A Model of Primate Visual-Motor Conditional Learning

Andrew H. Fagg
af0a@robotics.usc.edu

Michael A. Arbib
arbib@cs.usc.edu

Center for Neural Engineering and Computer Science Department
University of Southern California
Los Angeles, California 90089-0781

Abstract

Observations of behavior and neural activity in premotor cortex of monkeys learning to pair an arbitrary visual stimulus with one of a set of previously learned behaviors are modelled with a network comprising a large number of "motor selection columns." Reinforcement-learning is used to recognize new visual patterns and acquire the appropriate visual-motor conditions. The architecture employs a distributed representation in which a single pattern is coded by a small subset of columns. A column is initially able to respond to many different inputs; as it learns to trigger a motor program, its responses become more narrowly defined. Each column's output is a set of votes for the various motor programs. The votes for each program are collected by "selection units" which drive a winner-take-all circuit to determine whether or not a particular motor program is executed. The model is successful in reproducing the sequence of behavioral responses given by the subjects, as well as a number of phenomena that have been observed at the single-unit level.

1. Introduction

Mitz, Godshalk and Wise (Mitz, Godshalk, & Wise, 1991) examine learning-dependent activity in the premotor cortex of two rhesus monkeys required to move a lever in a particular direction in response to a specific visual stimulus. Figure 1 shows the protocol and expected response for one such trial. The monkey is initially given a ready signal, which is followed by a visual stimulus (instruction stimulus, IS). The monkey is then expected to wait for a flash in the visual stimulus (trigger stimulus, TS), and

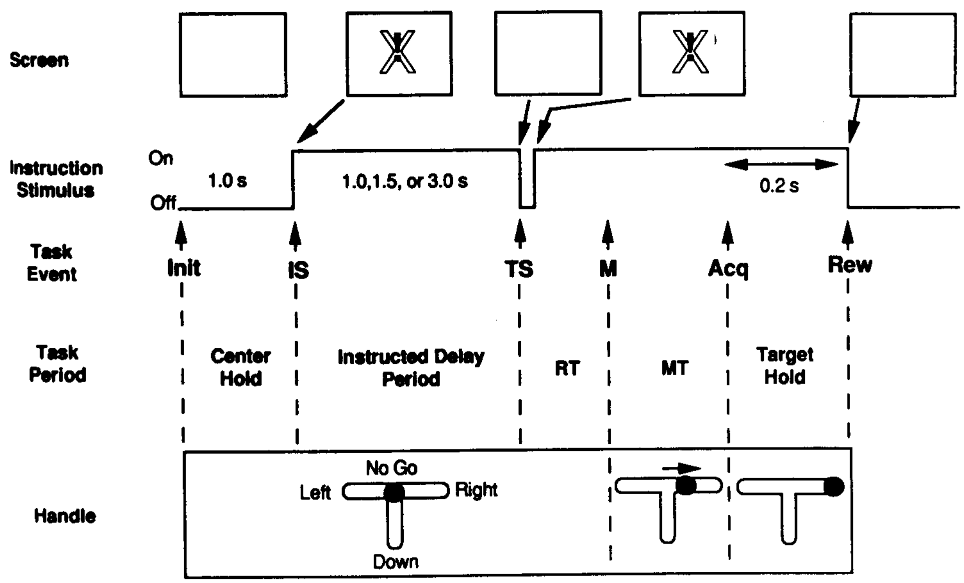


Figure 1: Top row : visual stimulus as seen on the video screen. Second row : temporal trace of the visual stimulus. Third and fourth rows : Primary events and periods of the experimental trial. Fifth row : expected motor response. (From Mitz et al., Figure 1; reprinted by permission of the Journal of Neuroscience.)

then produce the appropriate motor response. The four possible motor responses are: move the handle left, right, down, or no movement. When a correct response is produced, the subject is rewarded with a squirt of juice and a stimulus is picked randomly for the next trial. On the other hand, when an incorrect response is produced, no reward is given and the same stimulus is kept for the next trial.

During the initial training phase, the two subjects were trained to perform the task with a fixed set of visual stimuli. This phase taught the protocol to the subjects, including the four appropriate motor responses. Through the second phase of learning, which we model here, the subjects were presented with novel stimuli and were expected to produce one of the four previously-learned motor responses. It was during this phase that single-unit recordings were taken from neurons in the primary- and pre-motor cortices.

Figure 2 demonstrates the results of a typical set of second-phase experiments. The left-hand column shows the correct response, and each row of the right-hand column shows the monkey's response over time. Two features of this figure are particularly interesting. First, there are a number of cases in which the monkey exhibits an incorrect response, and even though it does not receive the positive feedback, it will continue to output the same response for several additional trials. In most of these cases, the no-go response is given, which appears to be the "default" response. The second interesting feature, demonstrated in almost half of these response traces, is that once the monkey exhibits the correct response, it may give one or more improper responses before producing the correct response consistently.

This behavior may be captured at a high level by considering a separate *decision box* for each

Correct response	Trial number 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Down (D)	L	+	+	+	+	+	+								
	R	L	L	+	R	+	+	+	+	+	+	+	+	+	
	R	+	R	L	+	+	+	+	+	+	+	+	+	+	+
	+	+	+	+	+	+	+	+							
	N	R	+	+	R	+	+	+							
	N	L	N	R	+	+	+	+	+	+	+	+	+	+	+
Right (R)	N	L	N	D	L	+	+	+							
	N	+	+	+	+										
	N	N	N	L	+	N	N	N	+	+	+				
	N	N	N	N	N	+	+	+	+						
	N	N	L	+	+	N	+	+	+						
Left (L)	R	D	+	N	+	+	+	+	+	+	+	+	+	+	+
	+	+	+	+	+	+	+	+	+	+	+	+	+	+	
	N	D	+	+	+	+	+	+	+	+	+	+	+	+	+
	N	D	+	+	+	+	+	+	+	+	+	+	+	+	+
	N	N	D	R	+	+	+	+	+	+	+	+			
	N	R	D	+	R	D	+	R	+	R	+	+	+		
No-go (N)	R	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	L	L	R	D	L	+	L	+	+	+	+	+	+	+	+
	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	D	R	D	+	D	L	R	+	L	D	R	D	+	+	+

Figure 2: Samples of responses to novel stimuli given example specific expected motor responses. Each row represents only those trials from an experiment that correspond to a specific desired motor response. Correct answers are indicated with a '+'. (From Mitz et al., Table 1; reprinted by permission of the Journal of Neuroscience.)

stimulus[1]. A box maintains a measure of confidence that each motor output is correct, given its particular input stimulus. When the system is presented with a stimulus, the appropriate box is chosen, and a motor output is selected based upon the confidence vector. When the monkey exhibits an incorrect response, positive reinforcement is not given. Therefore, the likelihood of the last response should be reduced slightly, while the probability of picking one of the other motor responses increases. When a correct response is given, the confidence value for the exhibited response is rewarded by a slight increase. Our challenge is to construct a neural implementation that is both distributed in nature and is capable of identifying novel stimuli as they are presented. The following data gives some hint as to how the implementation might look.

Mitz et al. recorded primarily from cells in the premotor cortex. A variety of cell types were identified. *Anticipatory* cells tend to fire between the ready signal and the IS. *Signal* cells respond to the presentation of a relevant stimulus, whereas *set-related* cells fire after the IS, in preparation for a particular motor response. *Movement-related* cells respond to the presentation of the TS and in some cases stay on for the duration of the movement. Most cells exhibit multiple response properties (e.g. combined set- and movement-related responses). Signal- , set-, and movement-related cells typically fired in correlation with a particular motor response. Thus, for any particular visual stimulus, only a

1 A more formal treatment of these computing elements (stochasitic learning automata) may be found in Bush (1958) and Williams (1988).

small subset of cells fired significantly during the execution of the corresponding motor program. As learning progressed, some cells were seen to increase in their response activity towards a stimulus, while others decreased in their response.

Figure 3 shows normalized activity and performance curves for one experiment plotted against the trial number. The normalized activity is computed for a particular stimulus by looking at the activity of the ensemble of units that show an increase in activity over the course of learning. The performance curve is computed as a sliding window over a set range of trials. It is important to note that the performance curve precedes the activity curve in its sudden increase.

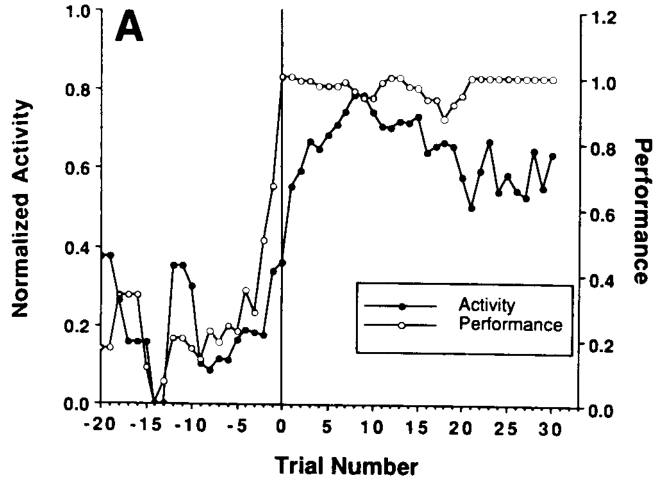


Figure 3: Normalized activity and performance curve plotted as a function of trial for the presentation of a novel stimulus. The rise in overall performance precedes that of cellular activity by about 3 trials. (From Mitz, Figure 3; reprinted by permission of the Journal of Neuroscience.)

Mitz et al. identified a number of key features of learning-dependent activity in these experiments :

a. The increase in cell activity (for those cells that increased their activity over the learning period) was closely correlated with, but was preceded by, the improvement in performance. Similar relations were seen in signal-, set-, and movement-related units.

b. Activity of a particular unit for correct responses was, in most cases, higher than that during incorrect responses in the same movement direction.

c. Activity for correct responses during times of good performance exceeded that at times of poor performance.

d. When multiple sets of novel stimuli were presented to the monkey, similar learning-dependent responses of the signal-, set-, and movement-related cells were observed for stimuli that yielded the same motor response.

e. The activity pattern resulting from a familiar stimulus closely correlated with the activity due to novel stimuli (after learning), although this correlation was not a perfect one. This and (d) demonstrate that a similar set of premotor neurons are involved in responding to all stimuli mapping to the same motor output. From this, we can conclude that the pattern discrimination is probably not happening within the premotor cortex. If this were the case, one would expect separate groups of cells to respond to different stimuli, even if these stimuli mapped to the same motor output.

This set of experimental results presents a set of modelling challenges. We here list both those that we meet in the present model, and those that pose challenges for future research.

1. Our neural model is capable of learning the stimulus/motor response mapping, producing qualitatively similar response traces to those of Fig. 2:

a. The appropriate number of trials are required to learn the mapping.

b. Incorrect responses are sometimes given several times in a row.

c. Correct responses are sometimes followed by a block of incorrect responses.

The model can generate the variety of response traces, with the network starting conditions determining the actual behavior.

2. The model produces realistic normalized activity/performance curves (Fig. 3) :

a. The performance curve leads the activity curve by a number of learning trials.

