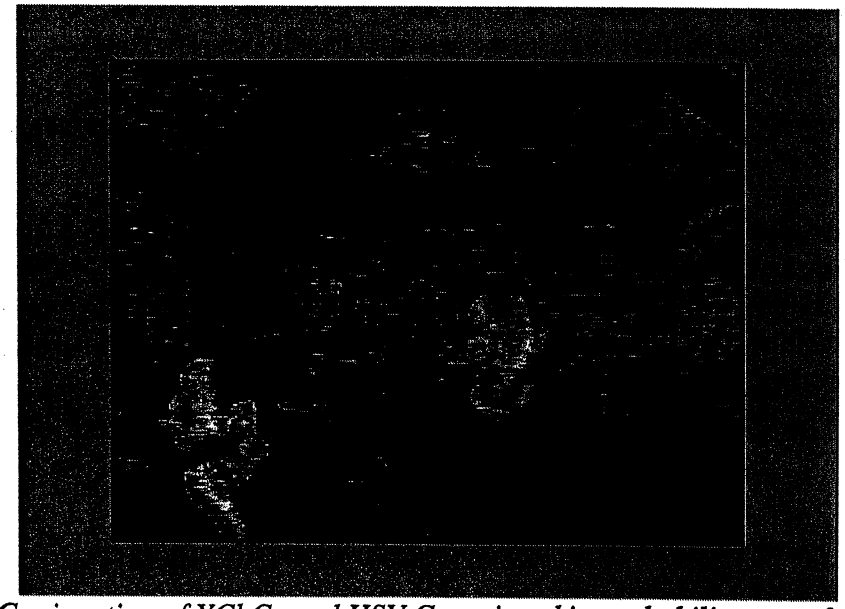


Figure 5: Conjunction of YCbCr and HSV Gaussian skin probability maps for the image shown in Figure 1. Note the error on the forehead of the rightmost subject.

The large missing segment in the forehead of the subject on the right of Figure 5 is black as the result of intensity overflow (and subsequently ill-defined non-luminance data).

The results of attempting to localize skin using color histogram information combined with Gaussian modeling were not wholly satisfying. In particular, the application of (1) and (2) to images yielded probability maps that were unstable in the vicinity of certain lighting effects (such as the fluorescent lights on the ceiling of the figures). In addition, noise in both the background and in skin regions was fairly heavy. These problems, taken together, suggested that a simple Gaussian model would not be sufficient for detecting skin. For this reason we looked for better models for skin detection.

2.2. Skin detection by training neural networks on image pixel color

Using a stand-alone graphics editing program, we manually marked the regions corresponding to skin for a pair of images (Figure 6).

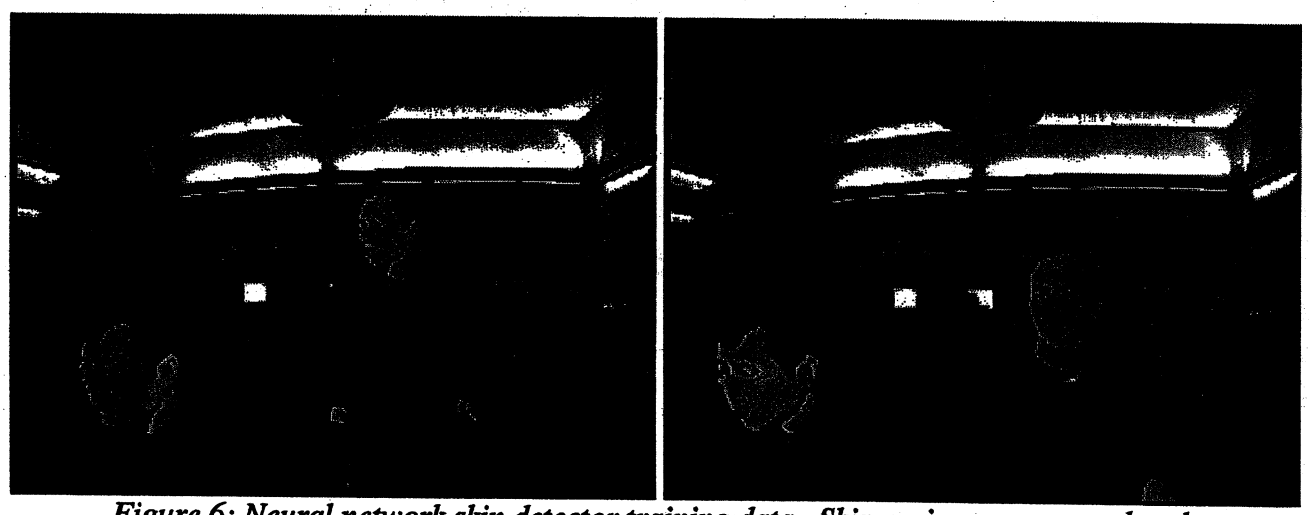


Figure 6: Neural network skin detector training data. Skin regions correspond to the lightly marked areas. Non-skin areas are not altered. Many of the skin detection techniques we tried picked out the khaki pants (left) as a face color, so we included this data to eliminate this problem. On the right is a typical image that was also included. Only two data sets were used in this analysis.

The images shown in Figure 6 were deconstructed into a set of flesh and non-flesh pixels. As we desired to avoid the computational overhead associated with color conversion, the original RGB color vector for each pixel was used as an input, testing against a binary output (1 for flesh and 0 for non-flesh), in a feedforward backpropagation neural network. Likewise, to save computation, we used only 5 hidden nodes (Figure 7).

The transfer function used in this neural network was the logsig function:

$$\text{logsig}(x) = \frac{1}{1+e^{-x}} \tag{3}$$

Levenberg-Marquardt (Press et al., 1993) was used for the network training function, gradient descent with momentum (learning rate 0.01, momentum 0.9) was used for the

weight learning function, and 100 iterations were used for training. Network recognition of skin versus non-skin was set at a threshold of 0.5 on the output node range of (0,1). This resulted in the detection shown in Figure 8, with corresponding statistics shown in Table 1.

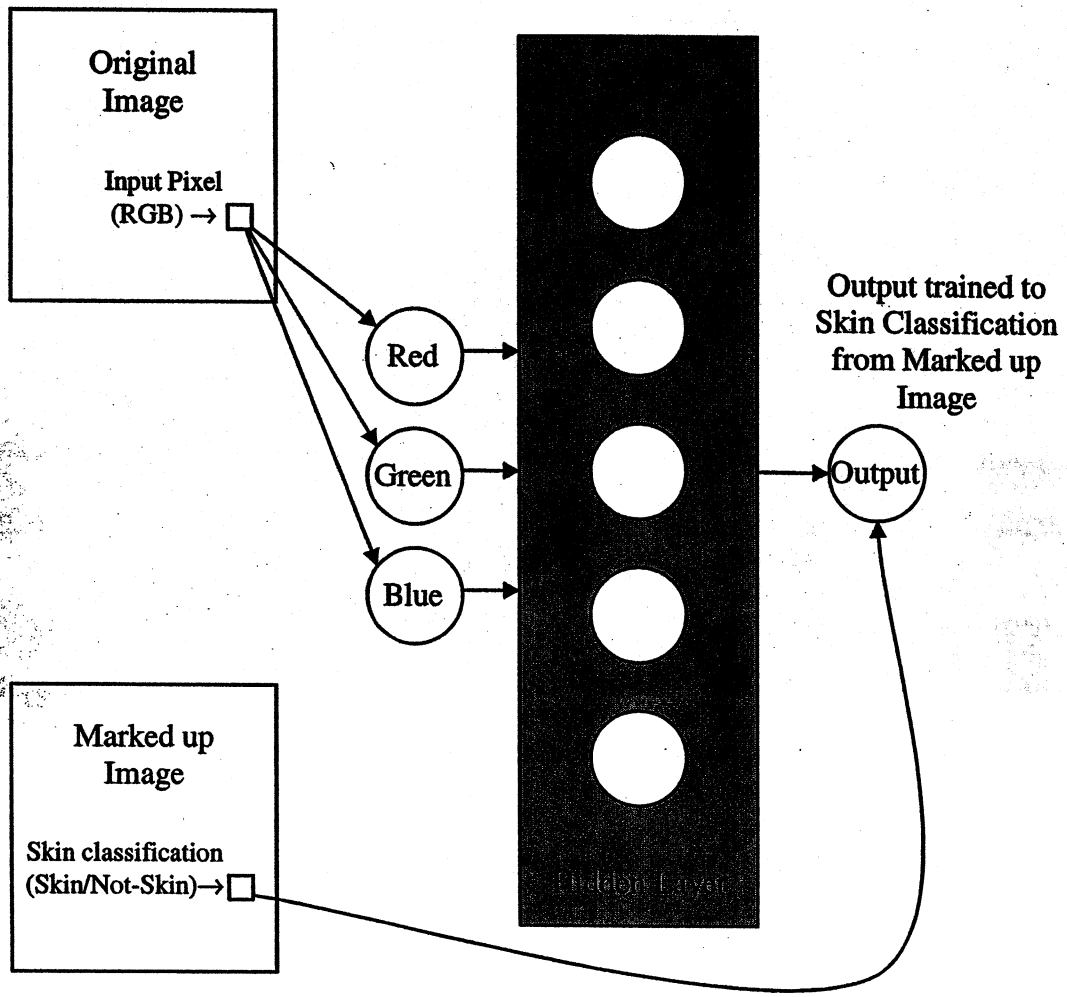


Figure 7: Pixel-level neural network architecture. Every pixel of the two training images shown in Figure 6 is used to train a neural network with 5 hidden nodes. The inputs are the RGB values of each pixel. The training output is the corresponding designation of skin or not-skin as determined by the manually marked up images of Figure 6.

