13 A Model of Primate Visual-Motor Conditional Learning[1]

A. H. Fagg and A. Weitzenfeld[2]

13.1 Introduction

Mitz, Godshalk and Wise (Mitz, Godshalk, and 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 13.1 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 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.

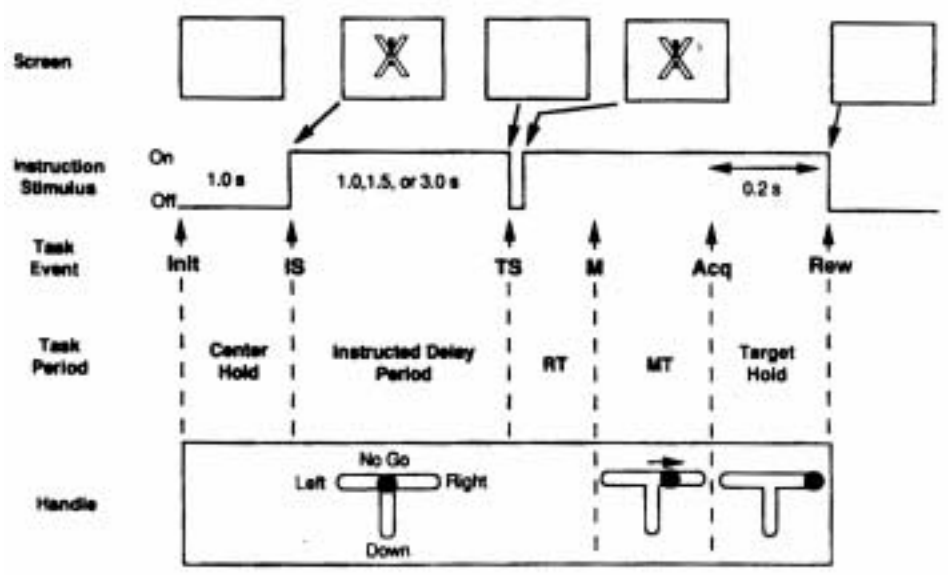

Figure 13.1
Top row: visual stimulus as seen on the video screen. Second row: temporal trace of the visual stimulus. Third and fourth rows: Primary events and periods of the experimental trial. Fifth row: expected motor response. (From Mitz et al., Figure 1; reprinted by permission of the Journal of Neuroscience.)

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 13.2 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.

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 13.2
Samples of responses to novel stimuli given example specific expected motor responses. Each row represents only those trials from an experiment that corresponds 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.)

This behavior may be captured at a high level by considering a separate *decision box* for each stimulus (A more formal treatment of these computing elements (stochasitic learning automata) may be found in Bush (1958) and Williams (1988)). 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 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 13.3 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.

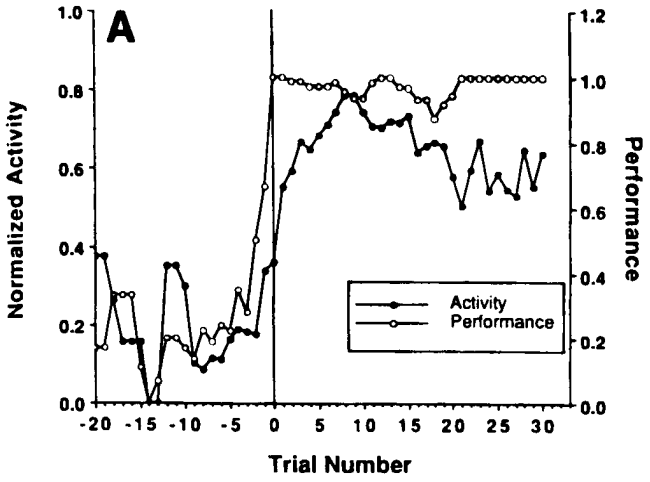

Figure 13.3
Normalized activity and performance curve plotted as a function of trial for the presentation of a novel stimulus. The rise in overall performance precedes that of cellular activity by about 3 trials. (From Mitz, figure 3; reprinted by permission of the Journal of Neuroscience.)

Mitz et al. (1991) 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 previous point (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 modeling 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 figure 13.2:
    a. The appropriate number of trials that are required to learn the mapping.
    b. Incorrect responses are sometimes given on several repeated trials.
    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 (figure 13.3). The performance curve leads the activity curve by a number of learning trials.
3. A complete model will also reproduce the temporal activity of various neurons in the premotor cortex, including: anticipatory units, signal-related units, set-related units, and movement-related units.

13.2 Model Description

Much of neural network research has concentrated upon supervised learning techniques, such as the generalized delta rule or backpropagation (Rumelhart, Hinton and Williams, 1986). In our modeling 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 with sigmoidal units 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 mapping and all weights are altered as a result of learning a single pattern.

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 for this unit, its response increases significantly for some stimuli, while it decreases for the remainder.

Network Dynamics

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 13.4). The overall network is composed of a large number of these columns, each performing a small portion of the stimulus-to-motor program mapping.

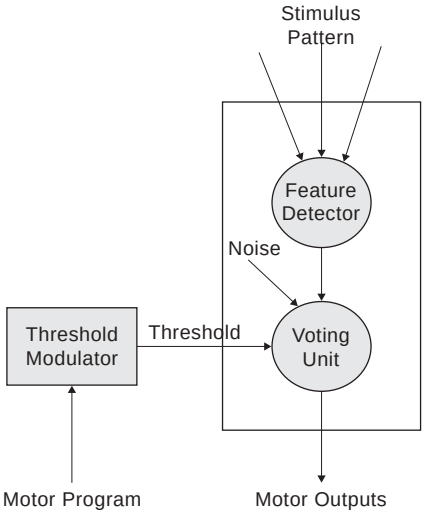

Figure 13.4
The motor selection column model. The feature detector detects specific events from the sensory input. The voting unit produces a vote for an appropriate set of motor programs. This unit, along with the noise and the threshold modulator, implements a search mechanism.

The feature detector recognizes small patterns (microfeatures) in 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.