Challenges that remain for future work include:

3. A complete model will also reproduce the temporal activity of various neurons in the premotor cortex: anticipatory units, signal-related units, set-related units, and movement-related units.

2. Model

Much of neural network research has concentrated upon supervised learning techniques, such as the generalized delta rule (backpropagation, (Rumelhart, Hinton, & Williams, 1986)). In our modelling efforts, we have chosen to explore other algorithms within an architecture that can be related (at least at a high level) to the biological architecture, while perhaps also offering greater computational capability.

Backpropagation suffers from the problem of global representation — in general, every unit in the network, and thus every weight, participates in a single input-output mapping. As a result, the gradient in weight space due to a single pattern will contain a component for almost every weight, and therefore learning can become rather slow. A related problem is that, in order to maintain an older memory for at least some amount of time, the learning of a new memory cannot alter the older memory to all but a very small degree. This is difficult to accomplish if all units are participating in every computation and all weights are altered as a result of learning.

With these problems in mind, we have sought *distributed representations* in which a single pattern (or task) is coded by a small subset of the units in the network. Although different subsets of units are allowed to overlap to a certain degree, interference between two patterns is minimized by the non-overlapping components. Inspired by the cell activities observed by Mitz et al., we see a unit that has not learned to participate in a motor program as being able to respond to a wide range of different inputs. As learning progresses within this unit, its response increases significantly for some stimuli, while it decreases for the remainder.

The Architecture of the Model

The primary computational unit in the proposed model is the *motor selection column*, each consisting of two neurons: the *feature detector unit* and the *voting unit* (Figure 4). The overall network is composed of a large number of these columns, each performing a small portion of the stimulus-to-motor program mapping.

The feature detector recognizes small portions (micro-features) of the input stimulus. Due to the distributed construction of the circuit, a particular signal unit is not restricted to recognize patterns from a single stimulus, but may be excited by multiple patterns, even if these patterns code for different responses. A particular signal unit is physically connected to only a small subset of the input units. This enforces the constraint that only a small subset of the columns will participate in the recognition of a particular pattern. As will be discussed later, this reduces the interference between patterns during learning.

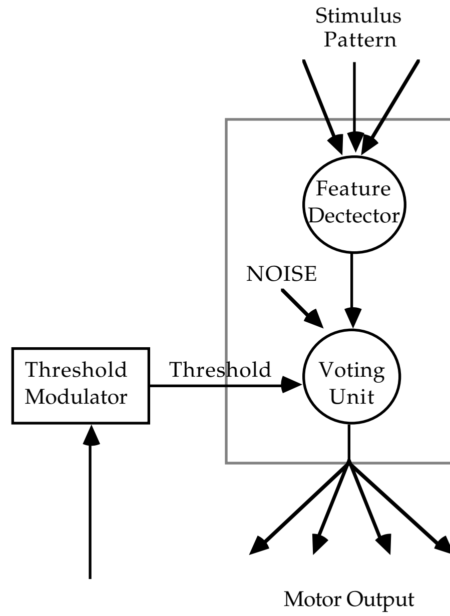


Figure 4 : The motor selection column model. The feature detector detects specific events from the sensory input. The voting unit produces a vote for an appropriate set of motor programs. This unit, along with the noise and the threshold modulator, implements a search mechanism.

The voting unit receives input from its corresponding feature detector as well as from a noise process and the threshold modulator. Based upon the resulting activity, the voting unit instantiates its *votes* for one or more motor programs. The strength of this vote depends upon the firing rate of this neuron and the strength of the connection between the voting unit and the motor program selector units. As shown in Figure 5, the votes from each column are collected by the *motor program selection units*, labeled "Left", "Right", "Down", and "No-Go". The final activity of these units determines whether or not a particular motor program is activated, and thus executed. The winner-take-all circuit (Didday 1976) ensures that at most one motor program will be activated at any one time. This is accomplished through the inhibitory neuron (S). When more than one motor program selection unit becomes active, this unit sends an inhibitory signal to the array of motor program selection units. The result is that all of the units will begin turning off, until only one is left (the unit receiving the largest total of votes from the motor columns; Amari and Arbib 1977). At this point, the one active unit will cue the execution of its motor program.

The reception of the trigger stimulus (TS) causes the execution of the selected motor program. Although only a single motor program selection unit will typically be active when the TS is received, two other cases are possible : none active, and more than one active. In both cases, the No-Go response is executed, irrespective of the state of the No-Go motor program selection unit. Thus, the No-Go response may be issued for one of two reasons : explicit selection of the response, or when the system is unsure as to an appropriate response by the time the TS is received.

The global threshold modulator and the local noise processes play an important role in the search for the appropriate motor program to activate. When a new visual stimulus is presented to the system, the feature detector units will often not respond significantly enough to bring the voting units above threshold. As a result, no voting information is passed on to the motor program selection units. The threshold modulator responds to this situation by slowly lowering the threshold of all of the voting units. Given time (before the TS), at least a few voting units are activated to contribute some votes to the motor program units. In this case, a response is forced, even though the system is very unsure as to what that response should be.

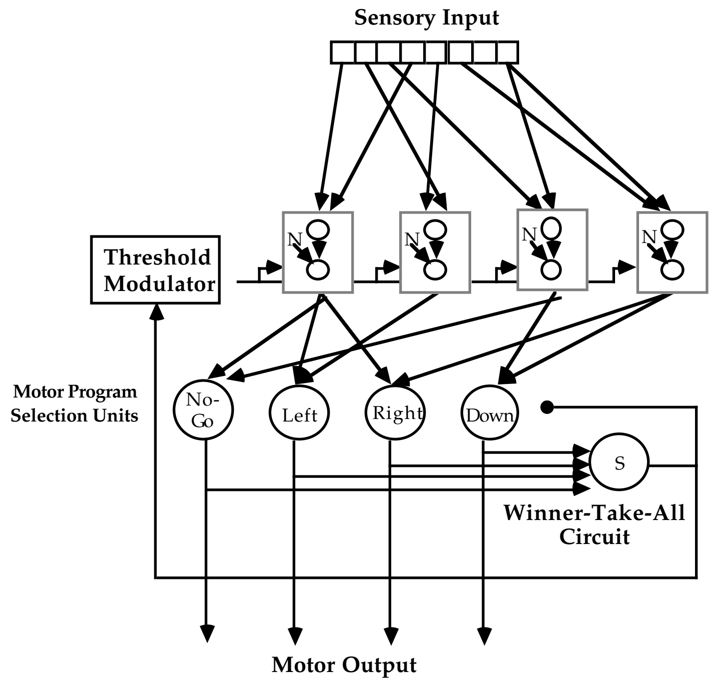


Figure 5 : The motor program selection model. A set of motor selection columns votes for the motor responses. The votes are collected by units representing the activity of the schemas for each of the legal motor responses. The winner-take-all circuit ensures that only one motor program is selected.

Noise processes have been used as an active element of several neural models. Noise is used in Boltzmann machines as a device for escaping local minima and as a way of breaking symmetry between two possible solution paths (Hinton & Sejnowski, 1986). Although the problem of local minima is not a concern in this work, the problem of choosing between two equally desirable solutions is a considerable one. By injecting a small amount of noise into the network, we randomly bias solutions so that a choice is forced within the winner-take-all (WTA) circuit. There are some cases in which two motor program selection units receive almost the same amount of activity. Due to the implementation of the winner-take-all circuit, this situation may send the system into oscillations, where it is not able to make a decision. The added noise coming into the voting units helps to bias one of the motor programs, to the point where a decision can be made quickly. Moreover, rather than always selecting the motor program that has the highest incoming feature support, the system is enabled by the noise to choose other possibilities. This keeps the system from prematurely committing to an incorrect solution, maintaining diversity during the search process (Barto, Sutton, & Anderson, 1983). Thus, the amount of time dedicated to the search process can be significantly decreased.

Network Initialization

The *layers* init-module creates the network by randomizing the input-to-feature and voting-to-

```
  INIT_MODULE(layers)
{
  if(!patterns_loaded_flag)
    {
      printf("Must load patterns before simulation initialization and execution.\n");
      exit(0);
    }
                         /* Feature detector weights. */
                         /* Random. */
  randomize(W_in_feature);                                                   (1.1)
                         /* Mask */
  select_random(W_in_feature_mask, W_in_feature_probability.elem());         (1.2)

                         /* Adjust weight values. */
  W_in_feature = (W_in_feature/2.0 + 0.5 + input_weight_bias)                (1.3)
    ^ W_in_feature_mask;
                         /* Normalize weights. */
  normal_col(W_in_feature);                                                  (1.4)

                         /* Voting unit weights. */
  randomize(W_vote_motor);                                                   (1.5)
  select_random(W_vote_motor_mask, W_vote_motor_probability.elem());         (1.6)
  W_vote_motor = (W_vote_motor/2.0 + 0.5 + voting_weight_bias)               (1.7)
    ^ W_vote_motor_mask;
                         /* Normalize as appropriate. */
  if(normalize_input_mode.elem() == 1.0)                                     (1.8)
    normal_col(W_vote_motor);
  else
    normal_row(W_vote_motor);

  counter = 0;                  /* Trial 0 */                                (1.9)
  pick_new_pattern(TRUE);  /* Pick first pattern. */                         (1.10)

  randomize(noise);             /* Intialize noise vectors. */               (1.11)
  noise = noise_gain * noise;

  randomize(motor_noise);
  motor_noise = motor_noise_gain * motor_noise;
}
```

motor projections and prepares the model to begin execution (see Apendix A for the network data structures).

First, the module configures the input-to-feature weight matrix (**W_in_feature**). (1.1) selects random values for each of the individual weights (uniform distribution in the interval [0,1]). (1.2) computes which physical connections actually exist. The call to **select_random()** initilizes **W_in_feature_mask** with a set of 0's and 1's (0 = no connection; 1 = connection). The probability that each element is set to 1 is determined by **W_in_feature_probability**. This weight mask is applied in (1.3) to **W_in_feature**, after a linear transformation is applied to the weights. After this operation, **W_in_feature** will consist of elements that are either 0 (when no connection exists), or selected from the distribution [**0.5+input_weight_bias, 1.0+input_weight_bias**]. The linear transform of the weight elements guarantees that those that exist take on significant initial values. The random trimming out of connections from the weight matrix is important for giving us a wide diversity of feature detectors to begin with. This will play an important role both in the initial behavior of the network, as well as in limiting the interference during learning.

The final operation that is applied to **W_in_feature** weight matrix is the normalization (1.4). The call to **normal_col()** L1-normalizes the columns of **W_in_Feature**. This ensures that the total output weight from any single input unit is 1 (presynaptic normalization). As these weights change during learning, this condition will continue to hold, thus implementing a form of competition between the connections leading from the input unit.

The vote-to-motor weights (**W_vote_motor**) are initialized in a similar manner (1.5, 1.6, and 1.7). However, normalization may be done one of two ways, depending upon the flag **normalize_input_mode**. Presynaptic normalization (**normalize_input_mode** = 1) is as above, maintaining the condition that the weights leading from the voting unit sum to 1. For postsynaptic normalization (**normalize_input_mode** = 0), the sum of the weights leading to the motor selection units would sum to 1. For the simulations results reported in this chapter, **normalize_input_mode** = 1 (presynaptic normalization).

