To Appear in New Perspectives in the Control of the Reach to Grasp Movement (K.M.B. Bennett and U. Castiello Eds.)

CHAPTER 13

REINFORCEMENT LEARNING FOR ROBOTIC REACHING AND GRASPING

A. H. Fagg

Center for Neural Engineering, Computer Science Department, University of Southern California, Los Angeles, California 90089-0781 (ahfagg@mensa.usc.edu)

SUMMARY

A reinforcement learning approach is used to train a neural controller to perform a robotic reaching task. Unlike supervised learning techniques, where the teacher must provide the correct sequence of motor actions, only an evaluation of the robot's performance is provided. From this limited information, the robot must discover the appropriate motor programs that best satisfy the teacher's evaluation criterion. This type of learning approach is important because in a real-world environment, the teacher is generally not able to describe the motor program that performs the desired motor skill. This chapter utilizes the language of schema theory [1] *as a mechanism for describing functional decompositions of motor programs. A connection is made from schema descriptions to a neural-level implementation of the schemas. It is at this low level of processing that we define a reinforcement learning algorithm that acquires motor programs that satisfy the reinforcement policy defined by the teacher.*

INTRODUCTION

In a laboratory situation, a primate learns to perform the task designated by the experimenter through a reward/penalty or reinforcement-based paradigm. This reinforcement information, however, is extremely sparse relative to all of the things the monkey must do in order to obtain a reward. Even with the simplest tasks (e.g. reaching to grasp a handle), a monkey has many different motor acts that are available, from which he must select some sequence. When a reinforcement signal is provided, he must somehow infer the critical elements of his actions that caused him to receive the reward, so that these elements may be repeated the next time that the same situation arises. Despite this very limited amount of information, the monkey is often able to learn the desired task.

Within the robotics domain, we find a somewhat similar problem, in that it is typically difficult to specify a robust motor program. A very common technique is to specify in great

detail the trajectory through space that the manipulator is to take in performing a task. This has worked fairly well in structured environments, but as the environment becomes more uncertain, it becomes more difficult for the programmer to anticipate all possible situations, let alone the appropriate actions that must be taken. We would therefore prefer to specify programs at a higher level: one in which it is more natural for a programmer or teacher to communicate. Our approach draws inspiration from learning in monkey, using reinforcement (or reward-based) information to specify the desired behavior of the robot, as opposed to specifying the motor program that produces that behavior (note that *supervised learning* techniques are one way of implementing this latter case).

Learning within a reinforcement-based paradigm, however, presents several key difficulties, which have been explored by a number of authors, including Barto, Bradtke, Dayan, Sutton, Watkins, Werbos, and Williams [2-6]. These are:

A) The reinforcement signal is only a scalar measure of the performance and does not provide explicit corrective information.

B) The reinforcement signal is not necessarily continuous in time (i.e., it may only be available at very discrete events).

C) The reinforcement signal can be temporally delayed relative to the *critical actions* taken by the neural controller. *Critical actions* are actions in the sequence generated by the controller that determines the final result in the environment (and indirectly determine the reinforcement).

Some elements of this reinforcement-based approach have also been explored in Fagg and Arbib [7], which presents a model of the work of Mitz, Godshalk, and Wise [8]. Their work examined the changes in behavior and in neural responses in the premotor cortex as a monkey learned an association task. In these experiments, monkeys were first taught to associate a set of four distinct visual patterns with a particular movement of a joystick. For example, when the monkey is shown the character *A*, then he is expected to move the joystick to the right. If the monkey responds correctly to the stimulus, then he is rewarded with a squirt of juice. Once the monkey has learned the overall task, sets of novel stimuli are presented. The monkey is to infer the appropriate motor response to each stimulus based upon the reward information that he is given.

The key results of the model were :

- The model produced a similar pattern of behavior as was observed in the experiments with the monkey.
- The modeled neural units behaved similarly to the premotor cortex cells observed by Mitz et al.
- The model represented the visual-motor transformation in a distributed manner and

updated this transformation based only upon the reinforcement signal received from the teacher. By *distributed representation,* it is meant that a particular transformation did not depend exclusively upon a single computational unit, but rather on the co-activation of a set of units.

In this work, these ideas are extended in several dimensions:

- Sensing and generating actions now become continuous processes, rather than a one-step sensor-to-motor transformation.
- The teaching signals are no longer in one-to-one correspondence with the actions taken by the learning system. In general, a whole sequence of actions is taken before reinforcement information is available. In addition, it is possible that this signal is delayed relative to the critical actions taken by the system. These problems are approached by modifying the learning algorithm such that the reinforcement signal is propagated backwards through time in an efficient and biologically plausible manner.
- We begin to approach the issue of different neural regions being involved in a computation and how their relative functions might work together to perform a task. It is of special interest to understand how learning may occur at different levels within a control hierarchy. For example, when a neural system is learning a new task, not only must it decide what must be learned, but also at what level the new information needs to be encoded. In some cases, the low-level components of the controller for the new task are already in place, and it is only necessary for the higher-level to make adjustments to bind them together in a unique way.

Schema Theory to Neural Networks

Schema theory [1] provides a language for describing functional decompositions of sensory and motor processes. An individual *schema* is a parameterizable description of a computational element that may actually be implemented as a network of *sub-schemas*. Traditionally, the lowest-level schemas are implemented as either C processes or as encapsulated neural networks. A *schema instance* is a parameterized copy of a *schema* that performs the specified computation based upon the schema description and the provided parameters. The theory allows for the simultaneous existence of multiple *schema instances*, each with their own set of parameters.

From a biological stand-point, however, schema theory does not provide a sufficient language for mapping between schemas and neurons. Although we allow a schema to be implemented as a neural network and then connect it into a network of other schemas, this is only done at a functional level. What is missing is a bridge from the functional level of analysis to an implementational one. At such a level, we would like to explicitly address the issues of:

- The distributed representation of schemas across sets of neurons, and potentially over

multiple layers.

- The participation of a single neuron within one or more schemas. These schemas may, in fact, be functionally distinct from one another, and the task that the neuron performs for each case may also significantly differ.
- Neural representation of information and the operators that act on these representations. Schemas tend to exchange state information as sets of real-valued numbers and/or symbols. However, on the neural side, we have (somewhat ill-defined) notions of firing rates, spatial codes, and cosine tuning functions.

In order to bridge the gap from *schema descriptions* to the *implementation of schemas* using neural hardware, we introduce the concept of a *μ-schema*. A *μ-schema* is a simple processing unit that is still at a level higher than that of a neuron. A *schema* is implemented by recruiting a collection of *μ-schemas*. Even though different *schemas* can take on radically different computational structures, all *μ-schemas* utilize a fixed computational structure. Thus, the different computational structures of two *schemas* are achieved by recruiting different (but potentially overlapping) sets of *μ-schemas*.