The state of the feature detector units are described by the equations:

$$\tau_f \frac{d\,Feature_{mem}}{dt} = -Feature_{mem} - Threshold_f + W_{in,\,feature} * Inputs \qquad (13.1)$$

$$Feature = ramp(Feature_{mem})$$

where:

- τ_f is the time constant (scalar) of membrane potential change.
- $Threshold_f$ is the internal threshold of the feature detector units (a scalar).
- $Feature_{mem}$ is a vector of membrane potentials for the set of feature detector units. The initial condition (at the beginning of a trial) is $Feature_{mem}$ = - $Threshold_f$ for all elements of the vector.
- $W_{in,feature}$ is the weight matrix between the input stimulus and the feature detector units. These weights are updated by learning.
- *Inputs* is the vector of stimulus inputs.
- *Feature* is the vector of firing rates of the feature detector units.

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.

The behavior of the voting units is governed by the equations:

$$\tau \frac{dVoting_{mem}}{dt} = -Voting_{mem} - Threshold_v(t) + Feature + Noise \qquad (13.2)$$

$$Voting = saturation(Voting_{mem})$$

where :

- τ_v is the time constant of the voting units.
- $Voting_{mem}$ is the membrane potential of the voting units (vector). The initial conditions are $Voting_{mem}$ = -$Threshold_v(t)$ for all units.
- $Threshold_v(t)$ is the time-dependent threshold determined by the threshold modulator (a scalar).
- *Feature* is the vector of firing rates. Each voting unit receives input **only** from its corresponding feature unit.
- *Noise* is a low-amplitude noise process that changes slowly relative to τ_v (vector).
- *Voting* is the firing rate vector.

As shown in figure 13.5, 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.

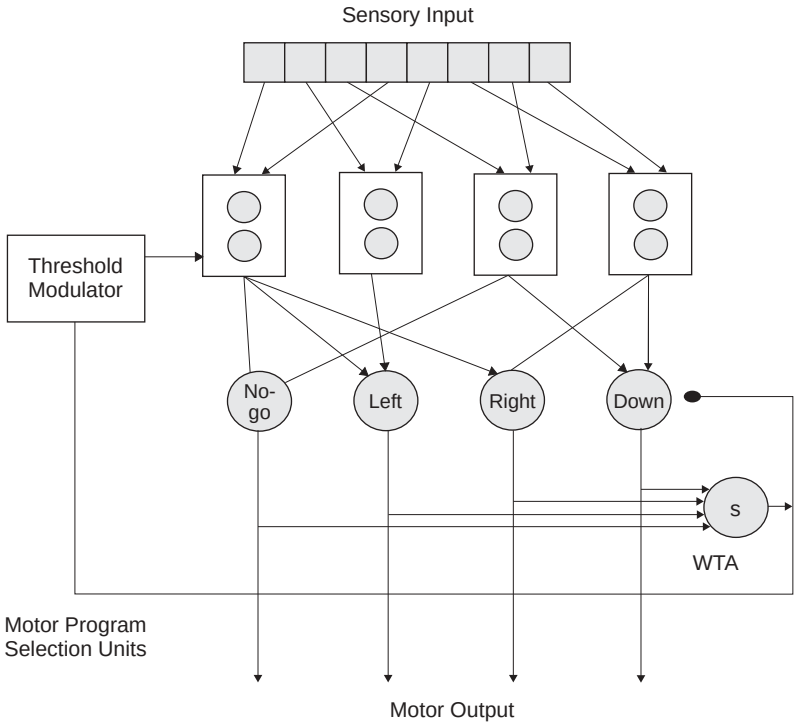

Figure 13.5
The motor program selection units label corresponds to the four action circles (no-go, left, right, down). A set of motor selection columns votes for the motor responses. The votes are collected by units representing the activity of the schemas for each of the legal motor responses. The winner-take-all circuit ensures that only one motor program is selected.

Depending upon the state of the voting units, the motor program selection units/ winner- take-all circuit attempts to choose a single motor program to activate. This selection process is governed by the following equations:

$$\tau_m \frac{dMotor_{mem}}{dt} = -Motor_{mem} - Threshold_m + W_{vote,motor} * Voting - S + Motor + Motor\_Noise \quad (13.3)$$

where :

- τ_m is the motor selection unit time constant.
- $Motor_{mem}$ is the membrane potential of the motor selection units (a vector). The initial conditions are $Motor_{mem} = -Threshold_m$ for all elements in the vector.
- $Threshold_m$ is the scalar threshold of the motor selection units.
- $W_{vote,motor}$ is the weight matrix representing the projection from the voting units to the motor selection units.
- S is the firing rate of the inhibitory neuron. The initial condition of this neuron is $S = 0$.
- *Motor* is the firing rate vector. Initially, Motor = 0 for all elements.
- *Motor_noise* is a low-amplitude noise process, that changes slowly relative to τ_m (a vector).
- 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).

$$\tau_S \frac{dS_{mem}}{dt} = -S_{mem} - Threshold_S + \sum_{i=1}^{N} Motor[i] \quad (13.4)$$

$$S = ramp(S_{mem})$$

- S_{mem} is the membrane potential of the inhibitory neuron (a scalar, since there is only one). The initial condition for this neuron is $S_{mem} = -Threshold_s$. $Threshold_s$ is the threshold of the inhibitory neuron.

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.

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.

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 $W_{vote,motor}$ corresponding to the fan-in and fan-out of figure 13.4, respectively. Only those columns that participate in the current computation adjust their weights. In the experimental setup, positive reinforcement is given when the monkey exhibits a correct response, but not otherwise. 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.

However, 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. Therefore, the currently active columns are 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. Without this additional term, negative reinforcement from the teacher is disastrous. The negative reinforcement further decreases the response 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 decreases 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). It is at this point that the symmetry between the two is broken.

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. 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. We thus have:

$$lgain = \begin{cases} negative\_factor\_f & \text{if } reinforcement < 0 \\ 1 & otherwise \end{cases}$$

$$\Delta W_{in,\,feature} = reinforcement * lgain * lrate_f * \left(\left(Input \cdot Voting^T \right) \wedge W\_in\_feature\_mask \right) \quad (13.5)$$

where:

- $lrate_f$ is the learning rate coefficient for the stimulus-to-feature mapping
- $Input \cdot Voting^{\mathrm{T}}$ is the outer product of the Input and Movement vectors.
- *lgain* scales the effect of negative reinforcement relative to positive reinforcement.
- $\wedge$ is a point/wise multiplication operator.
- *W_in_feature_mask* is a binary matrix indicating the existence of a synapse.

In this case, the effect of negative reinforcement on the weights is intended to be less than that of positive reinforcement. This is done because negative reinforcement can be very devastating to columns that are just beginning to learn an appropriate mapping.