The trial counter, **counter**, is initialized in (1.9). The call to **pick_new_pattern()** (1.10) causes the network state to be initialized in preparation for the first trial. The **TRUE** parameter forces the system to randomly select one of the available patterns to be presented to the network. Finally, the noise vectors, **noise** and **motor_noise** (which are injected into the voting and motor selection units, respectively), are initialized (1.11).

Network Dynamics

The following RUN_MODULEs implement a single time-step of the simulation. Each of the neurons are implemented as leaky-integrators (Arbib, 1989).

```
RUN_MODULE(FEATURE)
{
  diff(feature_mem, u_feature, -feature_mem - threshold_f +          (2.1)
       mat_mult_col_vec(W_in_feature, inputs));
  feature = NSLsat(feature_mem, 0, 1, 0, 1);                         (2.2)
}
```

The FEATURE module computes the membrane potential (**feature_mem**) and the firing rate (**feature**) of the feature detector units. In (2.1), **feature_mem** depends upon the threshold (**threshold_f**), and the activity received from the **inputs** layer. The call to **mat_mult_col_vec()** multiplies the weight matrix **W_in_feature** by the column vector **inputs**, and returns a vector containing the net input to the feature detector units. The firing rate of the feature detector unit is limited to the range [0..1] (2.2).

The **NOISE** module computes the next noise signals that are to be injected into the voting units and the motor selection units. What is implemented here are noise processes that change value on occasional time-steps. This slow change of injected noise is important for the behavior of the network. As will be seen in the next two modules, the voting units and the motor selection units are also implemented as leaky-integrator neurons, which implement a low-pass filter on the inputs coming into them. If the injected noise changed drastically on every time-step, this high-frequency noise would for

```
RUN_MODULE(NOISE)
{
                          /* Roll die to see if change is appropriate. */
  if(random_value2() < noise_change_probability.elem())                        (3.1)
    {
                          /* Change noise values. */
      randomize(noise);                                                        (3.2)
      noise = noise_gain * noise;
    }
                          /* Motor Program Selection noise. */
  if(random_value2() < motor_noise_change_probability.elem())                  (3.3)
    {
      randomize(motor_noise);                                                  (3.4)
      motor_noise = motor_noise_gain * motor_noise;
    }
}
```

the most part be filtered out. By forcing the noise process to change more slowly, the neurons are given an opportunity to respond in a significant manner.

The frequency at which a new noise vector (**noise**) is selected is determined by the parameter **noise_change_probability** (3.1). If it is time to update the noise vector, a completely new vector is generated (3.2), and then scaled by the **noise_gain** parameter. The same process is used to compute the **motor_noise** vector (3.3, 3.4).

```
RUN_MODULE(THRESHOLD_V)
{
  if(S.elem() <= 0.0)                                                          (4.1)
    diff(threshold_v, u_threshold_v, -threshold_v);                            (4.2)
}
```

The **THRESHOLD_V** module implements the dynamics of the threshold modulator. The threshold level, **threshold_v**, is initially set at the beginning of the trial to the parameter **init_threshold_v**. This value then decays exponentially (4.2). However, this decay only happens as long as no motor selection units have begun to fire (as measured by the activity level of the inhibitory unit; 4.1). In order for this event to occur, several voting units must have begun to fire (see next module), giving the system the ability to make some sort of decision.

```
RUN_MODULE(VOTING)
{
  diff(voting_mem, u_voting, -voting_mem - threshold_v + feature + noise);     (5.1)
  voting = NSLramp(voting_mem);                                                (5.2)
  voting_participation = NSLstep(voting);                                      (5.3)
}
```

The **VOTING** module computes the state of the voting units. The membrane potential of these units (**voting_mem**) is determined by the firing rate of the corresponding feature detector units, a noise signal, and the signal from the threshold modulator (5.1). When a visual stimulus is initially presented, the inhibitory signal from the threshold modulator is at a high level. If the stimulus is relatively unfamiliar, the input from the feature detector unit will typically not be above this threshold. As a result, no decision will be immediately made. However, the threshold modulator will begin to slowly drop this threshold, ultimately forcing several voting units to fire, causing a decision to be made at the motor selection unit level.

The noise process plays an important role in the search for the correct input/output mapping. At this level, the noise causes different columns to participate in the mapping from trial to trial. Over time, this allows the system to consider many combinations of sets of columns until an appropriate set can be found.

The firing rate of the voting units requires a membrane potential above some threshold (5.2). The vector **voting_participation** is used to display to the user which columns are participating within any particular computation.

```
RUN_MODULE(MOTOR)
{
  motor_inputs = mat_mult_col_vec(W_vote_motor, voting)/NUM_COLUMN;          (6.1)
  dmotor_mem = -motor_mem - threshold_m +
    voting_factor * motor_inputs -                                           (6.2)
      S + motor + motor_noise;
  diff(motor_mem, u_motor, dmotor_mem);
  motor = NSLstep(motor_mem);                                                (6.3)
}
```

The motor selection unit dynamics are determined within the **MOTOR** module. The membrane potential of these units is a function of the votes from the feature detector units (**motor_inputs**; 6.1), the inhibitory signal from the WTA (Winner-Take-All) inhibitory unit (**S**), and an injected noise signal (**motor_noise**) (6.2). The inhibitory unit ensures that when the system has reached an equilibrium point, at most one motor program has become selected. This style of distributed Winner-Take-All computation is due to (Amari & Arbib, 1977).

The noise signal is important at this point for providing a diversity in the search for the correct mapping. In addition, it helps to prevent the system from becoming stuck onto a saddle point, where it cannot decide between one of two equally-active motor selection units.

The motor selection cells fire maximally whenever the membrane potential exceeds the cell's threshold (6.3).

```
/*
 * RUN_MODULE(S_COMPUTE)
 *
 *  Inhibitory unit dynamics (for Winner Take All implementation).
 *
 */

RUN_MODULE(S_COMPUTE)
{
  diff(S_mem, u_S, - S_mem - threshold_s + motor.sum());                     (7.1)
  S = NSLramp(S_mem);                                                        (7.2)
}
```

The **S_COMPUTE** module implements the dynamics of the WTA inhibitory unit. The membrane potential of this unit is driven to a level that is essentially proportional to the number of motor selection units that have become active (these units either have an activity level of 0 or 1) (7.1). The firing rate of this unit also reflects the number of currently active motor selection units (7.2).

Trial Completion and Learning Dynamics

Learning in this model is reinforcement-based, and is implemented by modifying two sets of synapses: the sensory input to feature detector mapping and the voting unit to motor program selection unit mapping, i.e., the weight matrices **W_in_feature** and **W_vote_motor**. Only those columns that participate in the current computation adjust their weights. In the experimental setup used for the monkeys, positive reinforcement is given when the monkey exhibits a correct response, and nothing is given when an incorrect response is output. Similarly, in the model, a scalar quantity called **reinforcement** is set by the teacher to +1 if the selected motor program is correct, and to -1 otherwise (we assume that the monkey internally translates the lack of a positive reinforcement signal into a negative reinforcement signal).

When positive reinforcement is given, the weights leading into the feature detector units are adjusted such that the feature detector better recognizes the current sensory input. In the case of negative reinforcement, the weights are adjusted in the opposite direction, such that the current input is recognized by the feature detector unit to an even lesser degree. The voting unit to motor selection mapping is adjusted similarly. Positive reinforcement increases the weight of the synapse to the correct motor program. When negative reinforcement is given, the synapse strength is weakened, allowing the other motor program selection units more of an opportunity to win the competition the next time the same stimulus is presented. It is important to note that this reinforcement depends on whether or not the overall system response was correct, not on the output of any individual motor selection column.

A special case occurs when the system is unable to make a decision within the allotted time (causing the "No-Go" response to be selected). Two possible situations have occurred: no motor program selection units are active, or more than one are active. In the first case, the reinforcement term is set to +1 by the system itself, regardless of the teacher feedback. This causes the currently active columns to be rewarded, ensuring that the next time the pattern is presented, these columns will yield a greater response. Thus, they will have a greater chance of activating one of the motor program selection units the next time the same situation arises. Without this additional term, negative reinforcement from the teacher is disastrous. The negative reinforcement further decreases the connection strengths of the already poorly responding columns, further decreasing their response. The result is a self-reinforcing situation that can never discover the correct response.

In the second situation, where more than one motor program selection unit becomes active at one time, the reinforcement term is set by the system to -1. This will decrease the response of all columns involved, adjusting the input to the two (or more) motor program selection units until one is able to achieve threshold significantly before the other(s). This decrease in response levels, combined with the noise injected into the model will ultimately break the symmetry between the competing motor selection units.

The **check_completion** module detects these four termination conditions, computes the appropriate reinforcement, updates the weight matrices and prepares the network for the next trial.

```
RUN_MODULE(check_completion)
{
  int i;
  int count_greater;
  int count_below;
  int flag;

  timer_flag = 0;
                          /* Check the status of each motor program */
                          /*  selection unit. */
  for(i = 0, count_below = count_greater = 0, flag = 1;                              (8.1)
      flag && i < NUM_CHOICES;
      ++i)
    {
                          /* Above threshold. */
      if(motor.elem(i) > 0.0)                                                        (8.2)
            {
                   ++count_greater;
                   winner = i;
            }
                          /* No change in activity. */
                          /*  - used only in standard WTA. */
      if(fabs(dmotor_mem.elem(i)) < stable_detect_threshold.elem())                 (8.3)
            ++count_below;
    }

                          /* Exactly one unit active and */
                          /*  in first pole mode or all units are  */
                          /*  stable. */
  if(count_greater == 1 &&                                                           (8.4)
     (count_below == NUM_CHOICES || first_pole_mode.elem() != 0.0))
    {
      timer_flag = 1;            /* Made decision in time. */
      if(collect_mode)           /* Update weights and set up next trial. */
            log_and_pick_new(winner);                                                (8.5)
    }
                          /* Timeout: NO-GO case. */
  if(!timer_flag && ++time_counter >= (int) max_time_counter.elem())                 (8.6)
    {
      if(count_greater == 0)
            timer_flag = -1;     /* Signal no activity at MPSU level. */             (8.7)
      else
            timer_flag = 1;             /* Too many MPSUs active. */                 (8.8)
      if(collect_mode)           /* Update weights and set up next trial. */
            log_and_pick_new(NO_GO_CASE);                                            (8.9)
    }
}
```

The state of the motor program selection units is checked to determine whether or not the system is itself ready to generate an output (8.1). The counter **count_greater** is incremented for every unit that is currently firing (8.2); **count_below** computes the number of units that have reached an equilibrium point (8.3). The model is ready to output a motor response in one of two cases, depending upon the state of the **first_pole_mode** flag (8.4). When this flag is FALSE, this indicates that the standard WTA is being used as the competition mechanism. This mechanism requires that exactly one motor program be selected and that all motor program selection units have reached equilibrium. When the flag is TRUE, first-past-the-pole WTA is used, which relaxes the constraint that the motor program selection units be in an equilibrium state. In either case, the trial is terminated and the weights are updated in the

```
void log_and_pick_new(int winner)
{
  ++counter;                      /* Trial counter. */

  if((int) P_Output.elem(current_pat) == winner)
    {
                          /* Reward the system. */
      reward_cmd();                                                   (9.1)
      cout << "+ ";
      log_file << "+ ";
    }
  else
    {                             /* Punish the system. */
      punish_cmd();                                                   (9.2)
      cout << "- ";
      log_file << "- ";
    }
                          /* Pick the new input pattern. */
  pick_new_pattern(repeat_mode.elem() == 0.0 ||                       (9.3)
             (int) P_Output.elem(current_pat) == winner);
}
```

call to **log_and_pick_new** (8.6).