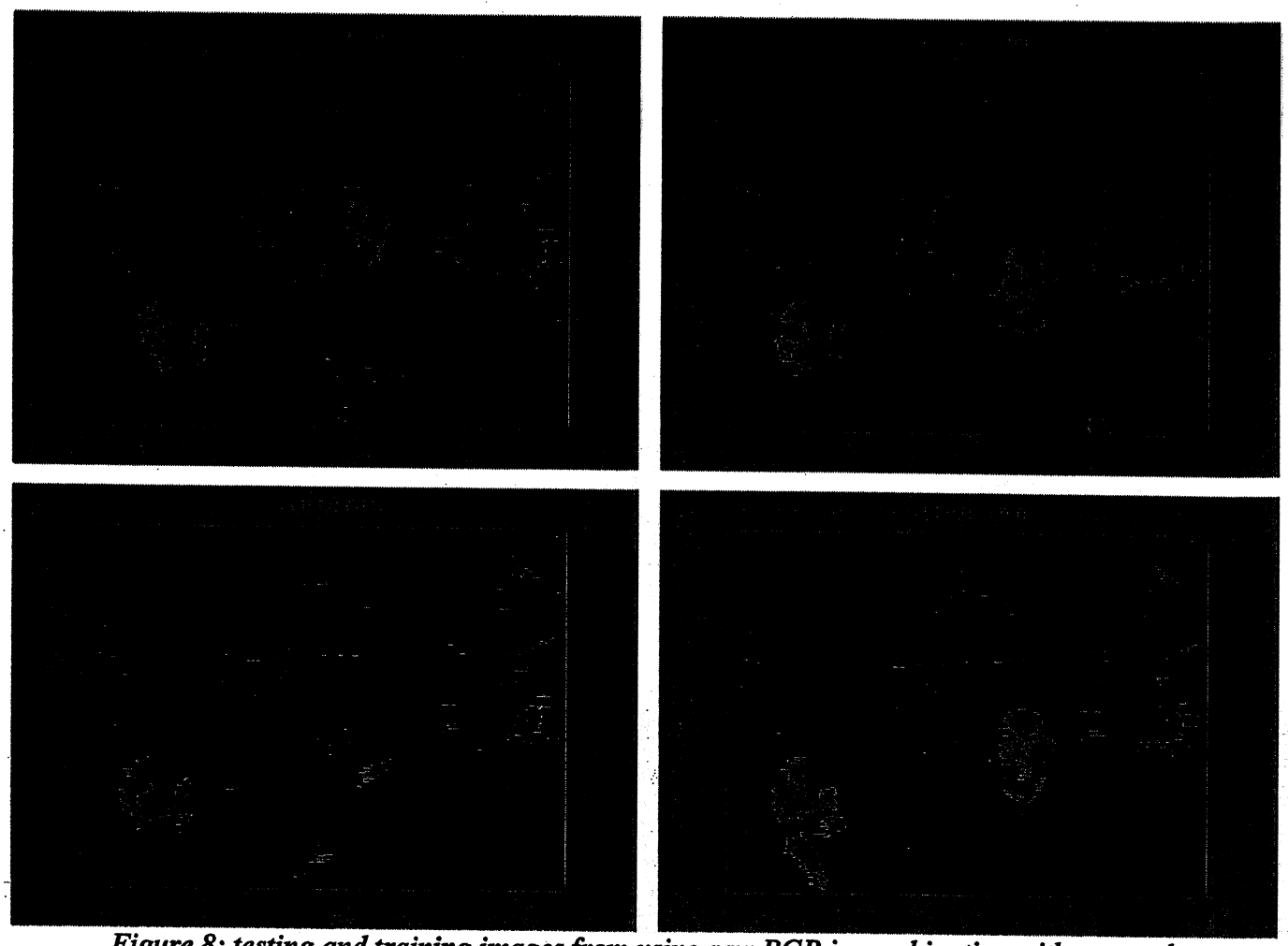


Figure 8: testing and training images from using raw RGB in combination with a neural network. Images are darkened so that the brightest areas correspond to skin pixels. The two top images correspond to the training data shown in Figure 6; the two bottom figures are test images. Recognition is fair. Note that, in the lower left hand corner, the skin detector completely fails to detect the slightly shadowed flesh of the rightmost subject.

	Training Data Set	Testing Data Set
Accuracy (Skin)	52.2%	42.2%
Accuracy (Non-Skin)	99.6%	99.6%
Accuracy (Total)	98.0%	97.4%
# skin pixels / image	2371	2988
# non-skin pixels / image	74430	73812
# total pixels / image	76800	76800

Table 1: NN performance on skin detection using only RGB pixel information

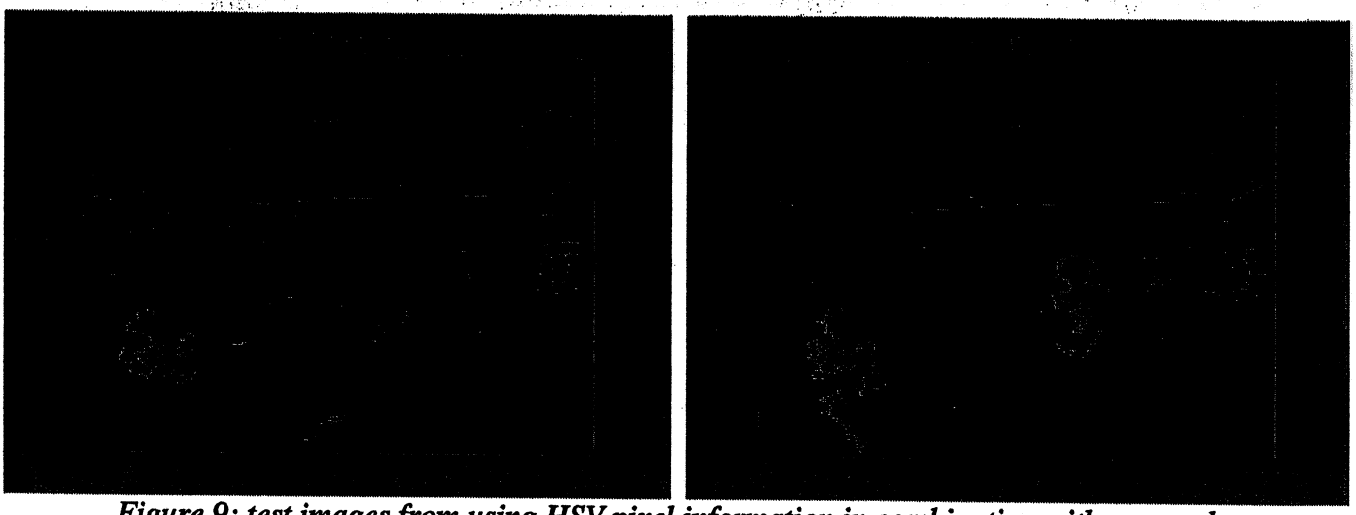


Figure 9: test images from using HSV pixel information in combination with a neural network. Note that there is almost no difference between this output and the output obtained by using RGB pixel information only (Figure 8).

	Training Data Set	Testing Data Set
Accuracy (Skin)	51.0%	40.7%
Accuracy (Non-Skin)	99.5%	99.6%
Accuracy (Total)	98.0%	97.3%
# skin pixels / image	2371	2988
# non-skin pixels / image	74430	73812
# total pixels / image	76800	76800

Table 2: NN performance on skin detection using only HSV pixel information. Again, note the similar performances on both skin and non-skin between results obtained using HSV pixel information and results obtained using RGB pixel information (Table 1)

Compared with the performance of the Gaussian probability maps (Figure 5), we can see that the ability to distinguish skin from non-skin is vastly improved (Figure 8). There is little noise in the background (99%+ accuracy), but performance on detecting skin is only fair (52.2% on the training data, and 42.2% on the test images). This is problematic because, at this level of accuracy, large skin structures such as a face or a hand could be broken into several smaller patches, potentially causing malfunction in subsequent algorithms for face detection.

We want to see if using another color space, such as HSV, can improve our results because, as we noted earlier while studying the Gaussian probability maps, HSV provides very sharp segmentation of skin. Also, there has been extensive research in the appropriate choice of color space in skin detection (Zarit et al, 1999), suggesting that the choice of color space can play an important role in skin detection performance. However, as far as neural nets is concerned, the use of even a "more appropriate" color space does not seem to influence results as seen by the almost identical skin determinations in images (Figure 9 compared with Figure 8) and statistical results (Table 2 compared with Table 1).