In the remainder of the chapter, we first present the task to be learned: reaching towards a specified target from different points in the workspace. A global (schema-level) view of the neural controller is then presented, followed by a description of the neural implementation of the model. We then present the learning algorithm that is used to acquire schemas that perform the desired task. Finally, through a set of simulation results, the behavior of the model is illustrated.

TASK TO BE LEARNED

In this chapter, we will illustrate the design and behavior of the neural system described in the next section with a simple reaching task. The learning system controls an X-Y robot (a robot with two prismatic joints) and is to learn how to reach towards a specified target location. The inputs to the system are a teacher-provided command signal and goal location, as well as a feedback signal that informs the system as to the current location of the arm. The neural controller specifies outputs in the form of incremental changes in position of the arm. Reinforcement learning techniques have been applied to a similar problem by Barto et al. [9, 10], except in their case the target position was always fixed.

The robot arm is located in a closed workspace. The teacher provides the system with several different types of reinforcement information, which are summed to create a global reinforcement signal (this measure is scalar and continuous). First of all, the system is positively rewarded when the endpoint of the arm reaches the target location. Secondly, if the endpoint of the arm reaches the edge of the workspace, it is prevented from further movement and it is given negative reinforcement. Finally, the system also receives a small amount of

positive (negative) reinforcement if the movement in the last time-step was towards (away from) the target location. One important question to be examined is the degree to which this third type of reinforcement is necessary for the system to learn the task within a reasonable amount of time.

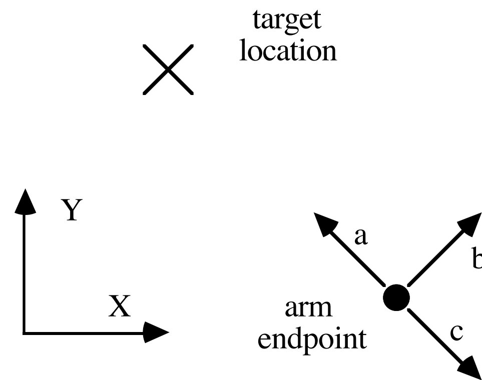

Figure 1. Several possible actions given a particular situation: a) move towards the target, receiving a significant amount of positive reinforcement; b) move perpendicular to the correct direction, receiving no reinforcement; and c) move directly away from the target, and therefore receive negative reinforcement (of the same magnitude as a*).*

This two degree-of-freedom manipulator provides a simple example through which to illustrate the neural architecture, but still presents interesting challenges. One primary difficulty in learning is that the controller must output both the correct x-dimension increment and y-dimension increment in order to move towards the goal and receive a reasonable amount of positive reinforcement for doing so (Fig. 1, action *a*). If the increment for only one dimension is correct (action *b*), but the other is incorrect (e.g. opposite in sign), then the system could receive either negative reinforcement or none at all. In a case such as this, it is impossible to determine which of the two control outputs was correct. Therefore, the system must rely upon multiple samples with different output actions to infer what the correct action is given a particular situation. Note that in this reinforcement scheme, the feedback resulting from the production of an action that takes the arm directly away from the target (action *c*) provides just as much information as does a movement towards the target.

MODEL DESIGN

The neural model described in this section can be viewed at several levels of abstraction. We will first look at the overall organization of the model, and then look closer at the details of the implementation.

Global Network Architecture

The global view of the network design is depicted in Fig. 2. The network consists of two main processing layers and several input/output layers. The *Command Vector* is the input into the network that calls up specific motor programs (high-level schemas). In other words, this defines the current task that the system is to perform. One such task (the one upon which we are concentrating in this chapter) is reaching towards a target. Another task might be to reach towards a specific location (regardless of the target input), or to reach towards a location opposite the target. For this example, however, it will be assumed that this input is fixed.

The *Planning Layer* is the neural structure that implements the *reach-toward-target* schema

(which has been selected by the *Command Vector*). The *Target Location* vectors are a neural representation of the X-Y position of the target object. The *reach-toward-target* schema in the *Planning Layer* makes use of this position information to select the schema within the *Execution Layer* that is responsible for moving the arm to the specified target location. This *reach-toward-a-specific-target* schema utilizes the current state of the arm (input from the *Arm Location* vectors) to generate movement commands for each of the two degrees of freedom. These output commands are sent to the *dX* and *dY* layers, where they are executed by the manipulator.

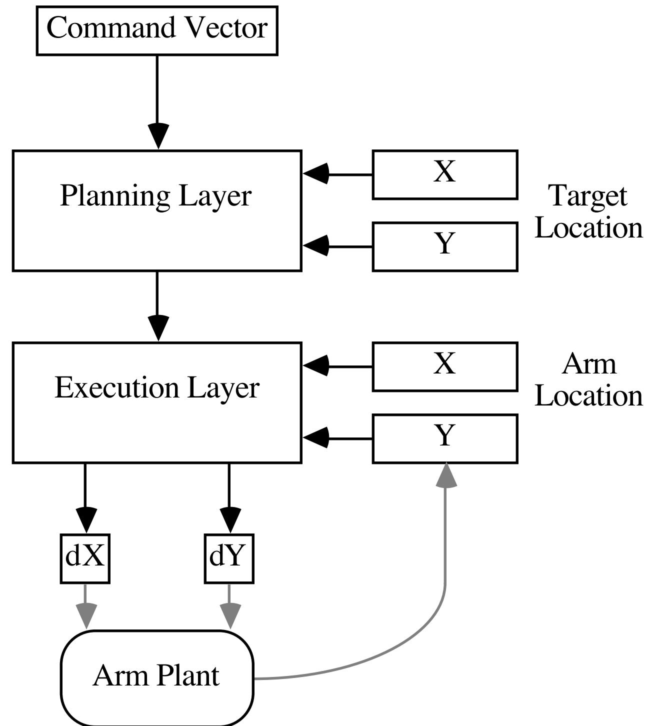

Figure 2. Overall view of the reach control network.

It is important to keep in mind that the global architecture presented in this section is specific to the particular reaching task. It is our intention that the neural-level implementation of the layers (described in the following sections) be generic in the sense that given other control problems, the same implementation would be useful, even if the global architecture has changed in some way (e.g. the addition of more processing layers).

Input / Output Coding

Each *Target Location* and *Arm Location* vector is a linear array of neurons that code a continuous value using a spatial code. In this case, a *Gaussian distribution* is used, where the location of the mean of the distribution is determined by the value being coded. The variance of the distribution is adjusted such that two values must be relatively close to one another to have significantly overlapping representations. The coding scheme is such that we are also able to represent a continuous range of values while utilizing only a finite number of neural units. Fig. 3 shows an example of Gaussian coding for three different values.