Termination of the current trial may also be forced if the go signal has arrived (8.6). As stated earlier, two cases are possible: no motor program selection units active (8.7), or more than one active (8.8). In either case, the trial is terminated and the weight matrices are updated (8.9).

The **log_and_pick_new()** routine generates the teacher's reinforcement signal, depending upon whether or not the output action was correct (9.1) or incorrect (9.2) and updates the weights. The routine then selects a new input pattern to present to the system (9.3) in preparation for the next trial. If **repeat_mode** is TRUE, then a new random pattern is selected only if the system generated the correct output action (otherwise, the current pattern is repeated). When **repeat_mode** is FALSE, then a random pattern is selected on every trial regardless of the motor act that was output.

The **reward_cmd()** and **punish_cmd()** routines simply translate the reward (10.1) and punishment (10.2) into reinforcement signals that are used to update the weight matrices. These routines may also be accessed by using the "user reward" and "user punish" commands in the NSL interpreter (see the nsl_main.c file for more information as to how this is implemented).

The **alter_weights()** routine is responsible for the internal modulation of the reinforcement signal and the update of the weight matrices. As discussed earlier, if the system was unable to make a decision in the alotted time and no motor program selection units were active, then the reinforcement signal is set to 1 (11.1). This will cause all of the currently active feature detector units to become a little more active the next time the same input is presented, improving the chances that a motor program selection unit will be activated. Note for the other degenerate case, where more than one motor program selection unit is active at the completion of the trial, **factor** has already been set to -1 (passed in to **alter_weights()**).

For this particular task, the information content of a negative reinforcement is much less than that

```
void reward_cmd()
{
  alter_weights(1.0);                                              (10.1)
  if(!collect_mode)
    cout << "Rewarded" nl;
}

void punish_cmd()
{
  alter_weights(-1.0);                                             (10.2)
  if(!collect_mode)
    cout << "Punished" nl;
}
```

of positive reinforcement. This is the case because positive reinforcement indicates the exact answer that is expected, whereas negative reinforcement only tells the system that the selected action was not the correct one. Because this is the case, the connection strength adjustment due to negative reinforcement should be smaller than in the positive reinforcement case. This is implemented here by discounting the negative reinforcement signal (11.2), and leaving the positive reinforcement signal intact (11.3).

voting_contribution (11.4) identifies those columns that are currently participating in the computation and the degree to which they are participating. Only those connections which carry signals in to or out of the active columns will change in strength. The update to the input-to-feature mapping is computed in (11.5). The call to **vec_mult_vec_trans()** computes the outer product of the two vectors, returning a matrix of elements which indicate coactivity of input unit and column pairs (Hebbian component). **W_in_feature_mask** filters out all elements of the resulting matrix except for those pairs between which a connection exists. **lrate_f** is the learning rate, and **f_factor** modulates the update based upon the incoming reinforcement signal. This update matrix is then combined into the weight matrix (11.6) (the call to **NSLramp()** ensures that all connection strengths are always positive), and the weights are normalized (11.7).

The call to **normal_col()** L1-normalizes the columns of the weight matrix. This continues to maintain the constraint that the total output weight from any single input unit is 1 (presynaptic normalization). In other words, each input unit has a fixed amount of support that it can distribute between the feature detector units. When positive reinforcement is received, more of this support is allocated to the currently active columns at the expense of those columns that are not active. Likewise, when negative reinforcement is received, the support for the active columns is reduced, to the benefit of the remaining columns (driving the search for a more appropriate group of columns).

A similar learning rule is applied to the voting-to-motor mapping. The change in weights is a function of the co-activity of voting columns and the motor program selection units, modulated by the learning rate (**lrate_v**) and the reinforcement signal (11.8). These delta values are then added into the weight matrix (11.9), and normalized. For this mapping, the type of normalization is selectable as

```
void alter_weights(float factor)
{
  float f_factor;
                         /* If no MPSUs are active, then modify */
                         /*  the reward signal so it is positive. */
                         /*  This is done so that some column will */
  if(timer_flag == -1)   /*  be more likely to be active the next */        (11.1)
    factor = 1;          /*  time the same input is given.       */

                         /* Apply correction factor if reinforcement */
                         /*  is negative.  Discussed in paper.      */
  if(factor < 0.0)
    f_factor = factor * negative_factor_f.elem();                           (11.2)
  else
    f_factor = factor;                                                      (11.3)
                         /* Compute the effective activity of the */
                         /*  voting units to be used in the hebbian */
                         /*  association.    */
  voting_contribution = voting_participation;                               (11.4)


                         /*  Input weights.  */

  dW_in_feature = f_factor * lrate_f *
    (vec_mult_vec_trans(voting_contribution, inputs) ^ W_in_feature_mask);  (11.5)

                         /* Keep weights within [0,1] range. */
  W_in_feature = NSLramp(W_in_feature + dW_in_feature);                     (11.6)

                         /* Normalize weights. */
  normal_col(W_in_feature);                                                 (11.7)

                         /*  Movement weights.  */

  dW_vote_motor = factor * lrate_v *
    (vec_mult_vec_trans(motor, voting_contribution) ^ W_vote_motor_mask);   (11.8)

                         /* Keep weights non-negative. */
  W_vote_motor = NSLramp(W_vote_motor + dW_vote_motor);                     (11.9)

                         /* Normalize weights, depending upon mode. */
  if(normalize_input_mode.elem() == 1.0)
    normal_col(W_vote_motor);                                               (11.10)
  else
    normal_row(W_vote_motor);                                               (11.11)

}
```

either presynaptic (11.10) or postsynaptic (11.11). For the results reported in this chapter, presynaptic normalization is used, implementing a competition between the different motor program selection units for support from the columns.

When a response is generated, learning is applied to each of the columns that are currently participating in the computation. The learning objective of an individual column is to recognize particular patterns (or subpatterns) and to identify which of the possible motor programs deserves its votes. Equations (11.5-11.7) attempts to create feature detectors that are specific to the incoming patterns. As these feature detectors begin to better recognize the correct patterns, the activity of the signal units will grow with respect to the pattern, thus giving the column a larger voting power. The feature detecting algorithm is related to the competitive learning of von der Malsburg (von der Malsburg, 1973) and Grossberg (Grossberg, 1976) (discussed further in (Rumelhart & Zipser, 1986)). Individual columns learn to become feature detectors for specific subpatterns of the visual stimulus. However, a column does not recognize a pattern to the exclusion of other patterns. Instead, several columns participate in the recognition at once. In addition, a column is responsible for directly generating an appropriate motor output. Therefore, the update of the feature detector weights not only depends upon recognition of the pattern (as in competitive learning), but also upon whether or not the network generates the correct motor output. In the case of a correct response, the feature detector weights become better tuned towards the incoming stimulus, as in the von der Malsburg formulation. For an incorrect response, the weights are adjusted in the opposite direction, such that recognition is lessened for the current input.

In this scheme, all of the columns that participate in the voting are punished or rewarded as a whole, depending upon the strength of their activity. Thus, a column that votes for an incorrect choice may still be rewarded as long as the entire set of votes chose the correct motor program. This method works, in general, because this *incorrect column* will always be active in conjunction with several other columns that do vote appropriately and are always able to overrule its vote. This scheme is similar to that used by (Barto, et al., 1983) in that one or more elements may correct for errors made by another element. In their case, however, the correction is made sequentially through time, rather than in parallel.

It should be noted that there is a tradeoff in this algorithm between the speed of learning and the sensitivity to noise. Because this protocol always gives the correct feedback and the possible motor outputs are finite and discrete, this tradeoff is not quite as evident. Imagine the case where learning is very fast and the reinforcement function occasionally makes a mistake (as can easily be imagined in real-world situations). If the system has discovered the correct response, but is then given no positive reinforcement for the correct response, extremely rapid learning would cause this response to lose favor completely. Likewise, if an incorrect behavioral response is positively rewarded, a high learning rate would cause the incorrect response to rise quickly above the alternatives.

Simulation Results

The following experiment utilized a variant of the winner-take-all (WTA) algorithm in the simulation, referred to as first-past-the-pole WTA. Rather than requiring that the network settle down into a stable state ($\frac{d\,Motor_{mem}}{dt} \approx 0$ for all motor program selection units), the first unit that achieves a membrane potential above the threshold is declared the winner. In the case that more than one unit activated at the same instant, the standard winner-take-all circuit is used to squelch the activity of all but one. Using this particular algorithm allows for a faster simulation, since more time is required if the units must settle down to equilibrium.

```
N : + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +

L : N + R + + + + + + + + + + + + + + + + + + + + + + + + + + + +

R : N + L D L N + + + + + + + + + + + + + + + + + + + + + + + + +

D : L L N N + + + + + + + + + + + + + + + + + + + + + + + + + + +
```

Figure 6: The behavioral responses of one experiment broken into sequences corresponding to a particular stimulus/motor output pair. +'s indicate correct responses, where letters indicate an incorrect response of a particular type.

During the testing/learning trials, a pattern is randomly presented to the system. The overall control system waits until a single selection is made (after the TS is presented), before moving on the the next trial. When the network produces an incorrect answer, the same pattern is presented on the next trial (as in the primate experiments). This protocol allows for much quicker learning, as opposed to a completely random sequence of stimulus/response pairs.

Primary Experiments

Figure 6 shows the behavioral traces resulting from a single experiment. In two of the three traces, the network produces a correct answer, and then attempts other choices (given the identical pattern). This happens due to the fact that the voting strengths are influenced by the noise process. Even though a correct answer has been given, there is still a probability that another answer will be output at a later time. Eventually, however, the learning biases the correct motor program to a level sufficiently above the noise. After this point, the correct motor program is always chosen.

For the same experiment, the pattern of activity for a particular motor response and the behavioral performance were compared to those of the monkey. The overall activity of the resulting voting unit response is measured by using the final voting unit activity pattern (i.e. after learning) as a reference. The overall activity is defined as the dot product between this reference and the voting unit activity pattern from every trial in the learning sequence. As in the results reported by Mitz et al., the normalized activity and performance curves for a single motor response are plotted together. The performance is computed by low-pass filtering the performance value (where 0 corresponds to incorrect response and 1 to correct).

Figure 7 shows the resulting set of curves for one such experiment. The solid curve corresponds to the activity measure and the dotted line is the behavioral performance of the subject. These values are plotted over the number of trials. In this, as in several other experiments, the performance begins its steady increase 3 to 4 trials before the activity measure becomes significant.