Finally, we tested using an input vector consisting of RGB, HSV, *and* YCbCr channels combined. Though each of these three color spaces is transformable to the others, we wanted to make sure that, due to the low number of hidden nodes used, some key *a priori* information was not somehow being overlooked. After conducting this experiment, however, we obtained results extremely similar to the results obtained by using the RGB or HSV color space alone (results not shown). It is questionable whether, in an adaptive technique, the choice of color space plays any role at all.

2.3. Skin detection by training neural networks on neighborhoods of color

We now consider the addition of local information to each pixel's determination of flesh versus non-flesh. In order to do this, we return to the RGB model and augment the input color stream with the color stream of a Gaussian blurred (10 pixel square, standard deviation of 5 pixels) version of the image. This proved to be exceptionally helpful, as shown by the images in Figure 10 and the results in Table 3. Note that the color oversaturation problem seen previously still remains (bottom right image of Figure 10, forehead of the rightmost subject). This might be correctable by using more local information. For example, we could use better choices for Gaussian blurring or incorporate gradient information (Sandeep and Rajagopalan, 1999) in order to increase the extent of neighborhood information available to the neural network at any given pixel.

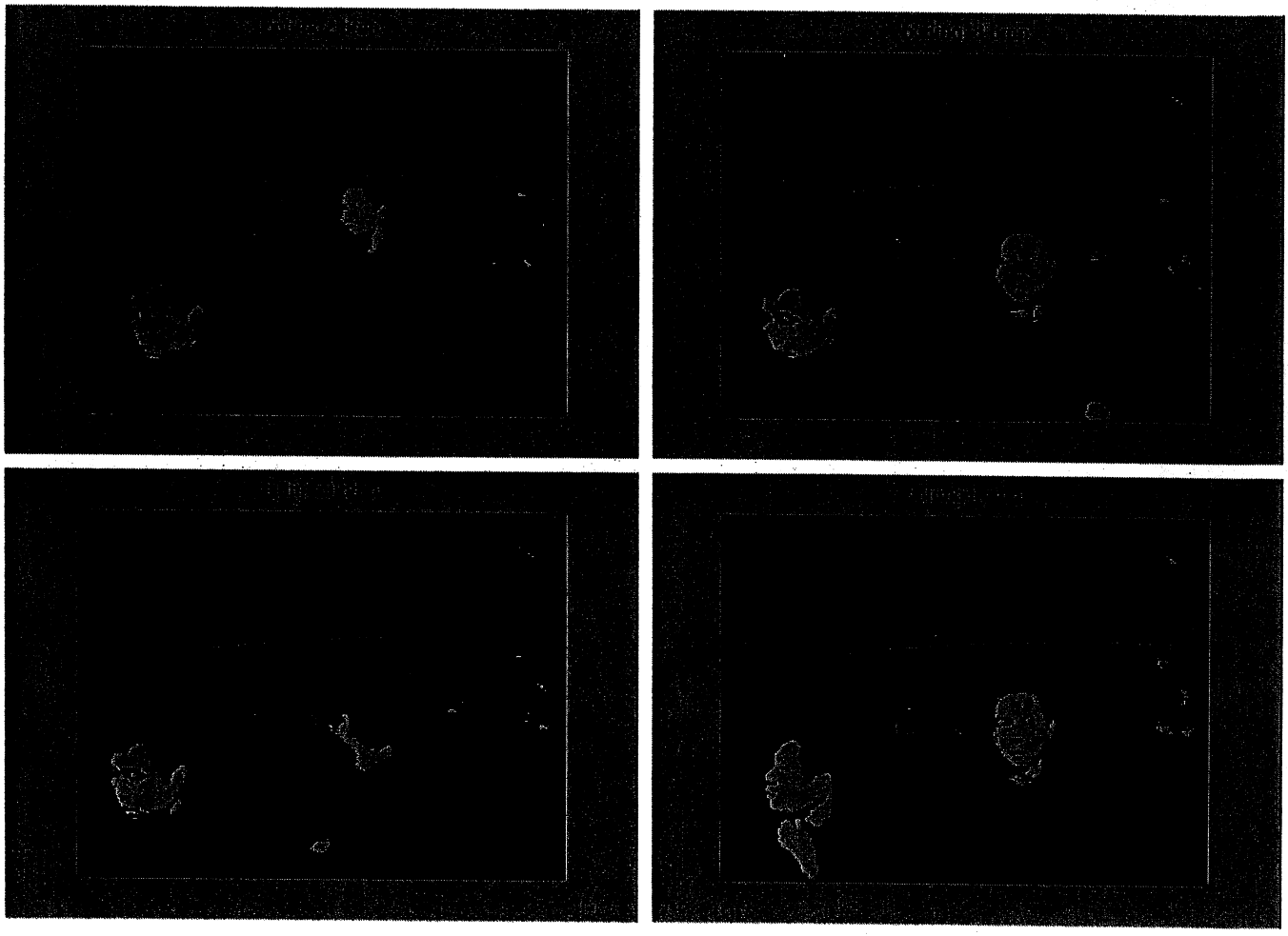


Figure 10: RGB channel combined with RGB Gaussian blurred channel. Note the finer accuracy and better characterization of features in comparison to the single pixel method.

	Training Data Set	Testing Data Set
Accuracy (Skin)	79.4%	71.7%
Accuracy (Non-Skin)	99.7%	99.7%
Accuracy (Total)	99.0%	98.7%
# skin pixels / image	2371	2988
# non-skin pixels / image	74430	73812
# total pixels / image	76800	76800

Table 3: NN performance on skin detection using RGB pixel and neighborhood information

We should note that the output is highly dependent on the subjective markup of flesh tone as training input. For instance, if we didn't pay quite so close attention to the features of faces, and used a coarser input (Figure 11), we would get the results shown in Figure 12 and Table 4.

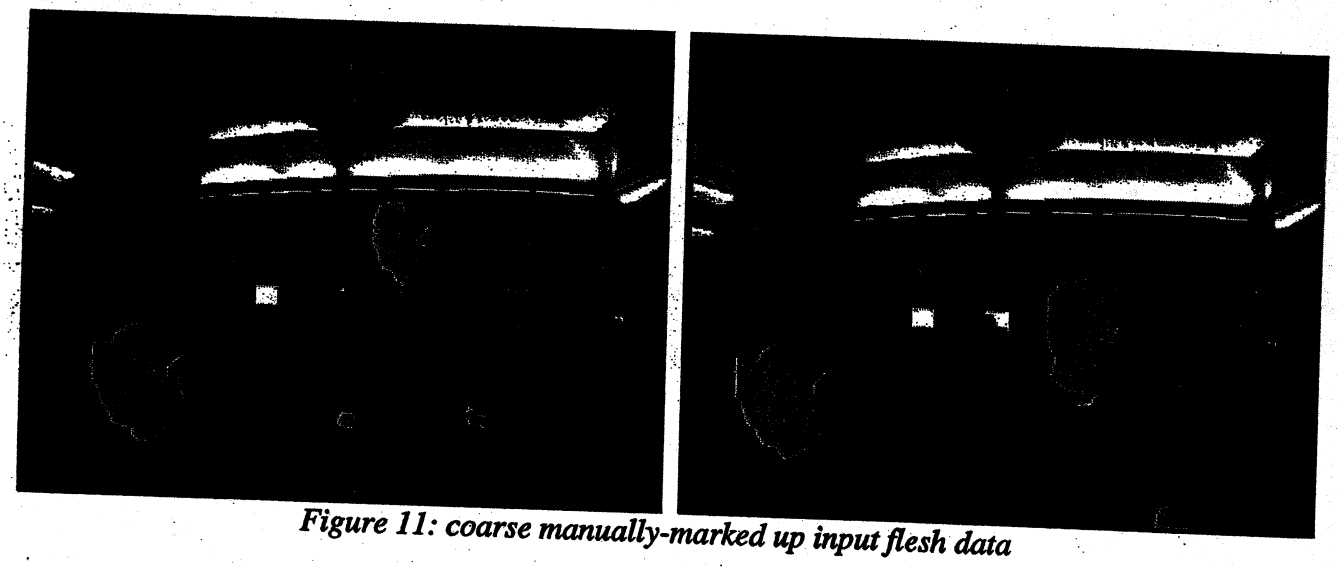


Figure 11: coarse manually-marked up input flesh data

	Training Data Set	Testing Data Set
Accuracy (Skin)	80.7%	80.2%
Accuracy (Non-Skin)	99.6%	99.5%
Accuracy (Total)	98.9%	98.7%
# skin pixels / image	2964	3211
# non-skin pixels / image	73837	73589
# total pixels / image	76800	76800

Table 4: NN performance on skin detection using RGB pixel and neighborhood information on a coarse input (Note: results are not necessarily comparable to previous results as the performance images have changed)