The *dX* and *dY* layers represent the output of the control network. Each layer consists of a linear array of units that spatially code one output variable (the increment of robot position along the X- or Y-dimension). A value is read out from a linear array of units by computing the *center of mass* of the activity levels:

$$x = \frac{\sum_{i=0}^{N-1} i * a_i}{\sum_{i=0}^{N-1} a_i}$$

where:

x is the decoded value.

i is the linear position along the array.

N is the length of the array.

a_i is the activity level of the ith unit.

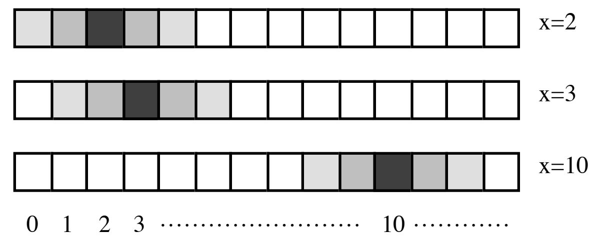

Figure 3. Example of Gaussian spatial coding in a linear array of neurons. Neuron firing rate is indicated by shading (white = not firing, dark = maximum firing rate). The values coded by the three linear arrays are 2, 3, and 10, respectively.

Processing Layer Implementation

The *Planning* and *Execution* layers of the global architecture (Fig. 2) consist of collections of *μ-schemas*. Physically, the *μ-schemas* are arranged into a two-dimensional grid. We first define the primary computational concepts that make up a *μ-schema* and then show how these concepts are implemented within neural hardware.

A *μ-schema* produces an output when it detects an incoming sensory feature. However, the activation (selection) of this mapping is constrained by both global and local inputs. In order for a μ-schema to become active, it must first be *primed* by some external input. When a higher-level process primes a μ-schema, it effectively grants its permission for the μ-schema to participate in a computation. For example, a high-level *grab object* process will need to recruit sub-schemas to execute the reach and grasp elements of the task. This is implemented by priming the μ-schemas that make up the two sub-schemas. In turn, these sub-schemas may be further broken down into more specific sub-schemas. Also, these two schemas may establish lines of communication for the purposes of coordinating their execution.

Due to the inherent simplicity of a μ-schema, implementing a single schema requires the activation of an entire set of μ-schemas. It is therefore necessary on an implementational level to provide a mechanism that ensures the co-activation of this set. On the other hand, some μ-schemas conflict with others by producing conflicting commands to a lower level or to an actuator. The co-activation of these μ-schemas is therefore not desirable.

These constraints are implemented through interactions between primed μ-schemas within a single layer. By allowing such an interaction to take place within each layer, the problem of deciding which processes are appropriate for execution is distributed throughout the network. As a result, the decision as to which sub-schema is appropriate for a given situation is left to the layer that has the contextual and sensory information necessary to make such a decision. In this work, these interactions are implemented as inhibitory and excitatory connections between the primed μ-schemas.

One simple way to implement μ-schema interaction is by connecting μ-schemas through a

Mexican-hat type operator. This connection scheme provides excitatory connections to physically near neighbor μ-schemas, and inhibitory connections to a ring of neighbors just outside the circle defined by the excitatory connections. Computationally, this scheme implements a contrast-enhancement operation that forces only a few neurons to be active in a layer at any one time.

Given this context, it is possible to more precisely interpret the concepts of *schema* and *schema instance*. A schema instance is a particular pattern of activity within a layer. A schema can be considered as the set of constraints imposed by a particular class of priming input and the interaction between the set of μ-schemas.

Figure 4. The structure of the column. Arrowheads indicate excitatory connections, inhibitory connections are indicated by small circles. Gray arrowheads represent modifiable synapses.

Neural Implementation of the μ-schema

The *μ-schema* is implemented using a tightly-coupled collection of neurons. These collections are referred to in this work as columns of neurons. Fig. 4 depicts the neural implementation of the column. We first define the behavior of the generic neuron model, and then discuss the detailed implementation of the column.

Leaky Integrator Model of the Neuron. The neurons in the model are implemented using the leaky-integrator model [1] as a basis. Each neuron is represented by a membrane potential and a firing frequency (Fig. 5). The dynamics of the generalized neuron are defined by the following equations:

$$\tau \frac{d\,mem}{dt} = -mem - threshold + \sum inputs$$

$$firing = f(mem)$$

where :

τ is the time constant of integration.
mem is the membrane potential of the neuron.
threshold is the neuron's internal threshold.
inputs is the set of external inputs into the neuron.
firing is the firing rate of the neuron.
$f()$ is a function (typically non-linear).

The Priming Unit. Inputs from other layers (primarily higher-level layers) prime the column. The inputs are summed to determine changes in the *priming unit's* membrane potential:

$$\tau_P \frac{d\,pr_i}{dt} = -pr_i - thresh_P + \sum_j w_{ji}^P * out_j$$

where:

pr_i is the membrane potential of the priming unit of column i.

$thresh_p$ is the threshold parameter for the priming units.

w_{ji}^P is the strength of the connection from column j (another layer) to priming unit i.

out_j is the output activity of column j of a preceding layer.

The firing rate of the priming unit is then computed by:

$$prime_i = NSLsat(pr_i)$$

where:

$prime_i$ = the firing rate of priming unit i.

$$NSLsat(x) = \begin{cases} 0 & x < 0 \\ x & 0 \le x \le 1 \\ 1 & 1 < x \end{cases}$$

The Activity Unit. The activity of a column is determined by the firing rate of the *activity unit*. Excitatory interactions between two columns are implemented as a positive connection between the corresponding *activity units*. Inhibitory interactions are implemented through the use of an *inhibitory unit*. When one column inhibits the activity of another column, it excites the *inhibitory unit* of the target column, which in turn inhibits the *activity unit*. The dynamics of these two units are as follows :

$$\tau_A \frac{d\,ac_i}{dt} = -ac_i - thresh_A + pr_i - inhibit_i + noise_i + \sum_{j \ne i} w_{ji}^A * act_j$$

$$\tau_I \frac{d\,inh_i}{dt} = -inh_i - thresh_I + \sum_{j \ne i} w_{ji}^I * act_j$$

$$act_i = NSLsat(ac_i)$$

$$inhibit_i = NSLsat(inh_i)$$

where:

w_{ji}^A and w_{ji}^I are connections from other columns (specifically their *activity* and *inhibitory units*) within the same layer. In this

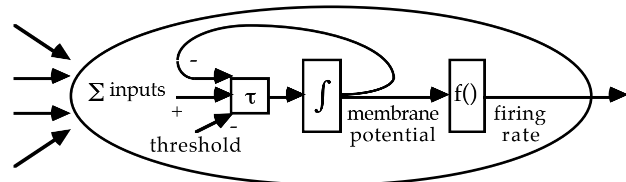

Figure 5. The leaky integrator model of the neuron. The membrane potential of the neuron is affected by the current set of inputs, the neuron's threshold, and the current state of the neuron. The firing rate is a non-linear function of the membrane potential.

implementation, connections are made only to adjacent columns.