When a naive network is first tested, the presentation of a pattern causes some random set of columns to be active, as determined by the initial weight values from the input units to the feature

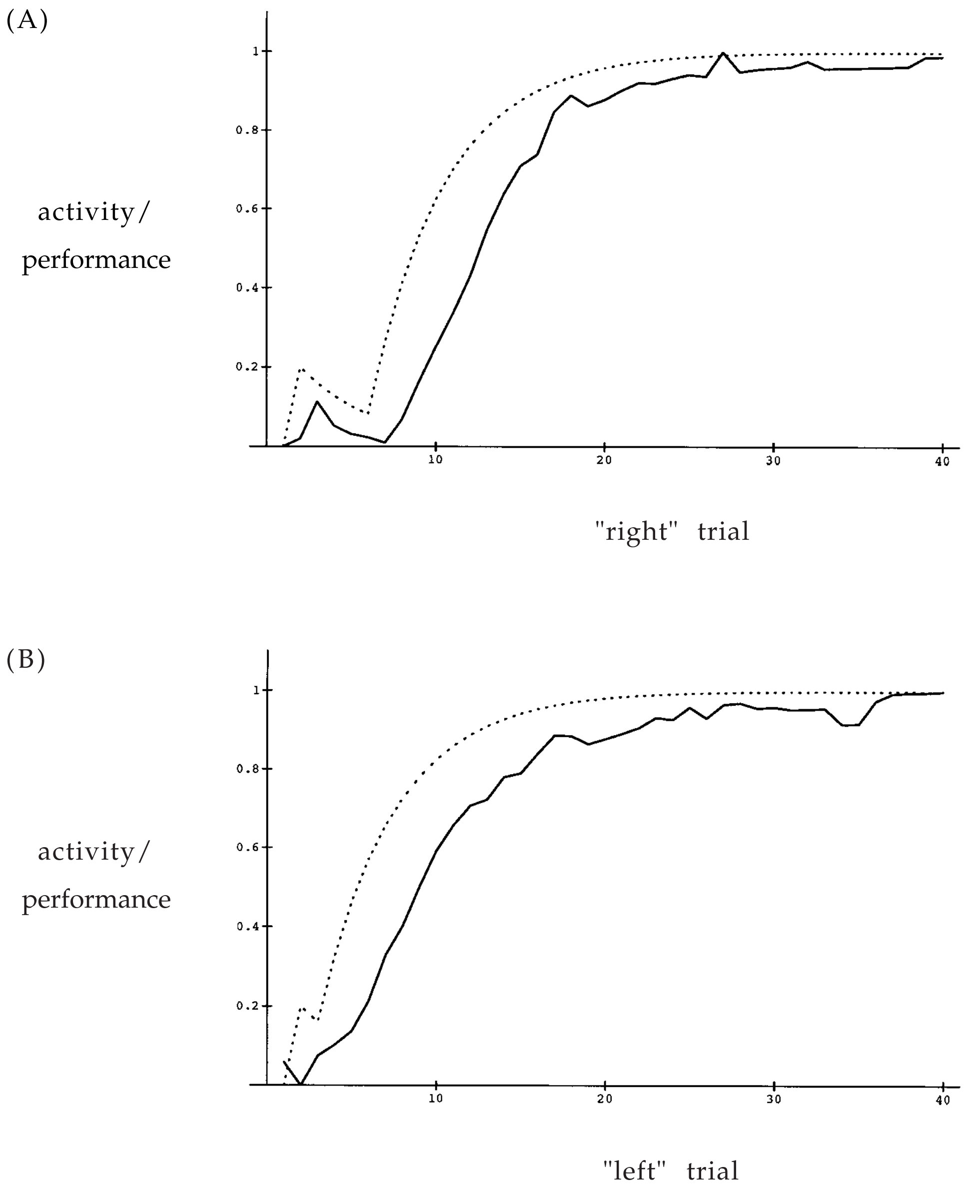


Figure 7: Overall activity (solid) curve and performance (dotted) curve plotted against trial for the (A) "Right" and (B) "Left" responses. As in the experimental curves, the performance curve begins to increase prior to the increase in the activity curve. Note that the trial axis represents only those trials for which the "Right" and "Left" responses are expected, respectively.

detector units. Based on the strength of the pattern match, the corresponding voting units may not immediately become active, but instead have to wait for the Threshold Modulator to lower the threshold to an appropriate level. Given time, this function forces the system to vote for some response, even though it is not very sure about what the correct response might be.

With respect to identifying the correct response, positive reinforcement gives the system more information than does negative reinforcement. For the case of positive reinforcement, we are telling the system what the correct response is (specific feedback), but negative reinforcement only tells the system that the correct response is one of three choices (nonspecific feedback).

Because the system is essentially guessing on these initial trials, the performance is very poor at first. Therefore, the system is primarily receiving negative reinforcement, keeping the overall response activity at a low level. An occasional correct response, in combination with the negative feedback for other choices, begins to bias the voting unit output towards the correct motor program selection unit. In turn, this effect begins to increase the probability of selecting the correct motor response.

Once the performance of a set of columns begins to increase, the positive feedback becomes significant enough to reward the correctly responding feature detector units on average, thus switching over from nonspecific to specific feedback information (in the weight update equations, the **reinforcement** term becomes +1 for the most cases). In Figure 7, as in other experiments, the overall activity does not begin to rise significantly until the performance passes the 0.5 mark. This also appears to be the case in most of the graphs provided by Mitz et al. Once the performance is correct on average, the activity of the feature detector units belonging to the "correct" set of columns increases. This increase comes from the fine tuning of the feature detector weights towards the incoming pattern. As a result, we see an overall increase in activity in response to the learned stimulus, and, most importantly, we see this increase **after** the increase in performance.

In addition to looking at the overall activity of the network, it is also possible to examine an individual column's response to input stimuli as learning progresses. Figure 8A shows the response activity of one voting unit from the same experiment. In this particular case, the unit initially responds equally well to two different stimuli. As learning progresses, however, the response to one stimulus grows to a significant level. Ultimately, this unit becomes allocated to the recognition of the stimulus pattern that maps to the Left response.

Figure 8B represents the same unit's orientation towards a particular motor program, as measured by the weight from the unit to the motor program selection unit. Initially, the unit supports the four motor program selection units almost equally, but within 12 trials, the weight corresponding to the Leftward motor unit begins increase above the others possibilities. After learning has completed, this weight completely dominates the others.

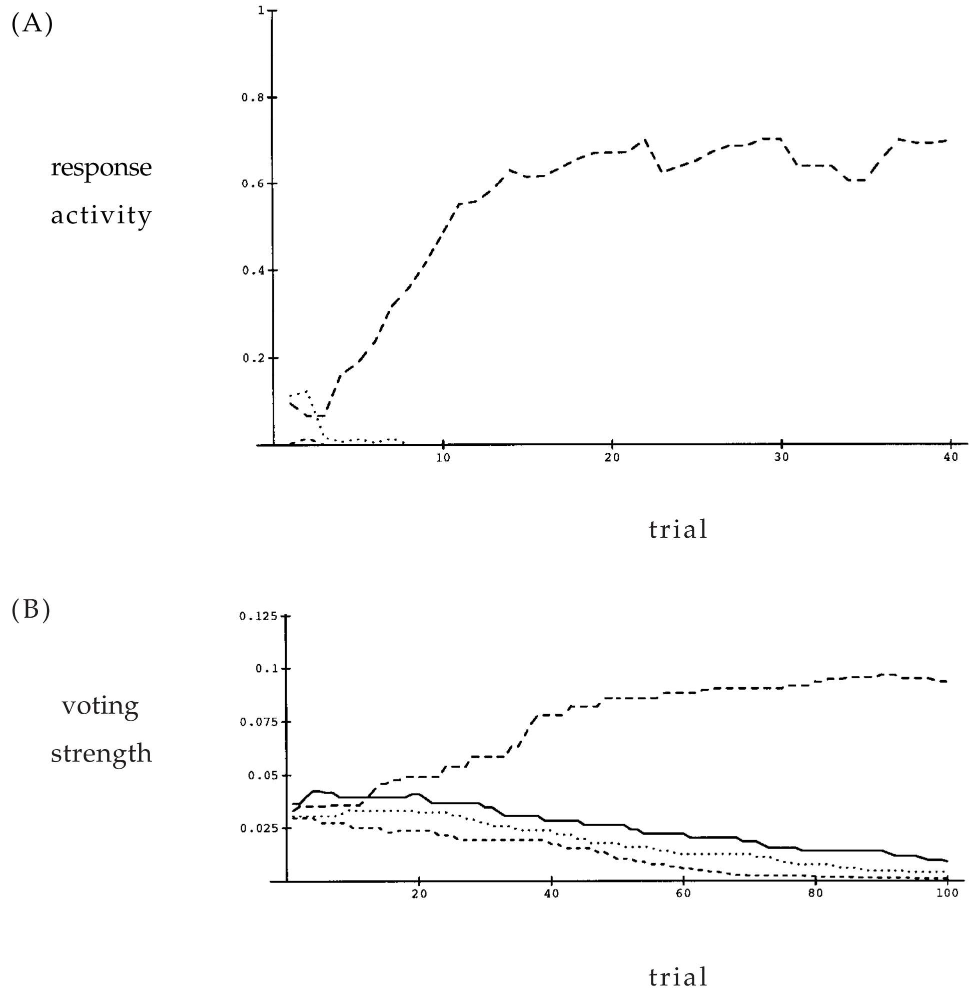


Figure 8: (A) Activity of a single voting unit plotted against trial number. The four curves represent responses to each of the four input stimuli (solid = no-go, long dash = left, dotted = right, and short dash = down). Note that trial number *N* corresponds to the Nth occurrence of each of the four stimuli, and does not necessarily correspond to the same moment in time. (B) Evolution of the voting strength of the same unit. The four curves (designated as above) represent the voting strength to each of the four motor program selection units. Note that in this graph, the trial axis represents all trials.

Changes in Protocol

In initially examining the protocol described by Mitz et al., we found it interesting that when the monkey responded incorrectly to a particular stimulus, the same stimulus was presented for the next trial. This repetition was continued until the monkey produced the correct response. The question that came immediately to mind was why a totally random presentation sequence was not used. We presented this question to our model through a simple modification of the protocol. The results shown in Figure 9 represent a typical behavioral trace under this new protocol. In this case, the system requires almost twice as many trials before it begins to perform the task perfectly. This is especially evident in the Rightward response.

This effect can best be explained by looking at the competition between the different stimuli. The degree of competition is determined by the amount of overlap between the sets of columns that are

```
N : D R D L + + R D + + + + + + + + + + + + + + + + + + + + + +

L : + D + + + + + + + + + + + + + + + + + + + + + + + + + + + +

R : L D D N N N N N N N N N N + + + + + + + + + + + + + + + + +

D : L + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
```

Figure 9: The behavioral responses of one experiment with a completely random sequence of stimulus presentations. Under these conditions, the task requires more trials of learning.