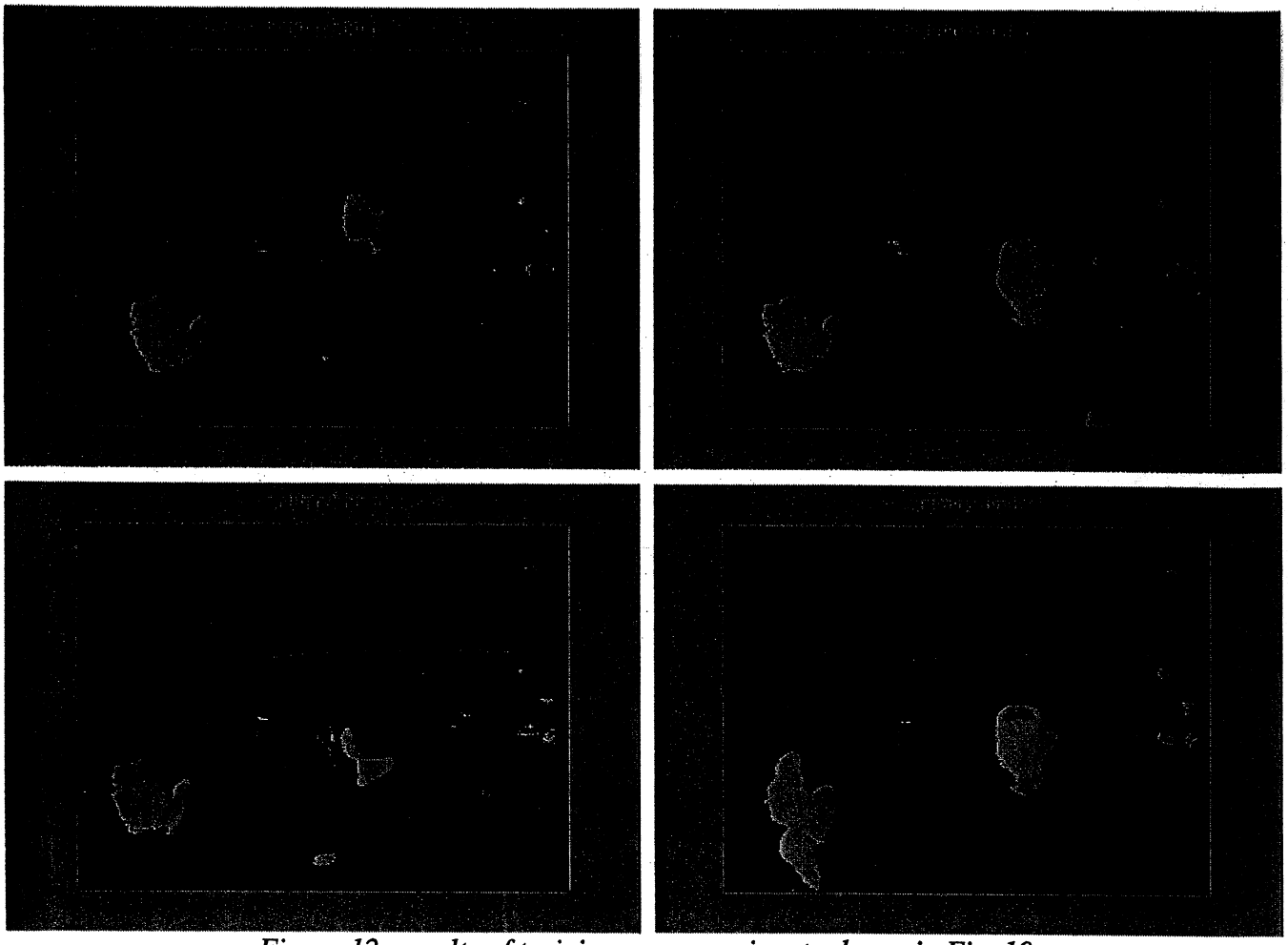


Figure 12: results of training on coarse inputs shown in Fig. 10.

Since we are interested in extracting head orientation, the use of coarse input is actually more appropriate. All subsequent analyses use the coarse RGB neighborhood model.

We should note that designation of skin versus non-skin is determined by a threshold on the output of our neural network. Without a threshold we get Figure 13.

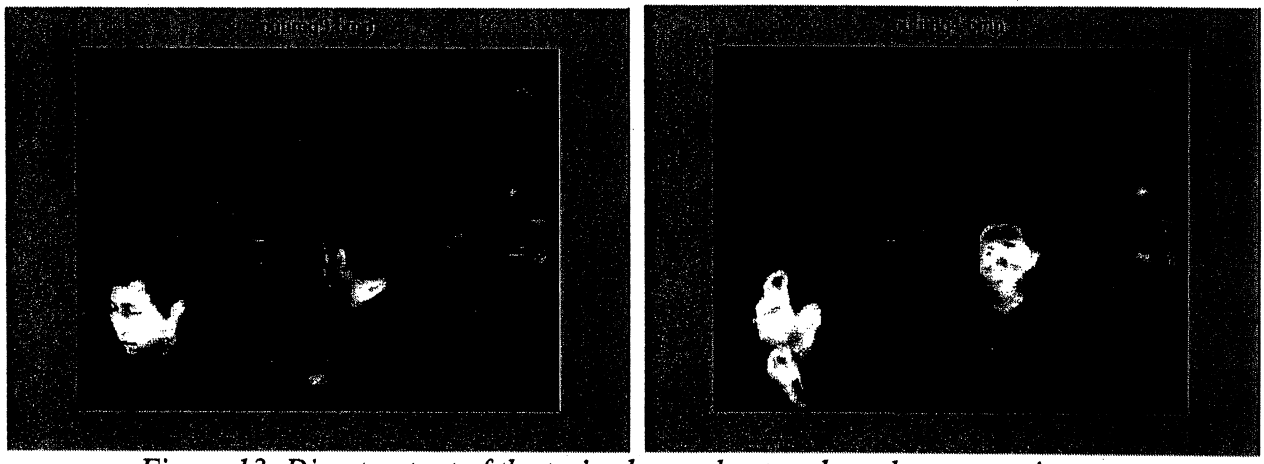


Figure 13: Direct output of the trained neural network on the two test images.

3. HEAD DETECTION

Once we have discovered potential sites of skin, we can examine these sites in order to determine if the skin sites correspond to faces. In order to do this we markup images as we did previously, only this time we only mark faces (Figure 14).

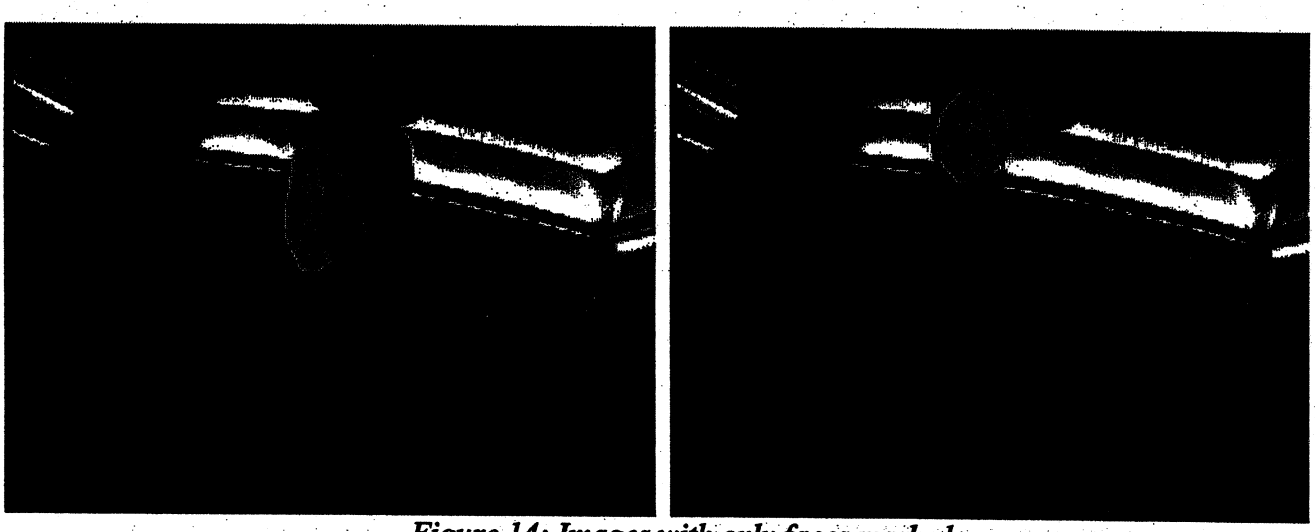


Figure 14: Images with only faces marked.

We markup 85 images, a much larger sample than was used in skin detection, and split the images between testing data and training data. The reason why so many more images were incorporated into this step was because we are no longer working at the local level, but instead, at a higher level of classification. For each image we perform the following pre processing algorithm:

1. Apply the skin-detecting neural network to an image.
2. Use 8-connected region growing to pick out areas corresponding to patches of skin (Figure 15, top).
3. Throw out any regions with less than 200 total pixels. This is done because beneath this threshold, images are too small to be accurately detected and because they most likely correspond to noise (Figure 15, bottom).
4. For every region passing the threshold test in step 3, send the minimal bounding box incorporating the region to a feature extractor.
5. The feature extractor takes the image and segments the image into a series of overlapping rectangles. All pixels in each rectangle are averaged together in order to build a single coefficient. For each level L in the system, the image was segmented into a L by L grid of rectangles. We used a level of 3 for this set of experiments (Figure 16).