To simultaneously weaken those weights that are not strengthened by reinforcement, we then set:

$$W_{in,\,feature} = Normalize\left(W_{in,\,feature} + \Delta W_{in,\,feature} \right) \quad (13.6)$$

where:

- *Normalize*() is a function that L1-normalizes the vector of weights leading into each feature detector unit to length 1, given by

$$Y_i = \frac{X_i}{\sum_j |X_j|} \quad (13.7)$$

- *W_in_feature_mask* is a matrix of ones and zeros that determines the existence of a weight between the corresponding voting and motor program selection units. The

elements of this matrix are point-wise multiplied with those of $\Delta W_{in,feature}$ to mask out weight deltas for weights that do not exist.

Equation (13.2) produces a competition between the weights associated with a particular unit. Thus, the weights are self-regulating, forcing unneeded or undesirable weights to a value near zero. If a column continues to receive negative reinforcement (as a result of being involved in an incorrect response), then it becomes insensitive to the current stimulus, and is reallocated to recognize other stimuli.

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 is weakened, allowing the other synapses from the voting unit to strengthen slightly through normalization. Thus, more voting power is allocated to the other alternatives:

$$\Delta W_{vote,motor} = reinforcement * lrate_v * \left(\left(Voting \cdot Motor^T \right) \wedge W\_vote\_motor\_mask \right) \quad (13.8)$$

$$W_{vote,motor} = Normalize\left(W_{vote,motor} + \Delta W_{vote,motor} \right) \quad (13.9)$$

where:

- $lrate_v$ is the learning rate coefficient for the voting-to-motor response mapping.
- *W_vote_motor_mask* is a weight matrix mask similar to the mask that appears in (13.2).

A similar type of reinforcement learning is utilized in Barto et al. (1983, see later discussion).

$W_{in,feature}$ and $W_{vote,motor}$ are initially selected at random. 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 deserve its votes given its view of the sensory input. Equation (13.1) 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, thus giving the column a larger voting power. The feature detecting algorithm is related to the competitive learning of von der Malsburg (1973) and Grossberg (1976) (discussed further in Rumelhart and 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.

Note that 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.

13.3 Model Implementation

The model is implemented by three top level models, **CondLearn** module, **TrainFile** module and **CondLearnModel** as shown in figure 13.6.

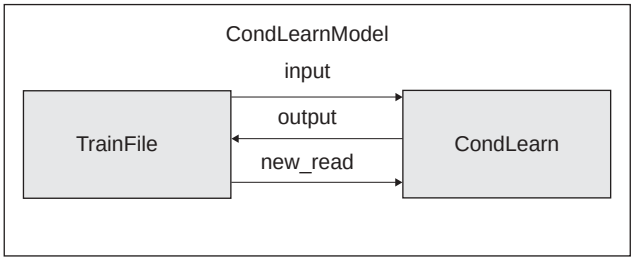

Figure 13.6
Conditional Learning model modules. CondLearn module where dynamics are described, TrainFile module where training data are read, and CondLearnModel which instantiates and connects the modules.

Model

The complete model is described in **CondLearnModel**. It is responsible for instantiating two modules, the **CondLearn** and **TrainFile** modules.

```
nslModel CondLearnModel ()
{
    private TrainFile tf();
    private CondLearn cl();
}
```

The **initModule** methods perform model initialization by reading training data and instantiating the number of layer elements dynamically specified.

```
public void initModule()
{
tf.readFile();
    NslInt0 inSize = tf.getValue("inSize");
    cl.memAlloc(inSize.getValue(),
    num_columns.getValue(),num_choices.getValue());
}
```

Note in this instantiation how we obtain the number of patterns from the **TrainFile** module as it reads this value from the training data file, and only then do we pass it to the **CondLearn** module to be used to instantiate data arrays. While the number of columns and number of choices are directly specified by the user, the input size is read from the training file. These sizes are then used to call all **memAlloc** methods in the model.

Train Module

The **TrainFile** module contains input and output ports for interconnections with the **CondLearn** module. It stores training data in memory similar to the **BackProp Train** module.

```
nslModule TrainFile()
{
public NslDinInt0 new_read();
    public NslDoutInt1 input();
    public NslDoutInt0 output();

    private NslFloat2 pInput();
    private NslFloat1 pOutput();
}
```

The **readFile** method reads the training data while the **intTrain** picks a new random pattern during each new epoch

```
public void initTrain()
{
    int pat = nslIntRand(numPats);
    input = pInput[pat];
    output = pOutput[pat];
}
```

CondLearn Module

The **CondLearn** module contains a number of submodules, **Feature**, **Vote**, **Motor**, **Threshold** and **WTA** (winner take all) modules, as shown in figure 13.7.