$noise_i$ is a noise signal that is injected into the membrane potential of the activity unit. This signal changes slowly relative to the time constant of the processing units; the distribution is uniform over a small range: $[-\alpha, \alpha]$.

The noise term plays an important role in the behavior of the system. In the early stages of learning, this noise helps to drive the search process when the controller is unsure of the correct action to be taken. As a result, the control space is more efficiently explored. When an *activity unit* is firing, the column (μ-schema) is participating in the current computation.

The Sensory Unit. The *sensory unit* detects sensory events from the environment or from the state of the robot. For example, the input into the *Execution layer* of the network consists of two arrays of units that specify the current end-point position of the arm. Thus, a *sensory unit* at this level can detect events such as the arm moving into:

A) A particular range of the Y dimension.

B) A specific region of the workspace (requires inputs from both the X and Y dimensions).

C) Multiple regions of the workspace.

A similar type of incidence coding scheme has been used by Mel [11], except that each sensory unit receives exactly one input from each dimension (e.g. X and Y). This results in only units of class B, which requires many more units to cover the entire space.

The sensory units in the *Planning* layer receive input from the *Target Object* vectors.

The dynamics of the *sensory unit* are similar to those of the *priming unit*:

$$\tau_S \frac{d\,sen_i}{dt} = -sen_i - thresh_S + \sum_{j \neq i} w_{ji}^S * input_j$$

$$sensory_i = NSLsat(sen_i)$$

where:

$input_j$ is an element of the sensory input vector for this particular layer.

The Output Unit. The firing rate of the *output unit* is the *sensory unit* firing rate gated by the *activity unit*:

$$output_i = activity_i * sensory_i$$

The activity level of the *output unit* represents the output of the column, which then connects to other layers in the network. The output from one layer provides input to *priming units* of the destination layer. Thus, schemas at one level prime sub-schemas implemented at lower layers of the network.

LEARNING SYSTEM

Given the overall structure of the network, the task of the learning system is to tune the

connection strengths between the various sets of units to develop a set of μ-schemas that accomplish the task specified indirectly by the reinforcement signal. This tuning must be done based upon the experience of the system interacting with its environment and the reinforcement signal that it receives from the teacher. As discussed in the introduction, this problem is difficult because:

A) The reinforcement signal is a scalar measure of the performance and does not provide explicit corrective information.
B) The reinforcement signal is not necessarily a continual signal.
C) The reinforcement signal can be delayed temporally relative to the critical actions that were taken by the neural controller.

Determining which connections should be updated is referred to as the *credit assignment problem*. Given that the teacher provides some instantaneous reinforcement signal, the learning system must identify which computational elements (columns) were responsible for generating the actions that ultimately led to the reinforcement (*structural credit assignment*), and at what time did these elements make the critical decisions (*temporal credit assignment*).

In the columnar structure, two sites are subject to updates in connection strength: connections from the *output units* of one layer to the *priming units* of another layer, and the connections from the sensors to the *sensory units* (gray arrowheads of Fig. 4). By tuning the set of connections from the output units of one layer to the priming units of another, the learning system adjusts the set of μ-schemas that are to be primed at the lower level, and thus determines the set of μ-schemas that make up the sub-schemas. At this level, the learning scheme effectively implements the following rule:

A) If the lower-level μ-schema is active (and thus is participating in the computation) during periods of time when the control system *tends* to receive positive reinforcement, then increase the connection strength from the higher-level column (which is active) to this column.
B) If this μ-schema tends to participate during times when the system is receiving negative reinforcement, then reduce the connection strength.

The connections from the sensors to the sensory units are adjusted in a similar manner. This adjustment implements the rule:

A) If a μ-schema is active and it is producing a non-zero output during a period of time in which the system *tends* to receive positive reinforcement, then increase the connection strength from those sensor inputs that are currently firing.
B) If the system *tends* to receive negative reinforcement, then it is possible that the μ-schema is attending to the incorrect sensory feature; therefore, reduce the connection strength to those sensor inputs that are currently firing.

In both of these loosely-defined learning rules, the terms *tends to receive positive reinforcement* and *tends to receive negative reinforcement* are very important. Even though the system finds itself in several very similar situations, the control system may produce different control actions (due to the noise injected into the *activity units*), or the teacher may provide apparently inconsistent reinforcement information (due to the inexactness or more qualitative nature of the reinforcement signal). As a result, the learning system must not make large adjustments based on the instantaneous reinforcement signal, but rather must take into account many experiences in constructing an effective control program.

The challenge, then, is to consolidate these experiences in an efficient manner - both in time and in storage space. The algorithm below presents one approach to solving this problem.

Eligibility as a Temporal Measure of Credit Assignment

The eligibility of a weight (connection between two units) measures the participation of the connection within the computation that is currently taking place. The *instantaneous eligibility* is defined as the coincidence between the pre- and post-synaptic cell activities. In our case, this product is also modulated by the strength of the connection between the two cells (this definition of eligibility was inspired by the work of Klopf [12], and Barto, Sutton, and Anderson [2]). Thus, the *instantaneous eligibility* is :

$$\tilde{e}_{ij} = a_i * a_j * w_{ij}$$

where:

$\tilde{e}_{ij}$ is the *instantaneous eligibility* between unit i and unit j.

a_i and a_j are the activity levels of the pre- and post-synaptic columns, respectively.

w_{ij} is the weight from unit i to unit j.

An exponentially-decaying memory of the *eligibility* can be implemented by applying a low-pass filter to the time series of instantaneous eligibilities :

$$\tau_e \frac{de_{ij}}{dt} = -e_{ij} + \tilde{e}_{ij}$$

where:

τ_e is the time constant of integration, or the decay of the memory.

e_{ij} is the eligibility of the connection.

When a reinforcement signal (R) is provided by a teacher, the eligibility of a connection is used to update the weight. More specifically:

$$\Delta w_{ij}(t) = \alpha * R(t) * e_{ij}(t)$$

where

$\Delta w_{ij}(t)$ is the change in weight.

α is the learning rate.