After all data was processed, we split the sets of coefficients evenly into testing and training data. Correct output was determined by checking for overlap in the marked-up face images discussed previously. These coefficients were passed into a 5-hidden

node neural network, with the same parameters as were used in skin-detection, and trained for 2,000 epochs. Finally, the output of the neural network was once again compared against a threshold (0.5) in order to determine if it was classified as skin or non-skin. The performance of the network during training, however, was extremely unstable, resulting in training accuracies ranging from 65% to 100%, and testing accuracies that ranging from 40% to 97%. In order to compensate for these effects, the neural network was run in a loop until it met a threshold of at least 95% accuracy in training and at least 90% accuracy in testing.

By looping on a measure of the testing data, we are effectively *training* on the testing data. We need to find an independent source for comparison, or our results would not be valid. We do this by examining 201 images that were not included in the original training or testing sets. For each image, we segment skin regions as described above, and then allow the neural network to classify each region. The results from this experiment were an accuracy of 86%, starting from an original training accuracy of 100% and testing accuracy of 97.2%.

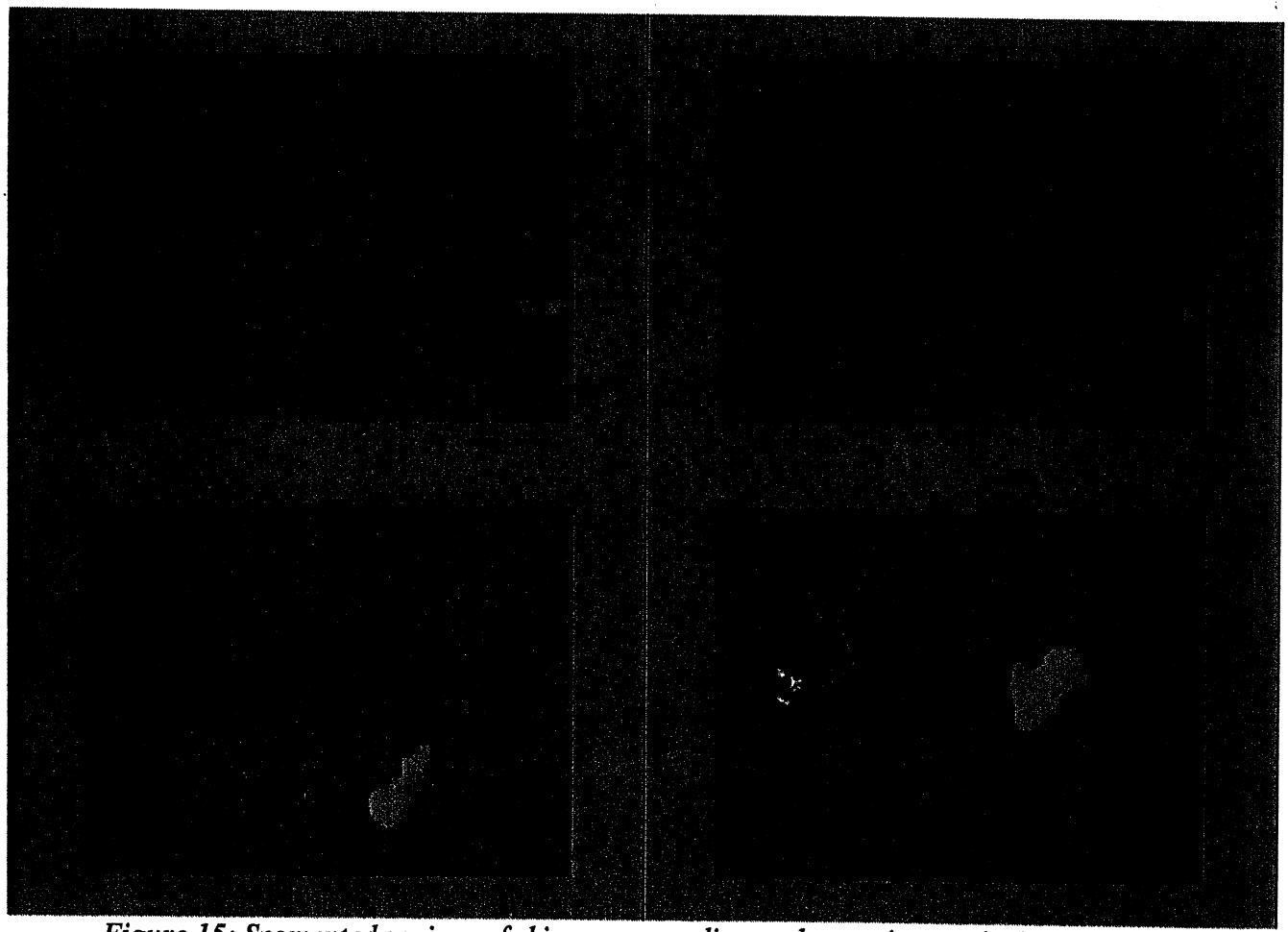


Figure 15: Segmented regions of skin corresponding to the two images in Figure 14. The top figures here are segmented regions before removing regions failing to meet an area threshold, and the bottom figures the reduced regions.

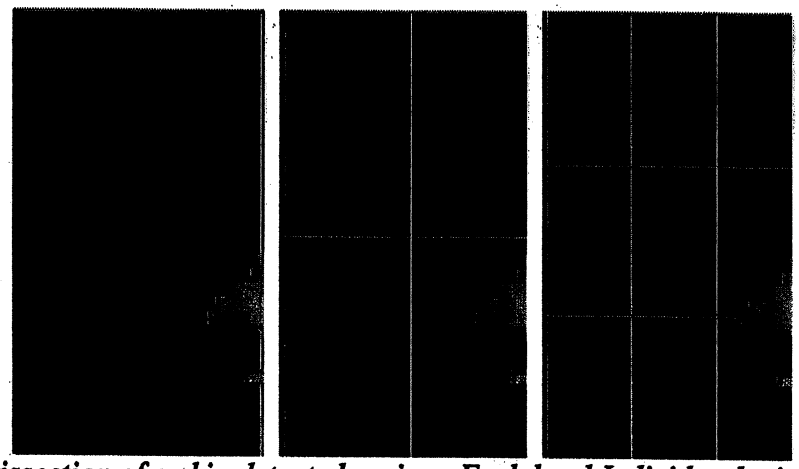


Figure 16: Dissection of a skin detected region. Each level L divides the image into an L by L grid. Each rectangle of the grid is locally averaged in order to provide feature cues to a neural network.

4. HEAD ORIENTATION

Now that the preprocessing components have been built, we can begin head orientation. Ideally we would like a system that can independently identify real valued components of pan and tilt. However, in order to build such a system there needs to be some accurate method of measuring accuracy. Without a stand alone device to judge orientation, we must rely on more primitive methods.

Using the same methods as for head detection, we assign, manually, to each image that is to be used in training or testing, a category number. This number corresponds to the following orientations (from the viewpoint of the camera, Table 5).

Code	Subject's Head is Directed
1	left
2	Forward
3	Right
4	Down

Table 5: Coding for head orientation

The outputs were first vectorized so that the i^{th} entry of the output was a 1 if the output code was i. The only difference between this step and head detection, besides the fact that the output format was different, was that the Gaussian map for detection (shown in Figure 16) used 4 levels rather than 2. The motivation behind this was that it should be able to, at some point, combine head/face detection and head orientation detection in order to gain more throughput at execution.

Though the setup was fairly simple, there was a lot of instability in the network convergence (as in Section 3). The network typically trained in upwards of 99% accuracy, and had a orientation accuracy, based on the above signatures, of 73.8%.

5. DISCUSSION

Many things in this experiment could be improved. Our method for detecting skin seems to be fairly robust, but head localization and head orientation seem insufficient. Using a multi-scale average technique for input into the a neural network for both head orientation and head localization may be unwarranted. For instance, we could use the face detection/head localization step to further refine the image of the face, eliminating spurious signals such as the neck and ears, by using a larger, more powerful model of local information, but similar to the one employed in skin detection.

It is important to note that all the images and data collected for this experiment revolved around very few subjects, and a very constrained environment. Not until the project has been extended to include actually movement on the part of the observer and a multitude of people of all different shapes, sizes, and pigmentations, will this project be complete.

6. CONCLUSION

We have presented a model for head orientation that uses local color space information to successfully detect skin from non-skin, and, less successfully, to localize the head and coarse orientation of the head. Future studies are warranted, as this is continuing work.