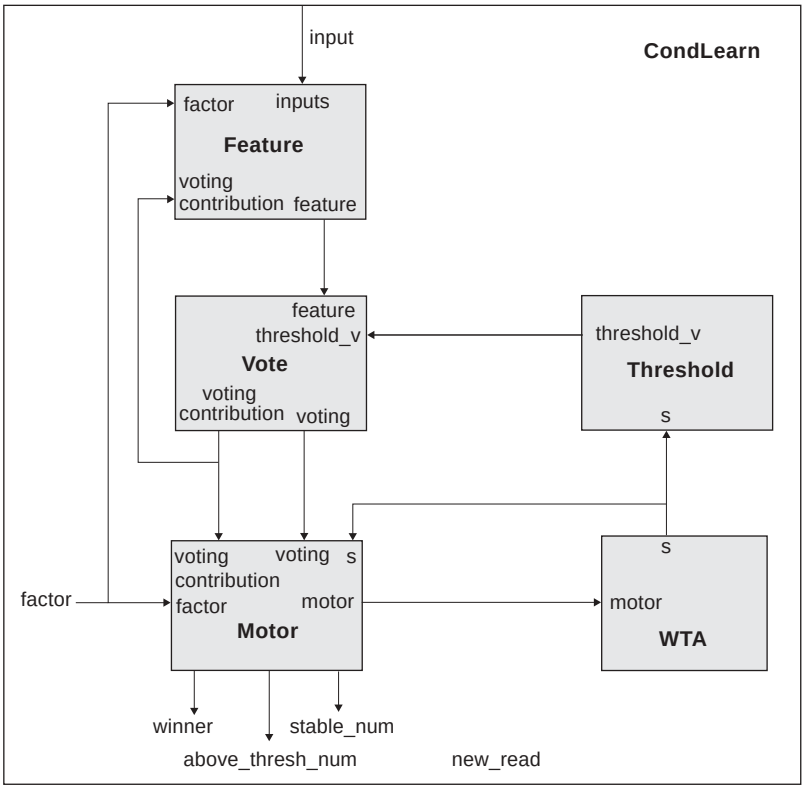

Figure 13.7
CondLearn module

The neural computational model for these modules are implemented using the leaky-integrator model (e.g., Arbib 1989), in which each neuron is represented by a membrane potential and a firing frequency. In the following set of equations, neural states are represented as vectors. Two vectors are connected through a set of weights through either a one-to-one connection scheme, or a fully-connected scheme (all neurons of the input layer are connected to each of the neurons of the output layer). The first case is represented by a vector addition, while the second case is represented by multiplying the vector *I* with a "mask" of synaptic weights *W* to yield *W@I*. The network is initialized by randomizing the input-to-feature and voting-to-motor projections and prepares the model to begin execution.

```
nslModule CondLearn()
{
    private Feature feature();
    private Threshold threshold();
    private Vote vote();
    private Motor motor();
    private WTA wta();
    public NslDinInt1 input();
    public NslDinInt0 output();
    public NslDoutInt0 new_read();
    public NslDoutInt0 factor();
    public NslDinInt0 above_thresh_num();
    public NslDinInt0 stable_num();
    public NslDinInt0 winner();
}
```

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

```
public void simTrain()
{
    if(above_thresh_num.getData() == 1 && (stable_num.getData()
       == _num_choices || first_pole_mode.getData() != 0.0))
{
    timer_flag = 1;
       punish_reward_func(winner.getData());
       system.breakCycles();
    }
}
```

The state of the motor program selection units is checked to determine whether or not the system is itself ready to generate an output. The model is ready to output a motor response in one of two cases, depending upon the state of the **first_pole_mode** flag. 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.

```
public void endTrain() // Timeout: NO-GO case
{
if (timer_flag == 1)
    return;

    if(above_thresh_num.getData() == 0)
        timer_flag = -1;
    else
    timer_flag = 1;

    punish_reward_func(NO_GO_CASE);
}
```

The trial is terminated and **timer_flag** is set to 1. Termination of the current trial may also be forced if the go signal has arrived. As stated earlier, two cases are possible: no motor program selection units active or more than one active. In either case, the trial is terminated and the weight matrices are updated. The reinforcement is set to 1 if the **timer_flag** = -1, meaning no winner has been found yet. Reward and punishment are the reinforcement signals used to update the weight matrices if there is at least one winner. Weights are modified in the **feature** and **vote** modules, respectively.

The **punish_reward_func()** routine selects a new input pattern to present to the system in preparation for the next trial.

```
public void punish_reward_func(int win)
 {
factor = 1;

if(output.getData() == win)
       new_read = 1;
   else
   {
       new_read = 0;
       if (timer_flag != -1)
           factor = -1;
   }
}
```

Feature Module

The **Feature** module computes the membrane potential **feature_mem**, the firing rate **feature** of the feature detector units and the feature detector weights, most important ones are **w_in_feature**, **w_in_feature_mask**, and **dw_in_feature**. It receives input **pInput** and the **voting_contribution** for learning, while its output is **feature**

```
nslModule Feature ()
{
    public NslDinFloat1 inputs();
    public NslDinFloat1 voting_contribution();
    public NslDinFloat0 threshold_v();
    public NslDinInt0 factor();

    public NslDoutFloat1 feature();

    private NslFloat1 feature_mem();

    private NslFloat2 w_in_feature();
    private NslFloat2 dw_in_feature();
    private NslFloat2 w_in_feature_mask();
    private NslFloat0 negative_factor_f();
}
```

The **initModule** method initializes the feature detector weights. Initially, the module configures the input-to-feature weight matrix **w_in_feature**.

```
public void initModule()
{
    nslRandom(w_in_feature);
    select_random(w_in_feature_mask,w_in_feature_probability.
        getData());
    w_in_feature = (w_in_feature/2.0 + 0.5 + input_weight_bias)

        ^ W_in_feature_mask;
    normal_col(w_in_feature);
}
```

A random value is selected for each of the individual weights (uniform distribution in the interval [0,1]). Then existing physically connections are found. The call to **select_random()** initializes **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**. The **w_in_feature_mask** weight mask is applied 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.

A normalization is applied to **w_in_feature** weight matrix. 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 **simTrain** method processes the dynamics of the feature detector,

```
public void simTrain()
{
    nslDiff(feature_mem, u_feature, -feature_mem - threshold_f
       + w_in_feature * inputs);

    if (LIMITED_ACTIVITY_FLAG == true)
       feature = nslSat(feature_mem, 0, 1, 0, 1);
    else
       feature = nslRamp(feature_mem);
}
```

where **feature_mem** depends upon the **threshold_f**, and the matrix product of weight matrix **w_in_feature** by column vector **pInput**, this 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].

The **endTrain()** 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 allotted time and no motor program selection units were active, then the reinforcement signal is set to 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 **endTrain()**).

```
public void endTrain()
{
    float f_factor;

    if(factor.getData() < 0.0)
       f_factor = factor.getData() * negative_factor_f.
          getData();
    else
       f_factor = factor.getData();

    dw_in_feature = f_factor * lrate_f *
       vec_mult_vec_trans(voting_contribution,inputs) ^
    w_in_feature_mask);

    w_in_feature = nslRamp(w_in_feature + dw_in_feature);

    normal_col(w_in_feature,L1_norm_mode.getData());
}
```

In the feature detector, the information content of a negative reinforcement is much less than that 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, and leaving the positive reinforcement signal intact

voting_contribution identifies those columns that are currently participating in the computation and the degree to which they are participating. Only those connections which carry signals into or out of the active columns will change in strength. The update to the input-to-feature mapping is computed in. 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 (the call to **nslRamp()** ensures that all connection strengths are always positive), and the weights are normalized.

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).

Noise Module

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 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.

```
nslModule Noise ()
{
   private Noise noise();
}
```

Noise initialization.

```
public void initExec()
{
   randomize(noise);

   noise = noise_gain * noise;
}
```

In the **noise** vector is initialized.