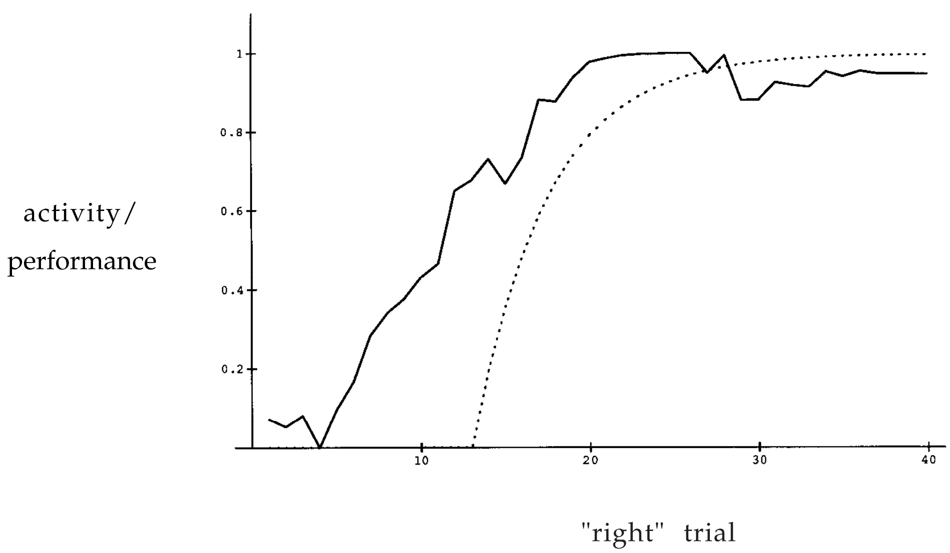
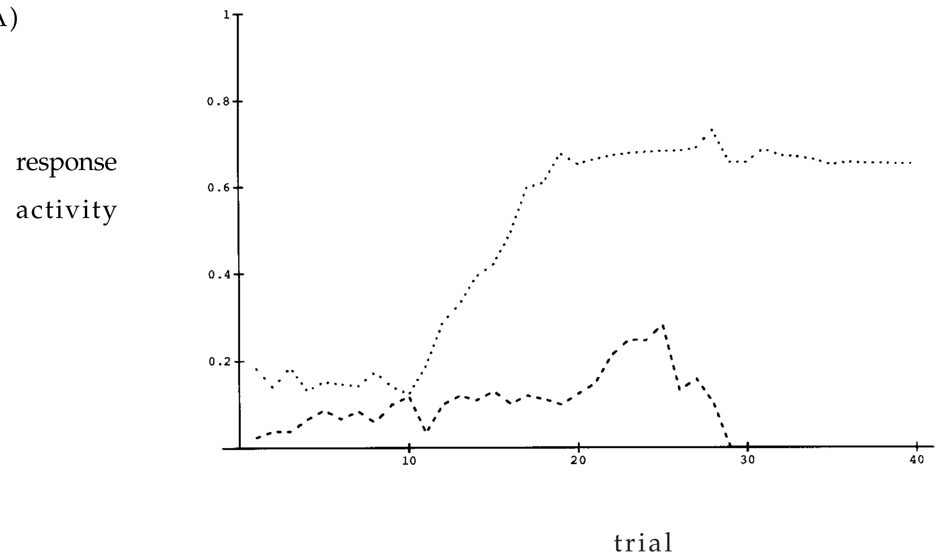


Figure 10: Overall activity response to the stimulus coding for the "Right" response (corresponding to Fig. 9). In this case, the increase in activity leads that of the performance (see discussion).

activated by each of the stimuli. In addition, certain stimuli may activate their set of columns more strongly than other stimuli, due to the initial random selection of weights. This activity difference can give the stronger stimulus a slight advantage in the learning process, since a weight update is related to the degree of activation of the voting unit. Therefore, given that a significant overlap exists between groups of columns, as well as an activity bias towards one or more stimuli, the learning induced by the stronger patterns can often cancel out any learning caused by the weaker stimulus. In the original protocol, this interference is not as much of a problem, since incorrectly mapped stimuli are allocated a larger number of consecutive trials. Within the new protocol, the probability of a favorable set of trials is relatively low.

Figure 10 shows the overall activity curve corresponding to the Right response in the above experiment. It is interesting to note that the activity curve increases prior to the performance curve. This can be explained by looking closer at the individual unit participation for the Rightward mapping. In this case, only a single column takes on the task of performing this particular mapping. During the early stages of learning, the network quickly learns the other three mappings. This

(A)



(B)

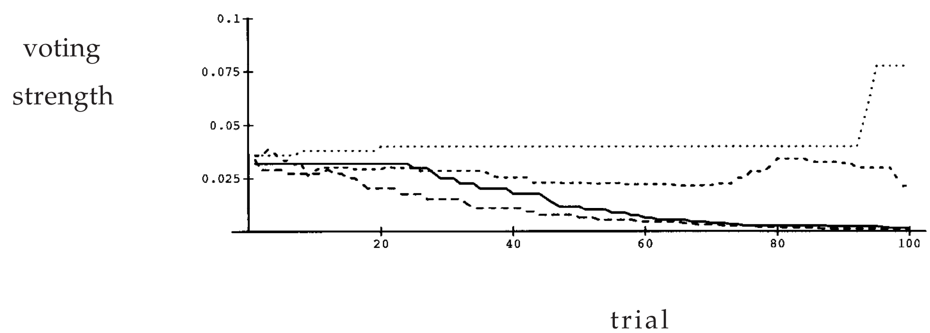


Figure 11 : (A) Activity of the one voting unit that learned to perform the mapping plotted against trial number. Because this is the only unit that learns the mapping, it determines the curve of Figure 10. (B) Evolution of the voting strength of the same unit. Only towards the end of the experiment (95th trial) does the unit discover the correct motor program selection unit.

particular column initially responds to both the Rightward and Downward stimuli (Figure 11 A). When the Rightward stimulus is presented, the support to the columns is so weak that the system does not make a decision in the allotted time. Therefore, the input/feature weights are adjusted to maintain recognition of the Rightward stimulus. As shown in Figure 11B, the system finally discovers the correct motor program to output at about trial 95. At this point, though, it still significantly supports the Downward response, but not enough to make incorrect decisions.

Reversal Experiments

Another set of experiments performed on the model asked about the system's behavioral and neural responses after a reversal takes place. In this experiment the network is presented with the standard set of four novel stimuli. After a given number of trials, the teaching system switches the mapping of

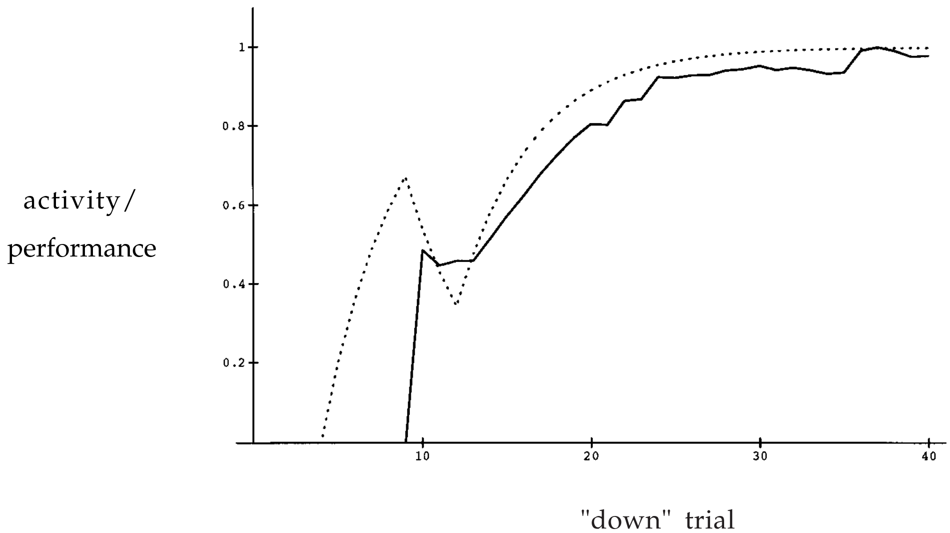


Figure 13: activity/performance curve for the reversal case ("Down" motor mapping). The solid (activity) curve corresponds to the overall activity in response to the stimulus that maps to the "Down" motor response, which switches between the 9th and 10th trials. The drastic change in overall activity after the reversal indicates that two separate sets of columns are being used to process the two different stimuli (recall that overall activity is measured by comparing the current activity pattern of the voting units to their activity pattern after learning is complete, in this case, trial 40). This also shows that the column continues responding to the same input before and after the reversal.

```
N : + +                D D D D D D R + + + + + + + + + + + + + +

L : N + R + + + +      + + + + + + + + + + + + + + + + + + + + +

R : N + L D L N + +    + + + + + + + + + + + + + + + + + + + + +

D : L L N N + + + + +  N N R + + + + + + + + + + + + + + + + + +
```

Figure 12: Behavioral response during a reversal task. The break in the strings indicates the point at which the reversal (between the No-Go and Down responses) takes place.

two responses. In this case, the stimulus that originally mapped to the No-Go motor response, now maps to the Down motor response, and vice versa. In looking at this experiment, we are interested in seeing how quickly the network is able to recover from the change in mapping and in understanding the underlying neural basis for this change. Figure 12 shows the behavioral results of one such experiment. After 26 trials, the visual/motor mapping had been learned perfectly for all cases. The first few responses that are generated after the reversal correspond to the original mapping. The system requires only a few trials of negative reinforcement to the Left and Right responses before the original mappings lose their dominance. At this point, the system continues its search as in the other experiments.