REFERENCES

1. Gee A & Cipolla R (1994). Determining the gaze of faces in images. *Image and Vision Computing*, v.12, n.10, pp.639-647.
2. Horprasert T, Yacoob Y, & Davis L (1996). Computing 3-D Head Orientation from a Monocular Image Sequence. *International Conference on Face and Gesture Recognition*, pp.242-247.
3. Zhao, Pingali, & Carlbom (2002). Real-time Head Orientation Using Neural Networks. *IEEE International Conference on Image processing,* I.1, pp.297-300.
4. Stiefelhagen R, Yang J, & Waibel A (2001). Tracking Focus of Attention for Human-Robot Communication. *Proceedings of the IEEE-RAS International Conference on Humanoid Roboti,* 9(2):257-265.
5. http://www.ascension-tech.com/products/flockofbirds.php, obtained December 15, 2003.
6. Pappu, R, & Beardsley PA (1998). A Qualitative Approach to Classifying Gaze Direction. *Proceedings of the Third IEEE International Conference on Automatic Face and Gesture Recognition*, pp.160-165.
7. Wu, Y. &Toyama K (2000). Wide-Range, Person- and Illumination-Insensitive Head Orientation Estimation. *Fourth IEEE International Conference on Automatic Face and Gesture Recognition*, pp.183-188.
8. Zarit B, Super B, & Quek F (1999). Comparison of Five Color models in Skin Pixel Classification. *Proc. of Int. Workshop on Recognition, Analysis, and Tracking of Faces and Gestures in Real-Time Systems, IEEE Computer Society,* , pp. 58-63.
9. Sandeep K and Rajagopalan AN (1999). Human Face Detection in Cluttered color Images Using Skin Color and Edge Information. http://citeseer.nj.nec.com/557854.html

10. Press W, Flannery BP, Teukolsky SA, and Vertterling WT (1993). *Numerical Reipes in C: The Art of Scientific Computing*. Cambridge, UK: Cambridge University Press.

Self-supervised Learning of Saccade Control with a Feed-forward Neural Network

Hao Wang
Yale University, Department of Computer Science
New Haven, CT 06511

Abstract
An approach to learn fast and accurate saccade control for a humanoid robot with a feed-forward neural network is presented. We solve the visual tracking problem by using a competent image correlation algorithm. The inconsistency in motor coordinates is handled by scaling. Training of the feed-forward neural network is based on error back-propagation learning. Simulation results are presented to demonstrate the effectiveness of this approach.
Keywords – saccade control, sensorimotor coordination, visual tracking, error correction learning, feed-forward neural network.

1. INTRODUCTION

Saccades are of current central interest in the field of active vision [1]. For instance, saccades have been used to attend to salient features in the visual field and pursuit of moving objects [2]. Yale Social Robotics Lab is building a humanoid robot called Nico. Nico has two eyes, each of which has two degrees of freedom. Each eye consists of two small color CCD cameras, one with a wide-angle low-resolution field of view, and the other with a narrow-angle high-resolution field of view. If it is to obtain more details about a visual target in the wide-angle low-resolution field of view, Nico must center the target in this view so that the target shows up in its narrow-angle high-resolution view. Therefore, saccade control is an integral component of Nico's vision system. Saccade control belongs to the broad field of sensorimotor coordination [3]. We shall show how a feed-forward neural network can be utilized to learn fast and accurate saccade control for Nico.

2. TASK OVERVIEW

In this paper, we consider the robotic task of learning saccade control for one of the two wide-angle low-resolution cameras. This task can be formulated as learning a saccade control function f that takes four arguments:

p, t – the current pan and tilt motor coordinates of the camera
r, c – the row and column of the center of the object of interest (specified by other modules of the robot, e.g. attention system) within the input image

and produces two results:

dp, dt – the change in the pan and tilt motor coordinates of the camera that will center the object of interest within the field of view of the camera.

Figure 1 shows a schematic representation of the saccade control learning system architecture. The system can be decomposed into four major components: visual tracking subsystem, saccade generator, saccade control function, and a training program.

The visual tracking subsystem takes input images from one of the wide-angle low-resolution cameras, and locates a visual target after a saccade. The saccade generator controls the pan and tilt eye motors to generate saccades. The training program is the central component of the whole system. It controls and coordinates with the other three components to accomplish the task of learning a satisfactorily accurate saccade control function *f*.

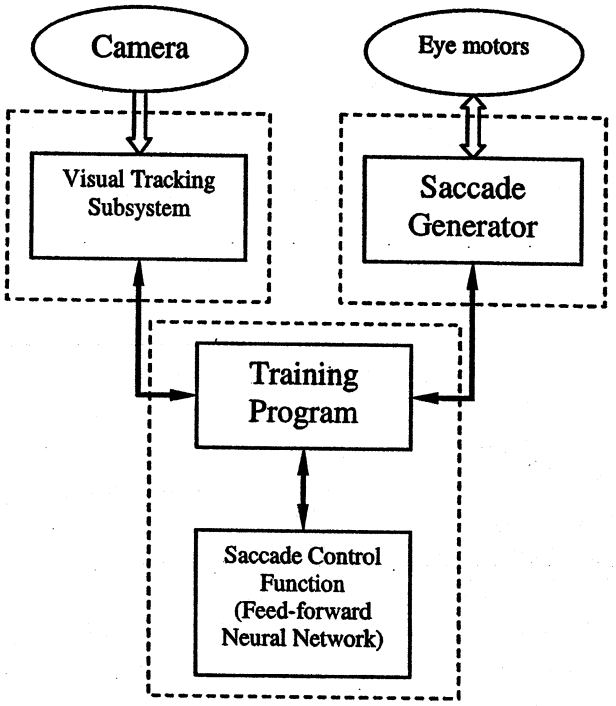


Figure 1: Schematic representation of system architecture. Dashed boxes indicate processes. Processes communicate through message passing.

A typical learning trial proceeds as follows:

1. The visual tracking subsystem takes an input image, which is referred to as the *pre-saccade image*. The training program randomly generates a row coordinate r and a column coordinate c. A small area centered at (r, c) in the pre-saccade image is chosen as the *visual target*. The coordinate of a visual target is taken to be the coordinate of its center.

2. (r, c), along with the current pan and tilt motor coordinates (p, t), are passed to the saccade control function f to generate the change (dp, dt) in the pan and tilt motor coordinates.

3. The new pan and tilt motor coordinates (p', t'), where $p' = p + dp$ and $t' = t + dt$, are sent to the saccade generator to perform a saccade.

4. After the saccade, the visual tracking subsystem takes a second input image, which is referred to as the *post-saccade image*, and locates the row and

column coordinates (r', c') of the visual target in the post-saccade image. If the saccade were perfect, (r', c') would be the center of the post-saccade image.

5. The difference between (r', c') and the center of the post-saccade image is transformed to an error signal (dp', dt') in motor coordinates. In another word, the desired output of the saccade control function is assumed to be (dp'', dt''), where $dp'' = dp + dp'$ and $dt'' = dt + dt'$.

6. Finally, the error signal (dp', dt') is used to adjust the free parameters of the saccade control function f.

3. VISUAL TRACKING

Theoretically, in order to learn the saccade control function f, only the coordinates (r, c) and (r', c') of a visual target in pre-saccade and post-saccade images are needed. Coordinate (r, c) is chosen by the training program in pre-saccade image to specify the visual target, and is therefore readily available. After a saccade, the training program must be able to determine coordinate (r', c') of the visual target in the post-saccade image. This is called the *visual tracking* problem.

The simplest solution to the visual tracking problem is to determine (r', c') by hand. After each saccade, one could examine the post-saccade image, and determine the coordinate (r', c') of the visual target. This manual marking process must be performed once for every training sample. Obviously, if a large amount of training samples, e.g. 1000 training samples, are needed, the amount of labor incurred will be unaffordable. In addition, one of the goals of this project is to implement self-supervised, online learning, which does not require human intervention. Thus it is necessary to automate the process of determining coordinate (r', c') of a visual target in a post-saccade image. This automation is achieved through the visual tracking subsystem.

The visual tracking subsystem takes as input a pre-saccade image, a visual target location (r, c) in the pre-saccade image, and a post-saccade image, and returns the location (r', c') of the visual target in the post-saccade image.

There have been considerable efforts in visual tracking [4] [5]. There are two major challenges for visual tracking in this project. First, even a slight movement of the camera may cause variation in background luminance. Second, the geometric shape of the visual target may also change due to camera movement. Our solution to these problems is a trade-off of simplicity and reliability based on the particular constraints of our task at hand. First, the variation in luminance is accounted for by a color space conversion from RGB to HSI. In HSI color model, color information (hue) is separated from luminance information (intensity). The visual tracking subsystem primarily relies on hue information, which is largely independent of variation in luminance. In cases where a target is of nearly constant color, intensity information is used as a substitute. Second, due to the presence of a reasonably good estimate of the saccade control function (which will be discussed later), the search for a target in a post-saccade image could be conducted within a small search area, which alleviates the effect of changes in geometric shape of the target. In addition, a limited search area greatly increases the speed of visual tracking, although our application is not very time-critical.