Noise is modified if necessary in the **simExec** method

```
public void simExec()
{
    if(random_value2() < noise_change_probability.getData())

    {
        randomize(noise);
    noise = noise_gain * noise;
    }
}
```

In the frequency at which a new noise vector (**noise**) is selected is determined by the parameter **noise_change_probability**.

If it is time to update the noise vector, a completely new vector is generated, and then scaled by the **noise_gain** parameter.

Vote Module

The **Vote** module computes the state of the voting units.

```
nslModule Vote ()
{
    private Noise noise();
public NslDinFloat0 threshold_v();
public NslDinFloat1 feature();
public NslDoutFloat1 voting();
public NslDoutFloat1 voting_contribution();

    private NslFloat1 voting_mem();
    private NslFloat0 voting_contribution_mode();
    private NslFloat0 voting_contribution_scale();
    private NslFloat1 voting_participation();
}
```

The **simTrain** method specifies local processing

```
public void simTrain()
{
   noise.simExec();
nslDiff(voting_mem, u_voting, -voting_mem - threshold_v +
   feature + noise.noise);

if (LIMITED_VOTING_ACTIVITY_FLAG == true)
      voting = nslSat(voting_mem, 0, 1, 0, 1);
   else
voting = nslRamp(voting_mem);

   voting_participation = nslStep(voting);

   if (voting_contribution_mode.getData() == LINEAR)
      voting_contribution = voting;
   else if (voting_contribution_mode.getData() == BINARY)
      voting_contribution = voting_participation;
   else if (voting_contribution_mode.getData() ==
      COMPRESSED_LINEAR)
     voting_contribution = nslSat(voting,0.0,
         voting_contribution_scale.getData(),0.0, 1.0);
   else if (voting_contribution_mode.getData() == JUMP_LINEAR)
      voting_contribution = nslSat(
       nslRamp(voting, 0.0,
   0.0,voting_contribution_scale.getData()));
   else
      nslPrintln("Unknown voting_contribution_mode:",
      voting_contribution_scale.getData());
}
```

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. 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.

The vector **voting_participation** is used to display to the user which columns are participating within any particular computation.

Threshold Module

The **Threshold** module implements the dynamics of the threshold modulator.

```
nslModule Threshold ()
{
    public NslDinFloat0 s();
    public NslDoutFloat0 threshold_v();
}
```

The **simTrain** method updates the threshold

```
public void simTrain()
{
    if(s <= 0.0)
        nslDiff(threshold_v,u_threshold_v, -threshold_v);
}
```

The threshold level, **threshold**, is initially set at the beginning of the trial to the parameter **init_threshold**. This value then decays exponentially. 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). In order for this event to occur, several voting units must have begun to fire giving the system the ability to make some sort of decision.

Motor Module

The motor selection unit dynamics are described within the **Motor** module.

```
nslModule Motor ()
{
    private Noise noise();
    public NslDinFloat1 voting();
    public NslDinFloat1 voting_contribution();
    public NslDinFloat0 s();
    public NslDinInt0 factor();

    public NslDoutFloat1 motor();

    public NslDoutInt0 above_thresh_num();
    public NslDoutInt0 stable_num();
    public NslDoutInt0 winner();

    private NslFloat1 dmotor_mem();
    private NslFloat1 motor_mem();
    private NslFloat1 motor_inputs();

    private NslFloat2 w_vote_motor();
    private NslFloat2 dw_vote_motor();
    private NslFloat2 w_vote_motor_mask();
    private NslFloat0 voting_weight_bias();
    private NslFloat0 w_vote_motor_probability();
    private NslFloat0 voting_factor();

    private NslFloat0 normalize_input_mode();
    private NslFloat0 stable_detect_threshold();
}
```

The **initModule** method initializes the voting weights

```
public void initModule()
{
    noise.initExec();
    winner = 0;

       randomize(w_vote_motor);
    select_random(w_vote_motor_mask, w_vote_motor_probability.
       getData());
    w_vote_motor = (w_vote_motor/2.0 + 0.5 +
       voting_weight_bias) ^
    w_vote_motor_mask;

    if (normalize_input_mode.elem() == 1.0)
       normal_col(w_vote_motor,L1_norm_mode.getData());
    else
       normal_row(w_vote_motor,L1_norm_mode.getData());
}
```

The vote-to-motor weights **w_vote_motor** are initialized in a similar manner. 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 motor selection unit dynamics are determined within the **simTrain** method.

```
public void simTrain()
{
    noise.simExec();
    motor_inputs = mat_mult_col_vec(w_vote_motor,
       voting)/_num_columns;
    dmotor_mem = -motor_mem - threshold_m +
       voting_factor * motor_inputs - s + motor + noise.noise;
    nslDiff(motor_mem, u_motor,dmotor_mem);
    motor = nslStep(motor_mem);

    above_thresh_num = 0;
    stable_num = 0;

    for (int i = 0; i < _num_choices; ++i)
    {
       if (motor(i) > 0.0)    // Above threshold
       {
          above_thresh_num = above_thresh_num + 1;
          winner = i;
       }
       if(fabs(dmotor_mem[i]) <
          stable_detect_threshold.getData())
          stable_num = stable_num + 1;
  }
}
```

The membrane potential of these units is a function of the votes from the feature detector units (**motor_inputs**), the inhibitory signal from the WTA (Winner-Take-All) inhibitory unit **s**, and an injected noise signal (**motor_noise**). 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. We consider that a selection has been made only when one motor program selection unit is firing.

The **endTrain()** 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 allotted time and no motor program selection units were active, then the reinforcement signal is set to 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 **endTrain()**).

```
public void endTrain()
{
    dw_vote_motor = factor * lrate_v *
    (vec_mult_vec_trans(motor, voting_contribution) ^
        w_vote_motor_mask);

    w_vote_motor = nslRamp(w_vote_motor + dw_vote_motor);

    if(normalize_input_mode.getData() == 1.0)
        normal_col(w_vote_motor,L1_norm_mode.getData());
    else
        normal_row(w_vote_motor,L1_norm_mode.getData());
}
```

A similar learning rule to that of the input-to-feature mapping 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. These delta values are then added into the weight matrix, and normalized. For this mapping, the type of normalization is selectable as either presynaptic or postsynaptic. 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.