This update equation says that if a connection has recently been participating in a

computation, then make a small incremental change to the connection strength. The sign of this incremental change is determined by the sign of the instantaneous reinforcement signal, $R(t)$. The magnitude of this increment is determined by the magnitude of the reinforcement signal, and by the degree of participation of the connection, $e_{ij}(t)$.

It is important that the rate of learning, α, is adjusted appropriately. When set at a value that is too small, the learning time can be longer than practical. If set too large, the noise component of the weight increments (due to the noise injected into the controller or to noise in the reinforcement signal) can be amplified above the level of the meaningful information.

Weight Normalization

Once the change in weights is computed, the actual connection strength is updated according to:

$$w_{ij}(t+1) = Normalize\left(w_{ij}(t) + \Delta w_{ij}(t)\right)$$

Biologically, normalization comes out of the limited resources that a neuron has to establish connections to other neurons. Computationally, normalization performs two important functions:

A) Individual weights are bounded within a finite range, thus alleviating some computational difficulties.

B) Normalization implements a form of competition between the individual weights. This weight competition can take one of two forms: presynaptic or postsynaptic.

For presynaptic normalization, the function $Normalize(\ )$ maintains the conditions $\sum_i w_{ij} = 1$ and $0 \le w_{ij} \le 1$ (note that w_{ij} is defined as the connection strength from unit **i** to unit **j**). In other words, the total output from the presynaptic unit is a constant value; as the weights change, it is only the distribution of the output that changes. This type of normalization is used for the connections from the output unit of one layer to the priming unit of another layer. Within this context, normalization can be interpreted as an active column (at the presynaptic side) searching for the appropriate set of lower-level columns to which to distribute its priming support.

Fig. 6a shows the effect of positive reinforcement on the connections from one column to a set of columns. Initially, the connections to columns *a* and *b* have significant strengths. However, columns *b* and *c* are the ones that are currently active. Thus, when positive reinforcement is received, the strength of these two connections increases. Due to normalization, the connection strengths to columns *a* and *d* are reduced. One way to interpret this behavior is that the higher-level column is becoming more *sure* of the correct set of sub-columns that it should prime so as to receive positive reinforcement in the future. Thus, it becomes more committed towards these columns through the increase of the weights.

When negative reinforcement is received, the opposite situation occurs (Fig. 6b). The

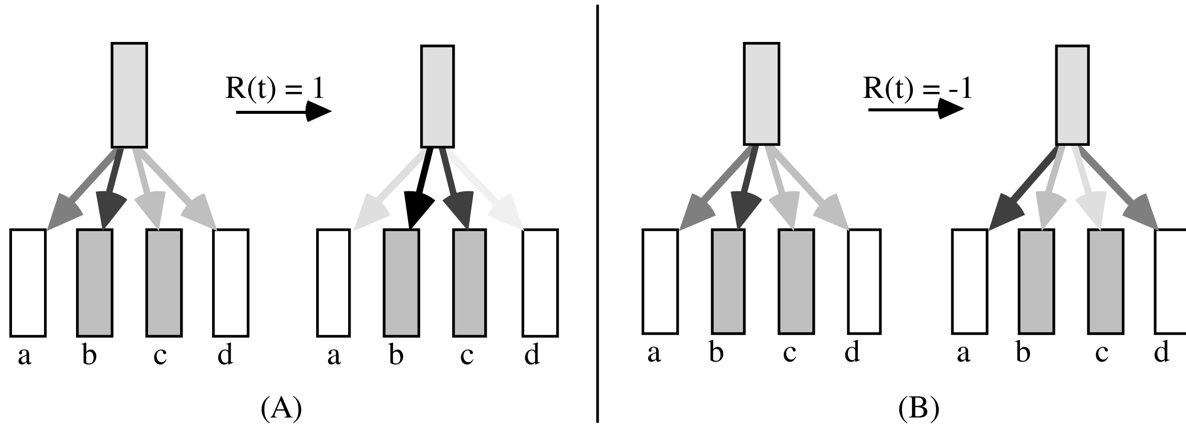

Figure 6. Demonstration of the effect of positive (A) and negative (B) reinforcement with presynaptic normalization. The upper box represents the higher-level column, whose output unit connects to the priming units of the lower-level systems (lower boxes). Column activity is represented by degree of shading. Connection strengths are also represented with different shading levels (light = small weight, dark = large weight).

weights leading to the active columns are decreased slightly. The remaining weights are then increased as a result of normalization. In this case, the higher-level column is not so sure about whether it should be priming the currently active columns, and hence the decrease in connection strength. By reallocating the connection strength removed from these active columns, the higher-level column gives other columns more of a chance to become active the next time the same situation occurs, thus driving the search process for more appropriate sub-columns.

When postsynaptic normalization is used, the weights are normalized across the opposite dimension of what is done in presynaptic normalization. So the function *Normalize*( ) maintains the condition $\sum_j w_{ij} = 1$. It is this type of normalization that is used for the projection from sensor inputs to *sensory units*. From the *sensory unit* point of view, the unit is attempting to select the sensor elements that tend to yield positive reinforcement when used as triggering signals.

MODEL BEHAVIOR

The network used for the experiments described here consisted of a 7 x 7 grid of columns for the *Planning* layer, and a 16 x 16 grid of columns for the *Execution* layer. The *dX* and *dY* layers each consisted of 3 columns. The inputs layers representing positional information consisted of arrays of 15 units each. On network creation, the connections and their strengths are randomly generated, yielding a network that is not committed to any particular set of schemas.

A learning trial begins by selecting one of two opposite corners as a starting position for the arm endpoint, with the target located roughly in the center of the workspace. The system is

then allowed to drive the arm until one of two events occurs: the arm arrives at the target position, or the arm reaches the side of the workspace. At this point, the final reinforcement is given and the arm position is again reset to a starting location. By using more than one starting location, the system is forced to explore a large region of the state space. However, using only a small number of starting locations allows for efficient experimentation and a more controlled analysis.

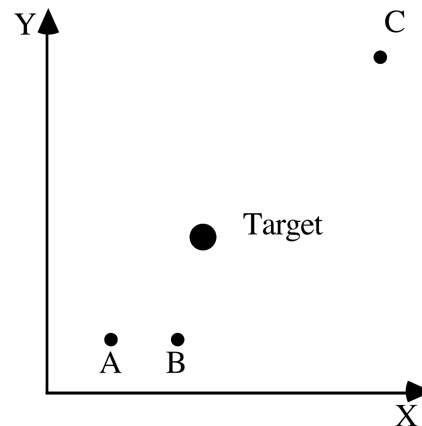

Figure 7. The workspace layout for three different arm positions.