More specifically, the RGB values of pixels in both the pre-saccade image and the post-saccade image are converted to HSI values. The hue values $H_{pre}(i, j)$ of a 40x30 area

around the visual target in the pre-saccade image are recorded. The hue values $H_{post}(i, j)$ of a 40x30 area around the estimated location of the visual target in the post-saccade image are correlated against those in the pre-saccade image. In mathematical terms, we seek to minimize the Manhattan distance of the difference of two image vectors:

$$\min_{x_0, y_0}\left(\sum_i\sum_j\left|H_{pre}(i,j)-H_{post}(x_0+i, y_0+j)\right|\right) \quad (1)$$

In cases where the 40x30 area around the visual target in the pre-saccade image is of nearly constant color, intensity information is used as a substitute formula. However, to compensate for changes in background luminance as the camera moves, the intensity vectors are normalized. Due to the normalization, formula (1) is no longer appropriate for determining maximum correlation. Instead, we see to maximize the cross-correlation of two image vectors:

$$\max_{x_0, y_0}\left(\sum_i\sum_j I_{pre}(i,j)\bullet I_{post}(x_0+i, y_0+j)\right) \quad (2)$$

A more sophisticated visual tracking subsystem may use these two methods at the same time, and adaptively combine the two results based on their reliabilities, which are indicated by the minimum Manhattan distance and maximum cross-correlation as given by (1) and (2), respectively.

One example of visual tracking is shown in Figure 2a and Figure 2b.

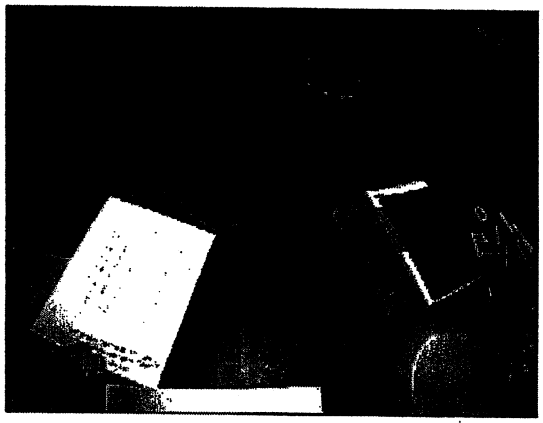


Figure 2a: The pre-saccade image. Visual target is the box at the bottom left corner.

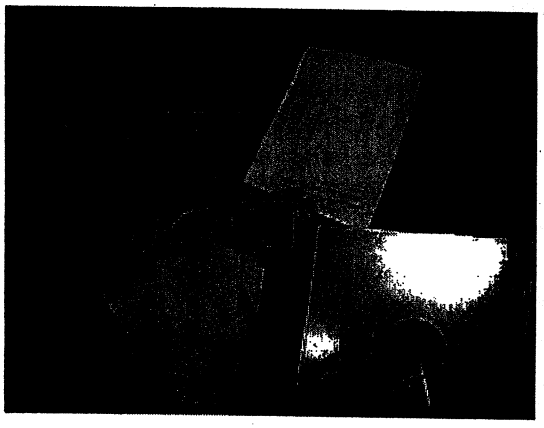


Figure 2b: The post-saccade image. Detected visual target is the box near the center of the image.

4. MOTOR CONTROL

A problematic property of the pan and tilt motors is: the same motor coordinate does not always lead to the same camera position. The inconsistency arises from hardware initialization. After the robot is powered on, a motor must be calibrated before it can be used. Depending on the initial motor position when the calibration begins, the same motor coordinate may lead to different motor positions in different runs of the robot. This difference, if not accounted for, will render the learned saccade control function useless after the robot is rebooted.

A remedy comes as the result of experimentation. The motor calibration process will first move the motor to its extremity in both directions, then position the motor half way in between, and return the limit to which the motor can move in either direction. Experimentation has revealed that although the limit returned by the calibration process may change significantly, the extreme positions, and thus the middle position of the motor largely remain the same.

The above observation leads to the use of 'proportional' motor coordinates instead of actual motor coordinates. The proportional coordinate p_r for the pan motor is defined as the ratio of the actual motor coordinate p to the limit of the motor l_p:

$$p_r = \frac{p}{l_p} \tag{3}$$

A similar definition applies to the proportional coordinate t_r for the tilt motor.

The saccade generator is in charge of the conversion between proportional and actual motor coordinates. Essentially, it provides a consistent motor control interface to the training program and the saccade control function.

5. TRAINING

The saccade control function f is represented by a feed-forward neural network. The network is trained using an error back-propagation learning algorithm.

Before error back-propagation could proceed, however, a fundamental problem has to be solved, i.e. to determine error signals for the outputs of the feed-forward network. This is best illustrated in the following schematic representation of data flow.

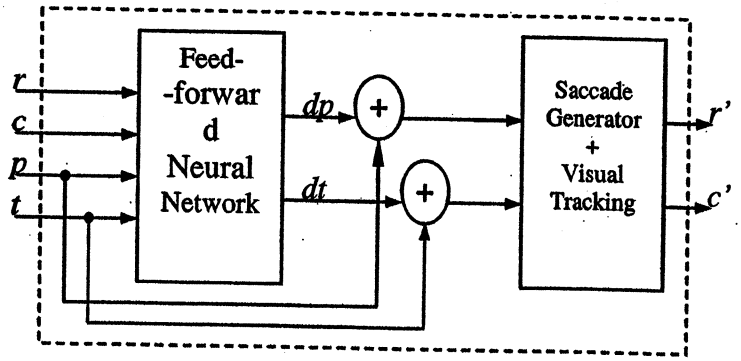


Figure 3: Schematic representation of data flow. Dashed box indicates the *composite performance system*.

In Figure 3, the *composite performance system* consists of the feed-forward neural network, the saccade generator and the visual tracking subsystem. A training example consists of input (r, c, p, t) and desired output (r'^*, c'^*), where (r'*, c'*) is the coordinate of the center of the post-saccade image. From the point of view of the composite performance system, the problem is a typical supervised learning problem. From the point of view of the feed-forward network, however, the problem can not be solved by supervised learning directly, because no desired output (dp^*, dt^*) of the feed-forward network is provided. This is called the *missing error signal* problem, because the error signal for the output of the feed-forward network is missing.

The missing error signal problem has been studied extensively in the more general context of distal supervised learning [6]. Although the general solution presented in [6] could have been used, there exist much simpler solutions for this specific problem. Kawato et al. [7] proposed to use an estimated error signal proportional to the error signal for (r', c'). This method has been successfully applied in [8] and [9].

The method used in [8] and [9] is very simple, but it only provides a very rough estimate of the real error signal for (dp, dt), which may cause the learning process to converge slowly. Our approach improves the accuracy of estimated error signals, and decouples the generation of training examples for the feed-forward network and the actual training of it, so that a standard error back-propagation algorithm could be used, instead of a customized one. This decoupling even facilitates the exploration of different network architectures and learning algorithms, since the same set of training examples could be used many times. The basic idea is to divide the learning process into two phases, and use a fixed approximation f' of the saccade control function f in the first phase. The approximation f' has to be reasonably good near the center of the field of view of the camera. This is possible because, for the wide-angle camera, the saccade control function is linear near the center of its field of view, but rapidly diverges towards the edges. A few experiments could yield a linear estimate f' of f which meets this requirement.

In the first phase, f' is used in place of f and remains unchanged. The sole purpose of this phase is to generate training examples using f'. One pass of this phase consists of the first five steps in the typical learning trial. After a saccade, as long as f' does not differ too much from f, we would expect the visual target location (r', c') to be near the center of the post-saccade image. Therefore, a second application of f' would provide the additional change (dp', dt') which is necessary to center the target in the post-saccade image. Obviously, the desired output of the saccade control function f is (dp'', dt''), where $dp'' = dp + dp'$ and $dt'' = dt + dt'$. It is easy to see that the error in (dp'', dt'') largely depends on the error of f' near the center of the field of view of the camera, which by definition of f' is very small. After each learning trial, an input and desired output pair is appended to a file. The first phase is repeated for as many times as needed to generate enough training examples.

The second phase then uses these training examples to train the feed-forward network for the saccade control function. This phase is a standard supervised learning problem. In this project, a standard error back-propagation learning algorithm is applied to a feed-forward network using MATLAB.

6. SIMULATION

The visual tracking subsystem and the saccade generator have been constructed. But due to some unexpected failure in the robot platform, there is not enough time to implement the training program to put them together. Instead, a simulation is carried out to demonstrate the effectiveness of the approach used in training the feed-forward network.

The simulation is based on the following two simplifying assumptions:

1. The axes of the pan and tilt motors are orthogonal.
2. The axes of the pan and tilt motors coincide at the center of the lens of the camera.

The first assumption implies that a change in pan motor coordinate only causes a change in column coordinate of a visual target, while a change in tilt motor coordinate only causes a change in row coordinate of a visual target. This property enables the decomposition of the saccade control function into two independent functions: a pan control function and a tilt control function. The second assumption further implies that the saccade control function is independent of the current pan and tilt motor coordinates. Therefore it suffices to study the saccade control function when $p = t = 0$. Furthermore, the row and column coordinates can be scaled to be within the range between -1 and 1. This is easily achieved by choosing the image center as the origin, and scale the row and column coordinates by half of the height and width of the image. In summary, the saccade control function can be represented by a pan control function $dp = f_p(c)$ and $dt = f_t(r)$, where $-1 \leqslant r, c \leqslant 1$ and $p = t = 0$. The simulation is performed on the pan control function f_p.