WTA Module

The **WTA** module implements the dynamics of the winner-take-all inhibitory unit.

```
nslModule WTA ()
{
    public NslDinFloat1 motor();
    public NslDoutFloat0 s();
    private NslFloat0 s_mem();
}
```

The **simTrain** method executes the wta dynamics

```
public void simTrain()
{
    nslDiff(s_mem, u_s, - s_mem - threshold_s + nslSum(motor));
    s = nslRamp(s_mem);
}
```

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). The firing rate of this unit also reflects the number of currently active motor selection units.

13.4 Simulation and Results[3]

Simulation

Parameters

The set of parameters used to produce the results presented in this paper are described next. Table 13.1 to 13.9 show the complete list of parameters and the values used in the simulation.

Network Parameters	Value	Description
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)

Table 13.1
Network Parameters

Simulation Parameters	Value	Description
delta	0.01	Integration step

Table 13.2
Simulation Parameters

Weight Initialization	Value	Description
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).
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.

Table 13.3
Weight Initialization Parameters

Feature detector parameters	Value	Description
threshold_f	0.1	Threshold
u_feature	0.05	Time constant

Table 13.4
Feature Detector Parameters

Threshold Modulator	Value	Description
init_threshold_v	0.2	Initial threshold (threshold is determined by the threshold modulator).
u_threshold_v	4.0	Time constant of threshold modulator for voting units.

Table 13.5
Threshold Modulator Parameters

Voting Unit	Value	Description
u_voting	0.05	Time constant
gain	0.045	Injected noise to voting units.
change_probability	0.01	Determines how often the injected noise term changes value.

Table 13.6
Voting Unit Parameters

Motor Program Selecion Unit	Value	Description
u_motor	2.0	Time constant
gain	0.05	Injected noise to motor units
change_probability	0.01	Determines how often the injected noise term changes value.
threshold_m	0.035	Motor program unit threshold.

Table 13.7
Motor Program Selection Unit Parameters

Analysis Parameters	Value	Description
α	0.8	Used to compute average performance.

Table 13.8
Analysis Parameters

Simulation Parameters	Value	Description
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 13.9
Simulation Parameters

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 effect will be very little noise injected into the system. Therefore, all eligible columns will fire together, and not in different subsets.

Table 13.10
WTA parameters.

WTA (Inhibitory Unit)	Value	Description
u_s	0.5	Time constant of membrane potential.
threshold_s	0.1	Unit threshold.

Table 13.11
Learning parameters.

Learning Parameters	Value	Description
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).

Table 13.12
Protocol parameters.

Protocol Parameters	Value	Description
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.

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 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 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 13.13 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 13.14.

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 13.13
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 13.14
Input patterns and expected motor responses (more difficult case). Each of the patterns overlaps at least one other pattern.

Simulation Results

Once NSL has been compiled for the model, the simulation is started by loading the *startup* script, which loads in the standard set of parameters (CondLearn.nsl) together with graphics (CondGraphics.nsl):

```
nsl% source startup.nsl
```

The system parameters involve 100 epochs of 200 training cycles each:

```
nsl set system.epochSteps 100
nsl set system.trainDelta 1
nsl set system.trainEndTime 200
```

The *seed* is used to configure the random number generator (useful for forcing the same conditions for multiple experiments):

```
nsl set condLearnModel.condLearn.seed 10
```

The *pattern file* contains a list of input patterns and the corresponding desired outputs.

```
nsl set condLearnModel.trainFile.pName a1.dat
```

For example, the "a1.dat" train file looks as follows:

```
4
1
24
1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0
0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  1
0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  2
0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0  3
```

The first line specifies the number of train patterns in the file, "4" in this example. The second line specifies the number of elements in the output pattern, while the third line specifies the number of elements in the input pattern, "1" and "24" respectively. Finally, each additional line specifies the actual train pattern consisting of the input pattern followed by its corresponding desired output.

Once configuration is complete, begin execution as follows:

```
nsl% nsl train
```

In this configuration, at the end of each trial the system reports if it is unable to make a decision by the time the trigger stimulus is received:

```
No-pick no-go!
```

On the following line, the number of time steps (equivalent to train steps) required to obtain a decision is printed (in this case, 200 is the maximum number of time steps).

```
200
```

On the next line, the system prints the current trial (followed by a ":"), presented pattern number ("p"), expected motor output ("s"), selected motor output ("w"), indication of correctness (+/-), and a measure of total activity of the voting units.

```
0 : p0 s0 w0 + 0.022166
```

Next the system prints the participation vector, which indicates those columns that were active at the end of the trial.

```
voting_participation
1 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0
```

To perform an entire experiment, the simulation continues executing 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, 50 time steps should be sufficient).

```
32
50 : p3 s3 w3 + 0.123120
voting_participation
0 0 0 0 0 1 0 0 1 0 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 1 0 0

77
51 : p0 s0 w0 + 0.057379
voting_participation
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0

55
52 : p1 s1 w1 + 0.094080
voting_participation
0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 0 0 0
.
.
.
68
58 : p2 s2 w2 + 0.069137

nsl set voting_participation
0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 0 0 0 1 0
```

In figure 13.8 we show a sample graphical display as the system has already learned.