When a target position is specified by the teacher, a small subset of *Execution* layer columns becomes active (about 25%). For a given arm position, the *sensory units* of some of these active columns fire in response to the arm position input. These columns then prime the *dX* and *dY* layers. Because the connections are randomly generated, the priming of each of the six columns in the *dX* and *dY* layers tends to be at about the same level as the others. Moving the arm position to different locations in the workspace yields very little change in the priming levels.

Figs. 7-11 demonstrate the responses of the *Execution*, *dY*, and *dX* layers before learning has occurred. Fig. 8 shows the response of the *output units* in the *Execution* layer when the arm is in position A (as defined in Fig. 7), and Fig. 9 shows the response of the *dX* and *dY* layers. The responses to the other two arm positions (B and C) are depicted in Figs. 10 and 11, respectively.

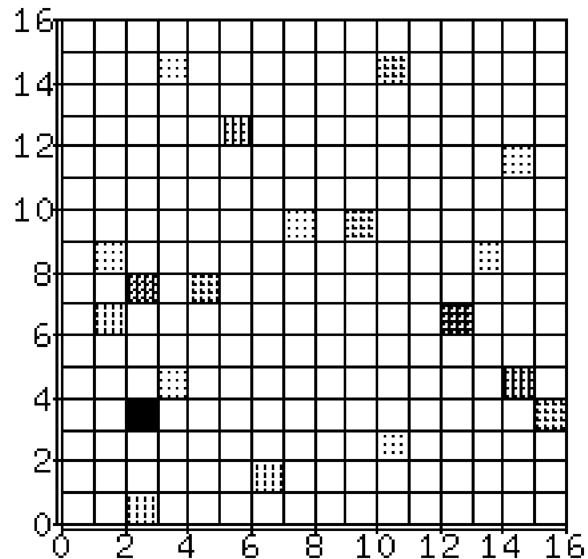

Figure 8. Pre-learning state of the output units of the Execution layer when the arm is located at position A. Each small box represents the state of one unit, with darker boxes indicating high firing rate. The high firing rate in the activity units of the execution layer indicates those columns that are currently participating in the motor program for this particular target location. It is these active units that prime the dX and dY layers.

Note that Figs. 8 and 10 show only some difference in the output activity of the Execution layer. This is due to the fact that the two arm positions are very near each other. There are, however, several *output units* that change significantly in their activity levels. After learning

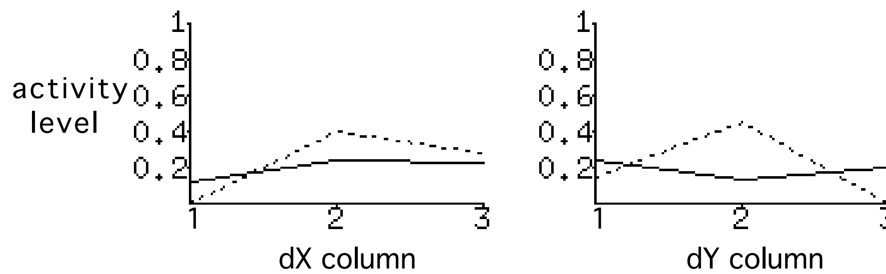

Figure 9. Firing rate plotted against linear position for the priming units (solid) and output units (dotted) for the dX and dY layers (same conditions as in Fig. 8). Each layer has three units. An activity peak centered near the second unit implies a command of zero magnitude; biased towards the first unit implies movement in the negative direction. Note the relatively uniform priming input in both cases. However, once this signal is contrast enhanced (output unit activity), it is possible to see slight bias in one direction. In this case, the system is commanding a slight positive movement for the X direction, and a slight negative movement in the Y direction.

takes place, it is these units that will encode the essential differences in motor output between these two positions. Those units that are active in both cases will encode the commonalities of the two motor commands.

The differences between Figs. 8 and 11 are much more significant. This is due to the physical separation of the two locations, which implies that the arm position input patterns for the two cases (B and C) do not overlap and therefore do not activate many common *sensory units*. This will make it much easier for the system to learn radically different motor outputs for

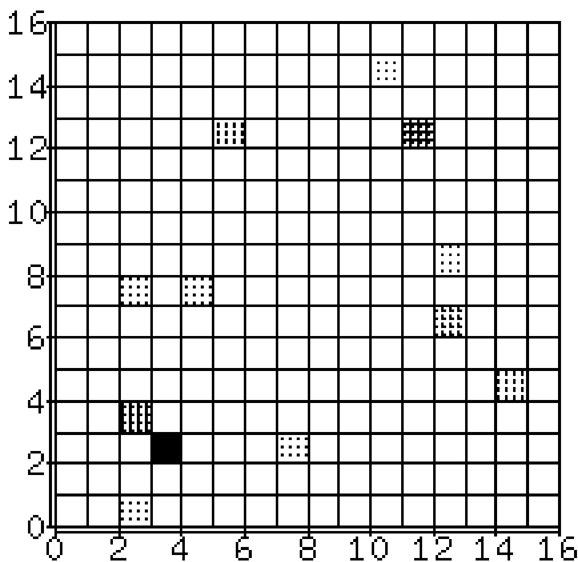

Figure 10. Pre-learning state for the output units of the Execution layer for arm position B. Note similar activity pattern as that in Fig. 8.

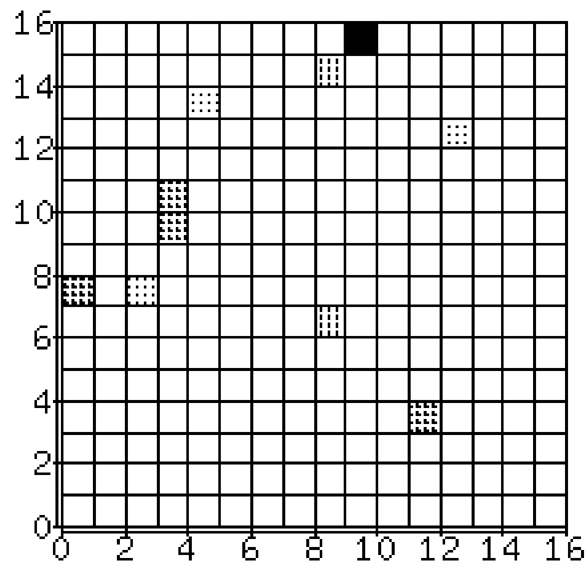

Figure 11. Pre-learning state for the output units of the Execution layer for arm position C. Note significant differences in the activity pattern as compared with Figs. 8 and 10. These differences are due to the radically different arm positions.

these two conditions.

In all three cases, however, the priming signals for the *dX* and *dY* layers are all at approximately the same level (the response to arm position A is depicted in Fig. 9), with only very small biases in one direction or another. Despite this relative non-commitment on the part of the *Execution* layer, the contrast enhancement that occurs between the activity units forces a choice of movement in one of the directions (this is especially evident in the firing rates of the output units). As learning proceeds, a particular schema instance will begin to prime very specific columns in these layers. As a result, the layer will rely less on the contrast enhancement as a way of selecting a specific output.