The activity/performance curve for the Left response is shown in Figure 13. Recall that the activity curve is computed by taking the dot product between current activity of the voting unit vector and the same vector of activity after the learning is complete. The sudden jump in the activity curve indicates point at which the reversal takes place. This jump happens because although the column

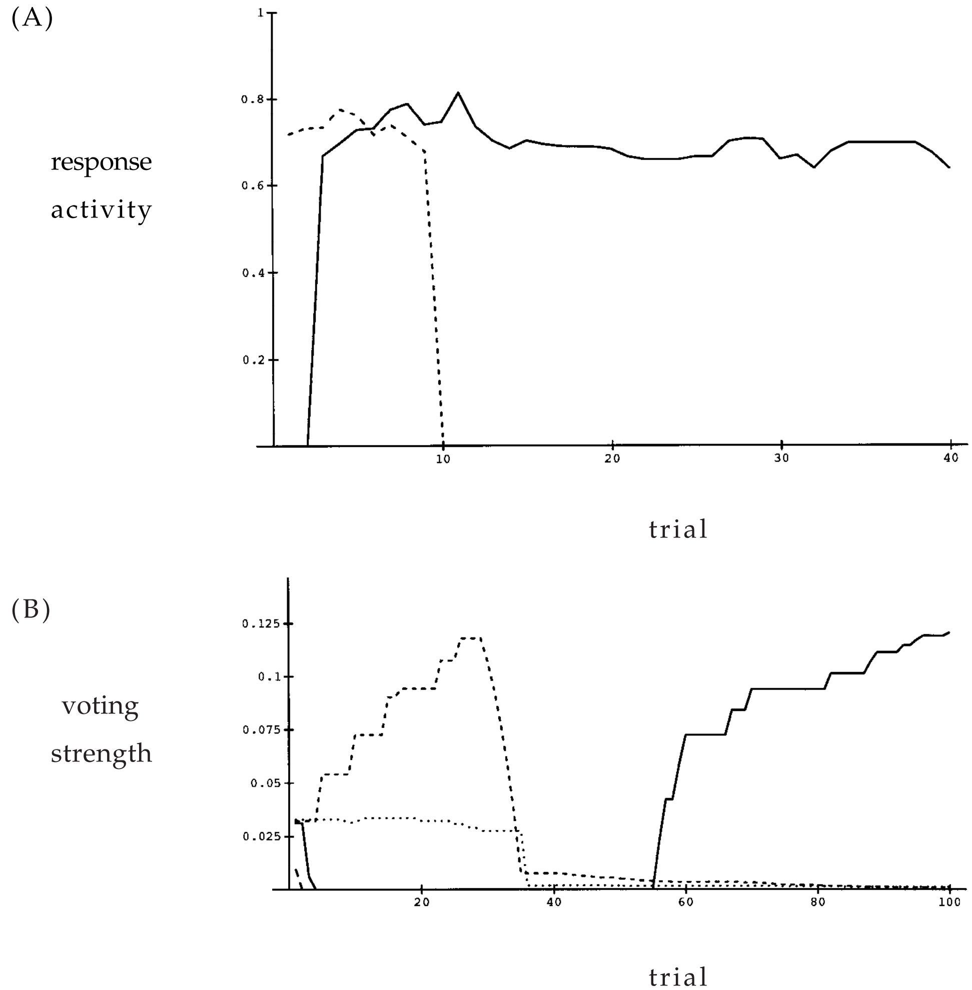


Figure 14: (A) A single unit's response to the various input stimuli over time. The solid curve represents the unit's response to the Nth occurrence of the stimulus that maps to the "No-Go" response. Likewise, the dashed curve represents the unit's response to the Nth occurrence of the stimulus that maps to the "Down" response. The drastic increase of the solid curve and decrease of the dashed curve indicate the point of reversal (after the 2nd occurrence of the stimulus that maps to "No-Go", and the 9th occurrence of the stimulus that maps to "Down, respectively). Note that the unit continues to respond to the same stimulus after **the reversal, although the stimulus now maps to a different motor program.**

continues to respond to the same stimulus, the stimulus is now supposed to map to the No-Go response (which has also been plotted). This and Figure 14 A demonstrate that the column maintains its mapping to the specific stimulus. Figure 14 B (the output weights from the same column) demonstrates that it is these weights that are adjusted to deal with the new mapping. Note in this figure, the reversal takes place over just a few trials (both in the reduction of the Downward weight and the increase of the No-Go weight.

Discussion

Our model has primarily addressed the computational issues involved in learning appropriate stimulus/motor program mappings. However, we believe that the functional role of voting units within our network may be related to that of set units within the premotor cortex. The actual visual/motor mapping is considered to be taking place further upstream from premotor cortex (within the $W_{in,feature}$ weights in our model). We believe this to be the case, due to the fact the Mitz et al. observed similar set unit activity patterns in response to *different* visual stimuli that mapped to the same motor response.

The motor programs, themselves, are most likely stored in regions further downstream from premotor cortex, as is the circuitry that chooses a single motor program to execute (motor program selection units and the winner-take-all circuit of our model).

This model was successful in meeting a number of the challenges set forth earlier It produces a behavior similar to that which was seen in the monkey experiments (goals 1a - c), and also produces normalized activity/performance curves that are qualitatively similar to the experimental data. Although neither of these two challenges (goals 2a and b) were explicitly designed into the neural algorithm, the two features dropped out of the original formulation of the model. Finally, the model produces neuronal activity phenomena that are representative of those observed by Mitz et al. (Mitz challenges a-c).

The primary computation within the model was performed using distributed coding of the information, thus demonstrating that not all of the relevant information need be present at a single location to perform a complex task. Rather, a distributed set of computers, each acting with a limited set of information, is capable of producing a global decision through a voting mechanism. However, in this model, votes were cast in a more centralized manner than is appropriate for a more faithful model of the brain's circuitry.

The concept of the column served to bind together a minimal set of computational capabilities needed to perform the local computation. This structure was then replicated to solve the more global computation. The claim here is not that a cortical column in the neurophysiological sense consists strictly of feature detector and voting units, but that a local organization is sufficient to perform a significant part of the computation. Allowing all neurons to connect to all other neurons is not practical from a hardware standpoint, and may impede the learning process.

The learning algorithm was a local one. Except for the reinforcement signal, the update of a particular weight only used the information available locally (the activation of the presynaptic and postsynaptic neurons, and the surrounding weights that shared common dendritic tree). This feature adds to the biological plausibility of the process, and may also have important consequences such as easy implementation in VLSI. In addition, the learned function was stored in a local manner (any particular column was active for only a subset of the inputs). This type of representation can limit the amount of interference between different input patterns, and thus the learning may be faster and more effective in achieving its goal. A first hint of this is demonstrated in our backpropagation experiments.

The model, however, does not attempt to account for the different types of units observed within the premotor cortex (goal 3). In particular, Mitz challenges d and e are not in general satisfied by the model (multiple stimulus patterns that map to the same motor response do not necessarily activate the same set of columns). This is due to the normalization operation that is performed on the input to the feature detector units. Again, in the premotor cortex of monkey, one would expect a set unit to continue participating in the same motor program after a reversal has taken place, rather than responding continually to the same input stimulus. This would be due in part to the fact that the monkey has already created and solidified its motor programs in memory (during the first stage of learning). Because the mapping from visual stimulus to motor program is transient (and/or most recent), the synaptic changes should be more likely taking place at this location.

Finally, the behavior of the model under different experimental conditions may yield some predictions as to the monkey's behavior under similar conditions. As discussed earlier, the use of a completely random sequence of stimuli (as opposed to repeating trials in which the incorrect response was given) significantly hindered the system's ability to learn the visual-motor mapping. From this observation, we would like to posit that the monkey would suffer a similar fate given the completely random trial presentation. This is not meant to say that the monkey would necessarily be unable to learn the task, but that the learning would at least be significantly more difficult. The degree to which this is true can ultimately feed back to future work on this model, since it would tell us something about the degree of interference between the different mappings.

Acknowledgments

This work was supported in part by a fellowship from the Graduate School, the School of Engineering, and the Computer Science Department of the University of Southern California, and in part by a grant from the Human Frontiers Science Program. We thank Steven Wise and Andy Mitz for correspondence which formed the basis for this project, and George Bekey for his help in the shaping of this document. In addition, we would like to thank Rob Redekopp for his aid in performing some of the backpropagation experiments.

Our model has been implemented in the Neural Simulation Language (NSL) (Weitzenfeld, 1991), and executes on Sun workstations. Both NSL and this model are available via anonymous ftp from the Brain Simulation Laboratory at USC (yorick.usc.edu). For further information, email may be sent to the author: ahfagg@robotics.usc.edu.

References

Amari, S., & Arbib, M. A. (1977). Competition and Cooperation in Neural Nets. In J. Metzler (Eds.), Systems Neuroscience Academic Press.

Arbib, M. A. (1989). The Metaphorical Brain 2: Neural Networks and Beyond. New York: Wiley-

Interscience.

Barto, A. G., Sutton, R. S., & Anderson, C. W. (1983). Neuron-like Adaptive Elements That Can Solve Difficult Learning Control Problems. IEEE Transactions on Systems, Man, and Cybernetics, **SMC-5**, 834-46.

Grossberg, S. (1976). A Theory of Visual Coding, Memory, and Development: Part 1. Parallel Development and Coding of Neural Feature Detectors. Biological Cybernetics, **23**, 121-134.

Hinton, G. E., & Sejnowski, T. J. (1986). Learning and Relearning in Boltzmann Machines. In J. L. McClelland & D. E. Rumelhart (Eds.), Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations (pp. 282-317). Bradford Book/The MIT Press.

Mitz, A. R., Godshalk, M., & Wise, S. P. (1991). Learning-Dependent Neuronal Activity in the Premotor Cortex. Journal of Neuroscience, **11**(6), 1855-72.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning Internal Representations by Error Propagation. In D. Rumelhart & J. McClelland (Eds.), Parallel Distributed Processing: Explorations in the Microstructure of Cognition (pp. 318-62). Bradford Books/MIT Press.

Rumelhart, D. E., & Zipser, D. (1986). Feature Discovery by Competitive Learning. In J. L. McClelland & D. E. Rumelhart (Eds.), Parallel Distributed Processing: Explorations in the Microstructure of Cognition Volume 1: Foundations (pp. 151-193). Bradford Books/The MIT Press.

Weitzenfeld, A. (1991). NSL - Neural Simulation Language Version 2.1 (TR No. CNE-91-05). Center for Neural Engineering, University of Southern California, Los Angeles, CA.

Appendix A: Network Data Structures

The following is a list of data structures used by the model (excluding parameters, which are discussed in the next section).

```
                                /*  Patterns.  */
VECTOR(inputs, NUM_INPUTS);                    /* Current vector of inputs.   */
MATRIX(P_Input, MAX_PATTERNS, NUM_INPUTS);     /* Set of all input patterns.  */
int num_pats;                                  /* Number of patterns stored in above.  */
VECTOR(P_Output, MAX_PATTERNS);                /* Corresponding set of expected outputs.*/
int patterns_loaded_flag;
                                /*  Feature units.  */
VECTOR(feature_mem, NUM_COLUMN);               /* Membrane potential. */
VECTOR(feature, NUM_COLUMN);                   /* Firing rate. */

                                /*  Voting units.  */
VECTOR(voting_mem, NUM_COLUMN);                /* Membrane potential. */
VECTOR(voting, NUM_COLUMN);                    /* Firing rate. */
DATA(threshold_v);                             /* Threshold (varying over time) */

VECTOR(voting_participation, NUM_COLUMN);      /* Binary pattern of which */
                                               /*  columns are active */
VECTOR(voting_contribution, NUM_COLUMN);       /* Output of voting units for learning. */


VECTOR(noise, NUM_COLUMN);                     /*  Noise process for columns.  */

VECTOR(motor_noise, NUM_CHOICES);              /*  Motor Program Selection Noise process*/

                                /*  Motor program selection units.  */
VECTOR(motor_inputs, NUM_CHOICES);             /*  Net input into the motor units.  */
VECTOR(motor_mem, NUM_CHOICES);                /*  Membrane potential.  */
VECTOR(dmotor_mem, NUM_CHOICES);               /*  Change in membrane potential. */
VECTOR(motor, NUM_CHOICES);                    /*  Firing rate.  */

                                /*  WTA Inhibitory neuron.  */
DATA(S_mem);                                   /*  Membrane potential.  */
DATA(S);                                       /*  Firing rate.  */

                                /*  input->feature mapping.  */

MATRIX(W_in_feature, NUM_INPUTS, NUM_COLUMN);   /*  Weights.  */
MATRIX(dW_in_feature, NUM_INPUTS, NUM_COLUMN);  /* Change in feature detector weights. */
MATRIX(W_in_feature_mask, NUM_INPUTS, NUM_COLUMN);  /* Matrix of 1s and 0s: determines */
                                                    /*  connectivity */

                                /*  voting->motor mapping.   */
MATRIX(W_vote_motor, NUM_COLUMN, NUM_CHOICES);  /*  Weights.  */
MATRIX(dW_vote_motor, NUM_COLUMN, NUM_CHOICES); /* Change in feature detector weights. */
MATRIX(W_vote_motor_mask, NUM_COLUMN, NUM_CHOICES);  /* Matrix of 1s and 0s-connectivity */


int winner;                                    /*  The current winner.  */
int current_pat;                               /*  Current pattern presented.  */
int counter;                                   /*  Trial number.  */
int time_counter;                              /*  Timestep for current trial. */
int timer_flag;                                /*  On completion, set to 1 if at least */
                                               /*  1 MPSU is active, -1 if none.      */
DATA(display_set_unit_flag);            /*  If true, then display set unit activation.  */
DATA(repeat_mode);                      /*  If true, then repeat stimulus when */
                                        /* response is incorrect.            */
DATA(stable_detect_threshold);          /* Changes in activity must be less than*/
                                        /* threshold for equilibrium to be reached. */
```

Appendix B: Network Parameters

The following section outlines the set of parameters used to produce the results presented in this paper. Table 1 shows the complete list of parameters and the values used in the simulation.