The function f_p is taken to be:

$$f_p(c) = \frac{2L}{\pi}\arcsin(c) \tag{4}$$

where L is the maximum change in proportional pan motor coordinate required to saccade to a point on the edges of an image. This form of f_p satisfies the requirement on its linearity: it is nearly linear when c approaches 0, and diverges when c approaches ± 1.

The linear estimate of f_p is simply taken to be the first order approximation of (4) near $c = 0$:

$$f_{est}(c) = \frac{2Lc}{\pi} \tag{5}$$

Suppose the initial column coordinate of a visual target in the pre-saccade image is c_1, the change in proportional pan motor coordinate for the saccade is given by:

$$dp_1 = f_{est}(c_1) = \frac{2Lc_1}{\pi} \tag{6}$$

However, due to imperfection in motor control, the actual proportional pan motor coordinate after the saccade will be:

$$p_1 = dp_1 + e_1 \tag{7}$$

where e_1 is a small random variable with some probability distribution.

After the saccade, the actual column coordination c_2 of the visual target in the post-saccade image satisfies the following condition:

$$f_p(c_2) + p_1 = f_p(c_1) \tag{8}$$

However, due to the error in visual tracking, the perceived column coordinate c_2' of the visual target in the post-saccade image will be:

$$c_2' = c_2 + e_2 \tag{9}$$

where e_2 is a small random variable with some probability distribution.

Our solution to the missing error signal problem defines the error signal for dp_1 to be:

$$dp_2 = f_{est}(c_2') = \frac{2Lc_2'}{\pi} \tag{10}$$

Finally, the desired change in proportional pan motor coordinate is given by:

$$dp = dp_1 + dp_2 \tag{11}$$

and the actual change in proportional pan motor coordinate is given by:

$$dp^* = f_p(c_1) \tag{12}$$

7. RESULTS AND DISCUSSION

In our simulation, L is set to 1, and e_1 and e_2 are set to Gaussian distributions with zero mean and standard deviation 0.05.

Figure 4 shows the errors of the training examples generated by one simulation.

These training examples are used to train a 3-layer feed-forward network with 1 input node, 3 hidden nodes, and 1 output node. All nodes use the *hyperbolic tangent* sigmoid nonlinearity as defined by:

$$\tanh(v) = \frac{2}{1+e^{-2v}} - 1 \tag{13}$$

Figure 5 shows the performance of a trained network.

These simulation results confirm the effectiveness of the approach used to generate training examples. Besides, these results illustrate how the performance of the

saccade control function is likely to degrade towards the edge of the field of view of cameras. In addition, repeated simulations have revealed that as the deviation of the random errors e_1 and e_2 increase to a significant level, such as 0.1, as depicted in Figure 6. This phenomenon highlights the importance in accurate visual tracking and motor control.

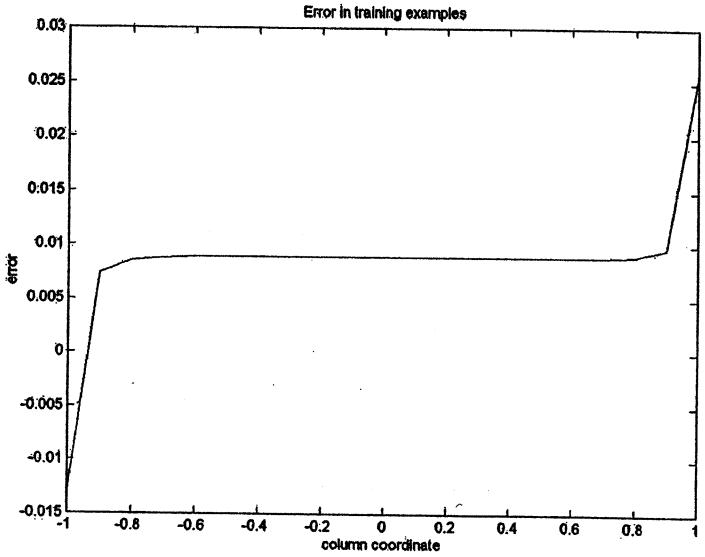


Figure 4: Error in training examples

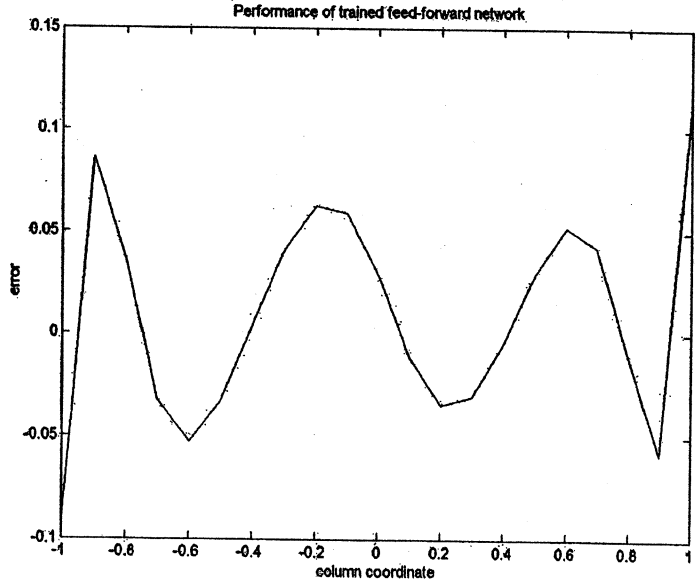


Figure 5: Performance of trained network

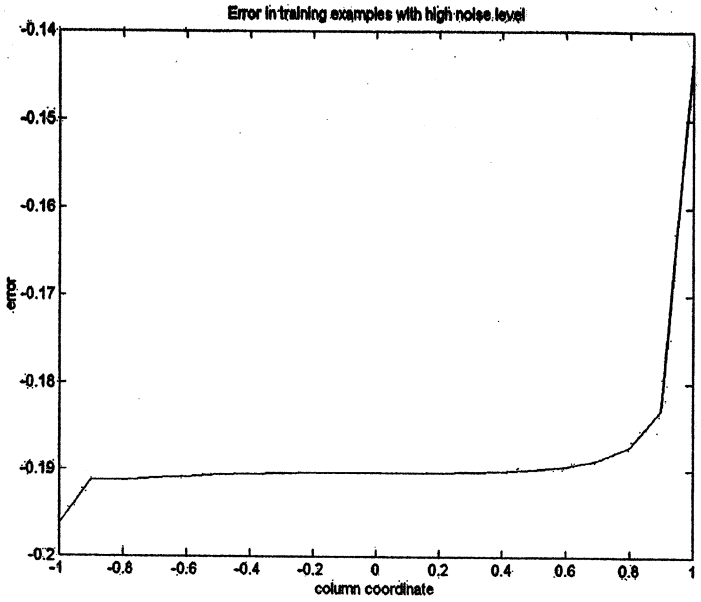


Figure 6: Error in training examples with high noise level

REFERENCES

1. Aloimonos, Y., Weiss, I., & Bandopadhay, A. (1987) Active Vision. *Int. J. Computer Vision*, Jan. 1987, vol. 1, pp. 333-356.
2. Murray, D.W., Bradshaw, K.J., McLauchlan, P.F., & Sharkey, P.M. (1995) Driving saccade to pursuit using image motion. *Int. J. Computer Vision*, vol. 16(3), pp. 205-228.
3. Massone, L. (1994) Sensorimotor learning. *The Handbook of Brain Theory and Neural Networks*, M.A. Arbib, Ed., Cambridge, M.A. MIT Press, to appear.
4. Swain, M.J., & Ballard, D.H. (1991) Colour Indexing. IJCV, pp. 11-32.
5. McKenna, S.J., Raja, Y., & Shaogang Gong. (1999) Tracking Colour Objects Using Adaptive Models. *Image and Vision Computing*, Vol. 17, pp. 225-231.
6. Jordan, M.I., & Rumelhart, D.E. (1992) Forward models: Supervised learning with a distal teacher. *Cognitive Science*, v. 16, pp. 307-354
7. Kawato, M. (1990) Feedback-error-learning neural network for supervised motor learning. *Advanced Neural Computers*, Elsevier, Amsterdam, pp. 365-372.
8. Bruske, J., Hansen, M., Riehn, L., & Sommer, G. (1996) Adaptive saccade control of a binocular head with dynamic cell structures. *Proc. Int. Conf. Artificial Neural Networks*, Bochum, Germany, Jul. 16-19, LNCS, Vol. 1112, pp. 215-220.
9. Marjanovic, M.J., Scassellati, B., & Williamson, M.M. (1996) Self-taught visually-guided pointing for a humanoid robot. *Proc. of the 4th Int. Conf. on Simulation of Adaptive Behavior (SAB'96)*, pp. 35-44.