The small amount of variation between the *dX* and *dY priming units* confirms that the network is not significantly biased to produce any particular action before learning has occurred. Therefore, the choice of action to output is primarily driven by the noise that is injected into the *activity units* of the *dX* and *dY* layers. The result of this random set of actions is that the endpoint of the arm tends to wander around the workspace through a random trajectory (e.g. Fig. 12a) .

As learning progresses, the controller begins to bias the noise signal in regions of the workspace that have been visited several times. This biasing must be done slowly so as to allow the system to explore several different possibilities before committing to one particular movement direction. It is at the point of full commitment that the bias provided by the controller reaches a level above that of the noise.

Examining the behavior of the system relative to a single starting position over many trials, one can observe the general strategy taken by the system. The system first begins by exploring a small local area around the starting point before wandering off to another region of the workspace (Fig. 12a). Over several trials, however, the same local area is explored every time (Fig. 12b). This common experience allows the system to decide upon the best action for this region of the workspace (Fig. 12c). In subsequent trials, the system executes this action, taking the arm to a different location of the workspace (closer to the target), where the controller is now relatively inexperienced. It then proceeds to repeat this process, overall taking small steps towards the target location (Fig. 12d). Once the entire path is discovered, repeating it several times solidifies the set of actions in memory (Figs. 12e,f).

For starting location 12,12, the control system caused the robot to collide with the edge of the workspace only the first three trials. By the 12th trial it had learned how to perform the task perfectly. For starting location (2,2), the robot collided with the side of the workspace a total of 7 times, and learned to navigate to the target by the 16th trial.

After learning has completed, the *Execution* layer is much more committed towards particular output commands. This is evident in the distribution of the firing rate of the priming

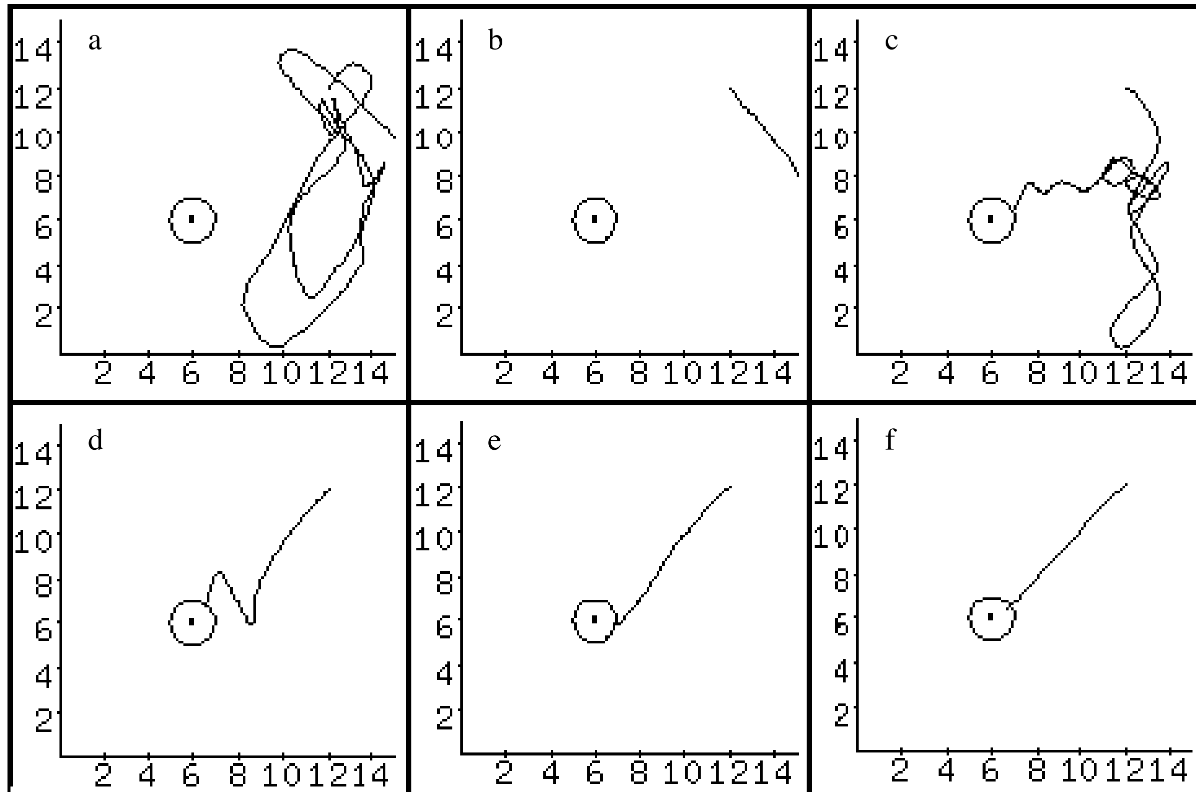

Figure 12. trajectories taken during the learning process. Panels a-f (top left to right, then bottom left to right) are several trajectories taken by the control system during learning. The starting position for each case is at coordinate 12,12, with the target located at coordinate 6,6. The circle around the target position is the area within which the system receives a large positive reinforcement signals. Panels a and b are cases where the system ran into the side of the workspace, where the trial was ended.

units in the *dX* and *dY* layers (Fig. 13). Instead of the roughly uniform distribution of activity that was seen before learning, the *Execution* layer now forces the system to execute specific motor commands.

KEY NETWORK DESIGN ISSUES

Through the design and implementation of this neural network model, we have touched on a number of important network design and neural computation issues. This section explores a number of these issues further.

Overlapping State Representations

The representation of state is a key problem that plagues neural learning systems in general. Some systems, such as the work of Barto, Sutton and others [2, 3] rely on orthogonal or linearly independent state representations. This leads to two problems:

A) A large number of discrete states must be learned and represented.

B) No sharing of information is done between distinct states that might actually yield similar control decisions.