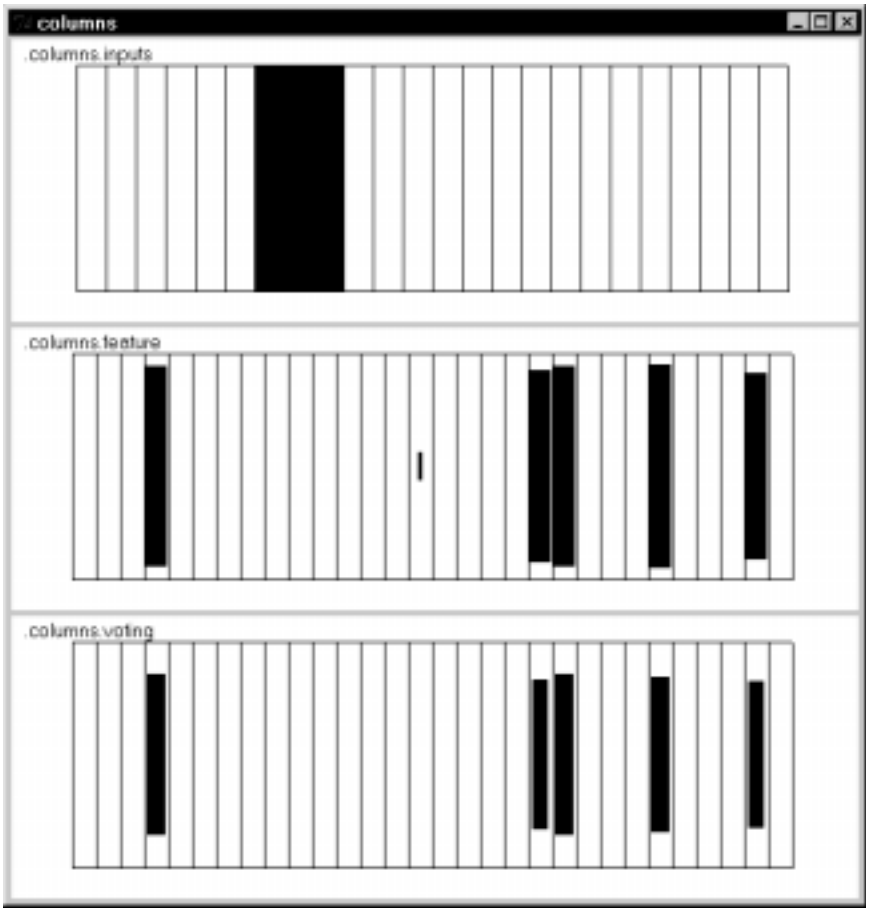

Figure 13.8
The top portion of the display, ".columns.input", represents the train pattern input, in this case corresponding to the third input pattern with corresponding desired output "2". The display in the middle represents the corresponding ".columns.feature" pattern while the bottom display represents the ".columns.voting" pattern. The more correspondence between the two bottom displays the better the learning.

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, 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.

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 to 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

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

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

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 an incorrect response and 1 corresponds to a correct response).

Figure 13.9 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.

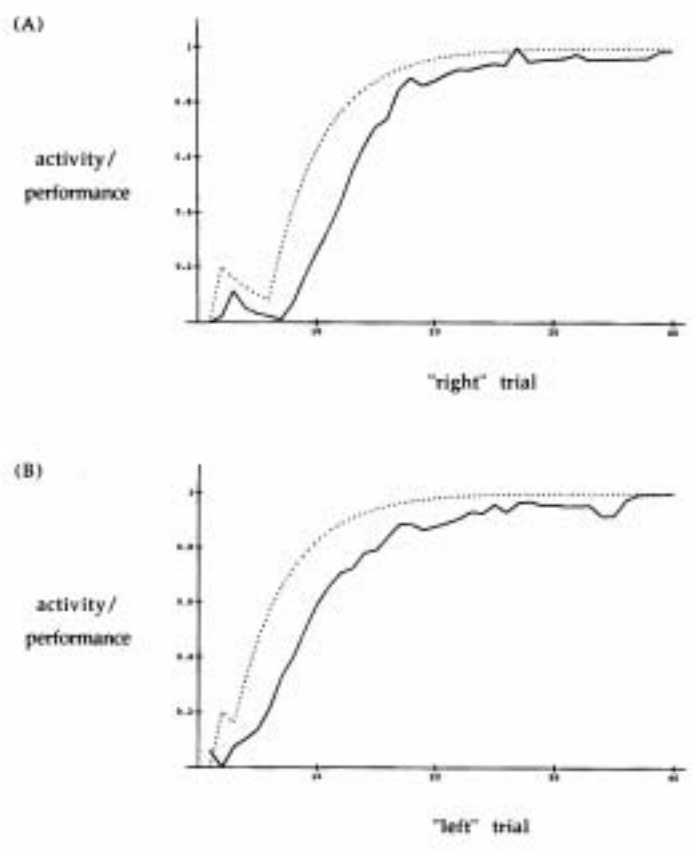

Figure 13.9
Overall activity (solid) curve and performance (dotted) curve plotted against trial for the (A) "Right" and (B) "Left" responses. As in the experimental curves, the performance curve begins to increase prior to the increase in the activity curve. Note that the trial axis represents only those trials for which the "Right" and "Left" responses are expected, respectively.

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 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 13.10, 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. (1991). 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.

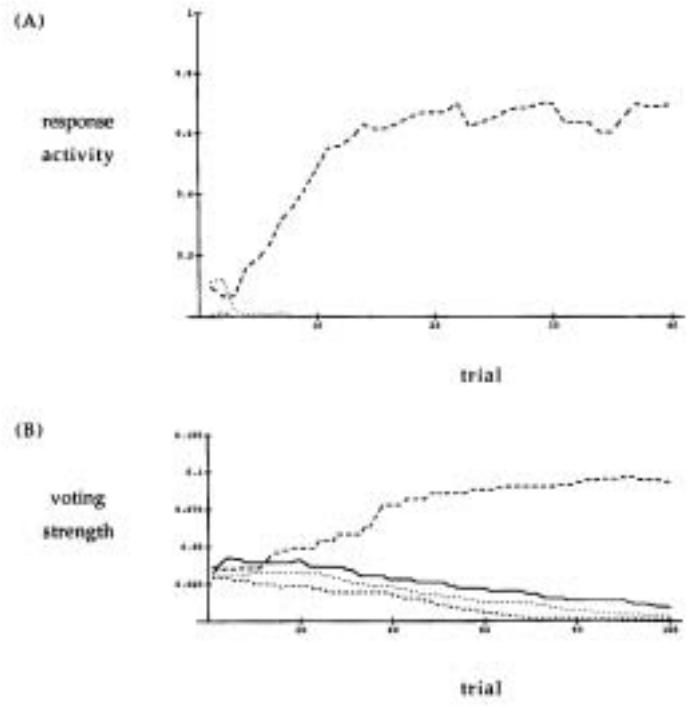

Figure 13.10
(A) Activity of a single voting unit plotted against trial number. The four curves represent responses to each of the four input stimuli (solid = no-go, long dash = left, dotted = right, and short dash = down). Note that trial number *N* corresponds to the Nth occurrence of each of the four 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.

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 13.10A 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 13.10B 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.

Changes in Protocol

In initially examining the protocol described by Mitz et al. (1991), 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 13.11 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.