A number of parameters play a crucial role in the behavior of the network. These are further discussed here :

W_in_feature_probability determines how likely that a connection exists between an input unit and a feature unit. For this work, it was important to keep this parameter at a low value (0.3). This serves to minimize the number of columns that will respond at all to an input stimulus, thus minimizing the interference between columns. If set too low, not enough columns will react to a particular input.

input_weight_bias determines the distribution of weight values for those weights that do exist. A high value forces the existing weights synapsing on a particular feature to be very similar. On the other hand, a low value causes the weights to be more randomly distributed. In the case of our simulation, this value is set to 1.0 (a low value), yielding a reasonable distribution that allows different columns to respond differently to an individual stimulus. Thus, the weight initialization procedure biases the symmetry breaking between stimuli that goes on during the learning process.

noise_gain determines the magnitude of noise injected into the voting units. It is important that this value is significantly less than **init_threshold_v**. Otherwise, the voting unit may fire spontaneously (without feature unit support) before the threshold is lowered.

noise_change_probability is set such that the noise value changes slowly relative to the time constant of the voting unit (**u_voting**). When the noise changes at this time scale, on average, the effects of the noise are allowed to propagate through the system before the noise value changes again. Thus, in the early stages of learning, different groups of voting units may fire given the same input stimulus, allowing the system to experiment with what the appropriate set of voting units might be. If the noise changes too quickly, then the average affect will be very little noise injected into the system. Therefore, all eligible columns will fire together, and not in different subsets.

The constraints on **motor_noise_gain** and **threshold_m** are similar.

lrate_f determines how much effect that one trial will have on the weight matrix that maps from the input units to the feature units (the value used in these simulations was 0.4). When set too low, the slope of the overall activity curve begins to decrease and the system will take longer before it achieves perfect performance. On the other hand, setting this parameter too high will amplify the interference between the various weights (this is critical during the early stages of learning). Thus, the learning of one pattern may erase (in one trial) the information associated with another pattern.

lrate_v is the learning constant for the vote-to-motor weight matrix (the value used was 0.035). Setting this constant too high, will cause the system to very quickly commit columns to particular motor responses. The result is that the network is able to learn the mapping much quicker than in the cases discussed in this paper. Although it appears to be advantageous to use a higher parameter value, we

Network Parameters		
num_columns	30	Number of columns in the middle layer.
num_inputs	14	Number of inputs into the columns.
num_choices	4	Number of motor program selection units (no-go, left, right, down)
NSL Parameters		
delta	0.01	Integration step
Weight Initialization		
input_weight_bias	1.0	Constant added to random weight value (see weight initialization)
W_in_feature_probability	0.3	Probability that a particular weight will exist.
voting_weight_bias	4.0	
W_vote_motor_probability	1.0	
normalize_input_mode	1	Determines whether postsynaptic or presynaptic normalization is used for this set of weights (0 = postsynaptic; 1 = presynaptic).
Feature detector parameters		
threshold_f	0.1	Threshold
u_feature	0.05	Time constant
Threshold Modulator		
init_threshold_v	0.2	Initial threshold (threshold is determined by the threshold modulator).
u_threshold_v	4.0	Time constant of threshold modulator.
Voting Unit		
u_voting	0.05	Time constant
noise_gain	0.045	Injected noise.
noise_change_probability	0.01	Determines how often the injected noise term changes value.
Motor Program Selecion Unit		
u_motor	2.0	Time constant
motor_noise_gain	0.05	Injected noise.
motor_noise_change_prob.	0.01	Determines how often the injected noise term changes value.
threshold_m	0.035	Motor program unit threshold.

Table 1: Network Parameters

would move away from the behavioral results seen in the Mitz experiments. In addition, the network may become more sensitive to interference, a problem that will show itself as the task difficulty is increased.

negative_factor_f scales the effect of negative reinforcement on the network. When this value approaches 1, the effect of a negative signal can be devastating to the network (see discussion of

S (Inhibitory Unit)		
u_S	0.5	Time constant of membrane potential.
threshold_S	0.1	Unit threshold.
Learning Parameters		
lrate_v	0.035	Voting/motor program selection unit lrate
lrate_f	0.4	Input/feature detector unit lrate
negative_factor_f	0.25	Input/feature factor for negative reinforcement
L1_norm_mode	1	1 indicates L1-normalization is used (0 indicates L2-normalization).
Protocol Parameters		
first_pole_mode	1	1 indicates first-passed-the-pole mode is turned on.
repeat_mode	1	1 indicates stimuli are repeated when an incorrect response is generated by the system.
max_time_counter	200	Maximum number of time steps alotted to the system for making a decision.
Analysis Parameters		
α	0.8	Used to compute average performance.
Simulation Parameters		
display_participation_mode	0	1 indicates that the participation vector is printed to the screen at the end of each trial.
collect_mode	0	If 1, collecting statistics.
	1	If no MPSUs are active at time of punishment, then reward to get voting activity up.

Table 1 (cont): Model Parameters

learning dynamics). In general, we found that too high of a value will decrease the slope of the overall activity curve (evident when the network begins to produce the correct answer, but then tries other responses).

Training Patterns

The patterns shown in Table 2 were used to train the network for most of the above experiments. The right-hand column denotes the expected motor response. For this case, the input patterns are orthogonal. Other training sets that were used for the comparison with backpropagation included overlapping patterns. One such training set is shown in Table 3.

Training Pattern	Expected Response
1 1 1 0 0 0 0 0 0 0 0 0 0 0	No-Go
0 0 0 0 0 0 0 0 0 0 1 1 1 0	Left
0 0 0 0 1 1 1 0 0 0 0 0 0 0	Right
0 0 0 1 0 0 0 1 0 1 0 0 0 0	Down

Table 2: Input patterns and expected motor responses.

Training Pattern	Expected Response
1 1 1 0 0 0 0 0 0 0 0 0 0 0	No-Go
1 0 0 0 0 0 1 0 0 1 0 0 0 0	Left
0 0 0 1 0 1 1 0 0 0 0 0 0 0	Right
0 0 0 1 0 0 0 1 0 1 0 0 0 0	Down

Table 3: Input patterns and expected motor responses (more difficult case). Each of the patterns overlaps at least one other pattern.

Appendix C: Sample Run

The following is an example learning session.

```
pax.usc.edu(2):nsl                                                    (C.1)
NSL Version 2.1.2
Initializing NSL Simulation System
LEARN : @(#)learn.c     1.4 12/9/92
NSL_MAIN : @(#)nsl_main.c       1.4 12/9/92

nsl> load startup                                                     (C.2)
load startup
load learn
set network LEARN
set delta 0.01
set end_time 10000.0
.
.
.

New seed => 10                                                        (C.3)
user seed
Pattern file name => a1.pat                                           (C.4)
num_pats = 4
read 4 patterns
user getpat
Cleared
init
Log file name => a.log                                                (C.5)
Opened.
user collect
Wlog file name => a.wlog                                              (C.6)
Opened.
user wlog

nsl> init                                                             (C.7)
nsl> init
nsl> step 1000                                                        (C.8)
nsl>
1: p0 s0 w0 + 0.00992368                                              (C.9)
0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0          (C.10)

200                                                                   (C.11)
No-pick no-go!                                                        (C.12)
2: p3 s3 w0 - 0.0132312
0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 1

200
No-pick no-go!
3: p3 s3 w0 - 0.0191826
0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 1

200
No-pick no-go!
4: p3 s3 w0 - 0.031615
0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 1

200
No-pick no-go!

nsl>
```

Once NSL has been compiled for the model, it is executed in the standard way (C.1). The best way to configure the system in preparation for simulation is to load the *startup* script (C.2), which loads in the standard set of parameters (learn.nsl), and prompts the user for a variety of parameters. The *seed*(C.3) is used to configure the random number generator (useful for forcing the same conditions for multiple experiments). The *pattern file* (C.4) contains a list of input patterns and the corresponding desired outputs. The *log file* (C.5) stores result information for each trial (including input pattern, model output, indication of correctness, overall activity level, and the voting unit activity levels on the final timestep). The weight matrix **W_vote_motor** is written to the *weight log* file (C.6) at the end of every trial.

Once configuration is complete, initialize the system (C.7) and begin execution (C.8). In this configuration, at each trial the system reports the following information: current trial, presented pattern number, expected motor output, selected motor output, indication of correctness (+/-), and a measure of total activity of the voting units (C.9). On the next line (C.10), the system displays the participation vector, which indicates those columns that were active at the end of the trial. On the following line (C.11), the number of timesteps required to obtain a decision is displayed (in this case, 200 is the maximum number of timesteps). When the system is unable to make a decision by the time the trigger stimulus is received, the message in (C.12) is also displayed.

To perform an entire experiment, continue executing the simulation (through additional 'step' commands) until the system has learned the mapping completely. A good indication of this is that all mapping have been learned, and all decisions are made in a very short amount of time (for the given parameters, 20-30 timesteps should be sufficient).

Appendix D: Obtaining the Model

ftp site: yorick.usc.edu

login id: anonymous

password: home user id

Download the following files:

/pub/NSL2.1/nsl.tar.Z	- NSL
/pub/NSL2.1/nsl_manual.ps.Z	- NSL manual.
/pub/NSL2-LIBS/alib.tar.Z	- Additional library of routines.
/pub/NSL2-MODELS/	
release.v1.tar.Z	- Version used to generate results in paper.

Setup:

install and compile NSL libraries (see NSL documentation)

install and compile alib (in directory support).

Configure the Makefile.

cd nsllibs; make all; make install

cd socket; make all; make install

compile model (in directory release.v1):

update nsl_link and nsl_makelink to contain the correct paths

nsl_link learn.c

execution: see README file in release.v1 directory