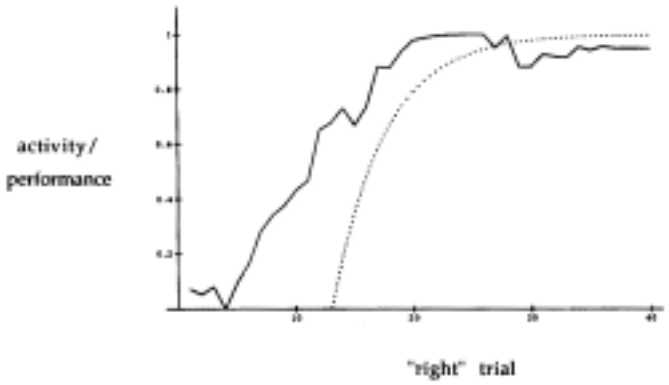

Figure 13.11
The behavioral responses of one experiment with a completely random sequence of stimulus presentations. Under these conditions, the task requires more trials of learning.

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 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 13.11 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 particular column initially responds to both the Rightward and Downward stimuli (figure 13.12 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 13.12B, 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.

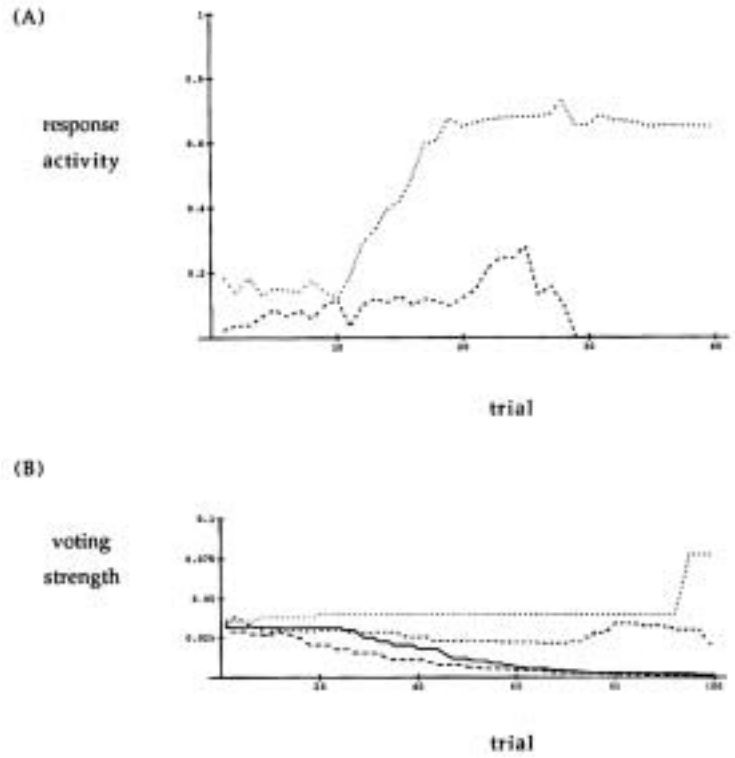

Figure 13.12
(A) Activity of the one voting unit that learned to perform the mapping plotted against trial number. Because this is the only unit that learns the mapping, it determines the curve of figure 13.10. (B) Evolution of the voting strength of the same unit. Only towards the end of the experiment (95th trial) does the unit discover the correct motor program selection unit.

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 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. Next we show 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. 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.

```
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 + + + + + + + + + + + + + + + + + + +
```

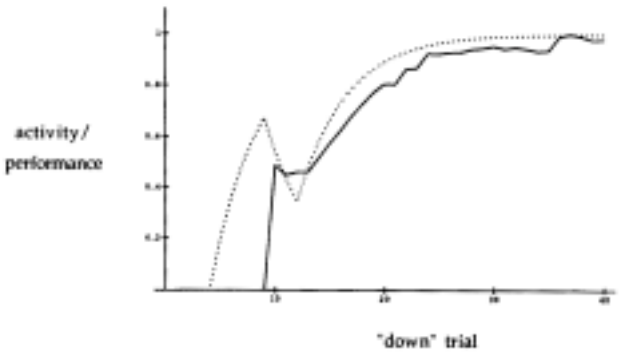

Figure 13.13.
Graph displays activity/performance curve for the reversal case ("Down" motor mapping). The solid (activity) curve corresponds to the overall activity in response to the stimulus that maps to the "Down" motor response, which switches between the 9th and 10th trials. The drastic change in 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.

The activity/performance curve for the Left response is shown in figure 13.13. 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 the point at which the reversal takes place. This jump happens because although the column 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 13.14 A demonstrate that the column maintains its mapping to the specific stimulus. Figure 13.14 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.

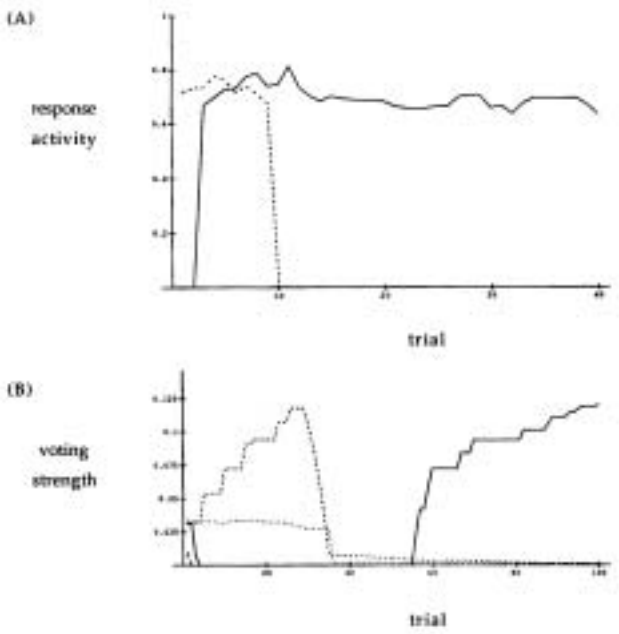

Figure 13.14
(A) A single unit's response to the various input stimuli over time. The solid curve represents the unit's response to the Nth occurrence of the stimulus that maps to the "No-Go" response. Likewise, the dashed curve represents the unit's response to the Nth occurrence of the stimulus that 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.

13.5 Summary

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. (1991) 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 (goal 2) were explicitly designed into the neural algorithm, the two features resulted from 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.

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 trans-

ient, the synaptic changes are more likely taking place in regions upstream from premotor cortex.

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.

Notes

1. 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 that 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.
2. A. Weitzenfeld developed the NSL3.0 version from the original NSL2.1 model implementation written by A.H. Fagg as well as contributed Section 13.3 to this chapter.
3. The Primate Visual-Motor Conditional Learning Model model was implemented and tested under NSLC.