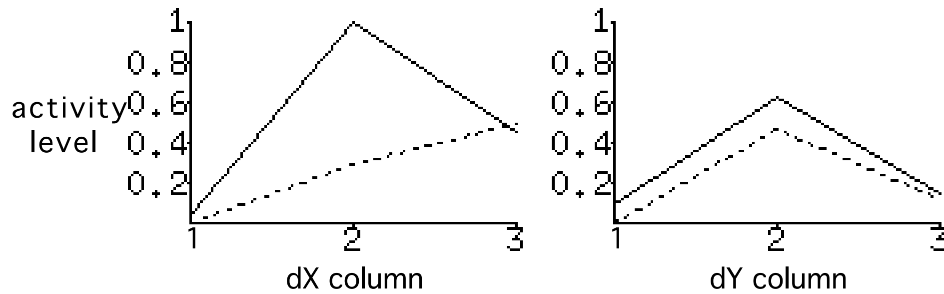

Figure 13. Firing rate plotted against linear position of the priming units (solid) and output units (dotted) of the dX and dY layers, after learning has taken place (arm position A). Note the difference with Fig. 9: the Execution layer now has a much clearer idea as to which units should be primed. In this case, we see a significant commitment to move along the positive X direction, and somewhat along the positive Y direction.

On the other hand, many error propagation techniques, such as backpropagation [13] often require that all computational units participate in every mapping that is learned (this is especially true early in the learning of a training set). Because every hidden unit participates in every mapping, it and the connections that synapse onto it are subject to increments for each input/output mapping. Typically, the modification to a weight for a single mapping conflicts with that of another mapping. These types of conflicts increase the amount of time that is necessary to learn the set of mappings, especially as the number increases.

We would therefore like to find some sort of middle ground where a mapping need not require its own representation, nor should all mappings be encoded by all computing units. Some of these issues have been approached by Jacobs, Jordan, and Barto [14]. The network architecture presented in this chapter is designed to address these issues explicitly. The key features that accomplish this include:

- **Sensory Coding**. Input variables (from sensors) represent values spatially, using a Gaussian distribution of activity. Thus, two similar values are represented by overlapping patterns of activity, but two very different values have representations that do not overlap.
- **Schema Coding**. Generally, different schemas that share common functions will utilize overlapping sets of columns. This is important in terms of efficiently representing the set of schemas, and by the fact that learning in one schema can provide important information for another.
- **Contrast enhancement**.

Importance of Contrast Enhancement for Learning

The contrast enhancement operation provided by the *Mexican-hat* operator possesses three key computational properties.

- The number of active columns within a layer is limited to be some subset of the entire

layer. This implies that only a subset of the columns are allowed to be involved in the representation of a schema, leaving other columns for the representation of other schemas. Consider the case where a large number of columns are allowed to become active and thus participate in a computation. When a reinforcement signal is received, all active columns are considered to be responsible, and are therefore updated in an attempt to improve the system's performance. When learning several different schemas at the same time, this can result in a high degree of overlap, causing the learning due to one schema to interfere significantly with another. Because the contrast enhancement operation reduces this overlap, the interference can be greatly reduced.

- During the learning process, the *Mexican-hat* operator tends to induce a topological representation of the schemas that it learns to represent [15, 16]. Because a μ-schema tends to be active in correlation with its close neighbors, they will tend to acquire similar (but not exactly the same) functions. Thus, a slight shift in activity within a layer (due to some small change in priming) will tend to produce a small difference in the function of the active schema.

Importance of Noise

Noise plays a key role in the search mechanism employed by the system as it attempts to identify the correct outputs given a particular situation. Before learning has occurred, the noise injected into the *activity units* (of the *dX* and *dY* layers) drives the outputs of the system, forcing the exploration of the local workspace. As learning proceeds, the noise is biased by the outputs of the *Execution* layer, until the bias overcomes the noise altogether. This form of stochastic search and biasing of noise is reminiscent of the SRV (stochastic real-valued) units of Gullapalli et al. [17, 18].

Noise is also injected into the *activity units* of the *Execution* layer. This noise affects the set of columns in the execution layer that participate in a control computation. This random switching on and off of columns allows the controller to experiment with different subsets of columns. Through learning, those columns that tend to participate at times when positive reinforcement is received will begin to participate more often. This is done until a set of columns is selected that is most able to learn the control problem at hand.

Modularity of Structure

The column and layer structures have been designed with some degree of modularity. The difference in function between two layers should not be determined by a difference in layer implementation, but by the type of information that flows into and out of a layer. The network described in this chapter utilizes the same structure for the *Planning* and *Execution layers*.

However, the inputs and outputs of these two layers differ significantly.

Using this same modular structure, it is possible to build up more interesting network architectures, such as one that controls a combined arm/hand system. This type of network may be *grown* from the one presented in this chapter by adding additional processing layers for control of the hand, and adding cross-connections between the hand and arm layers.

FUTURE DIRECTIONS

One dimension of future exploration will be the implementation of such a reaching and grasping network, which will ultimately interface to a Puma 560 Arm and a Belgrade/USC Hand for experiments in reaching and grasping within a real environment. We are interested in the development of real-time, on-line control and learning systems that can be used to teach robots to perform interesting tasks within a short period of time. One approach to this problem of learning efficiency is the use of teaching by example, where the teacher demonstrates a motor program to the robot (Lin [19]). The robot first learns to mimic the teacher, and then through reinforcement-based feedback (either provided by the teacher or generated internally), refines the motor programs to increase their success and generality.

In our primate modeling research, we see this work as providing one possible way of understanding why certain brain regions take on particular functions in sensing and motor control, and why different regions are connected together in specific ways. One question that can be asked is that given a set of constraints from a particular network architecture (set of regions and connections) and a set of behavioral requirements (as specified by the environment or experimenter), what functions do specific regions and even neurons take on through the learning process?

The work in modeling of primate behavior and neural systems, and the work in learning in robots has progressed in parallel. The primate domain provides important hints as to how a learning system is able to efficiently acquire the ability to perform new tasks, both from a behavioral and a functional point of view. Robotics provides an environment in which models from the primate side may be tested, analyzed, and improved in agents that must also behave in a real environment. Ultimately, predictions that arise from these models may be brought back to the primate domain in the form of new experiments to be tried or as a better understanding as to how the primate system functions.

ACKNOWLEDGEMENTS

The author would like Professors Michael Arbib, George Bekey, and Ken Goldberg for their help and support in the development of this work. Many thanks are also due to Peter Dominey, Amanda Bischoff, Mike McHenry, Nicolas Schweighofer and David Lotspeich for their many helpful comments on earlier drafts of this chapter.

The simulation of this network was implemented in NSL (Neural Simulation Language) [20], which is available by anonymous ftp (from yorick.usc.edu). Contact Alfredo Weitzenfeld (alfredo@rana.usc.edu) for more information.

REFERENCES

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

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

[3] Barto, A. G. and Bradtke, S. H., *Real-Time Learning and Control using Asynchronous Dynamic Programming*. (1991) TR 91-57, Department of Computer Science, University of Massachusetts, Amherst.

[4] Sutton, R. S., First Results with Dyna, An Integrated Architecture for Learning, Planning and Reacting, in *Neural Networks for Control,* W. T. Miller, R. S. Sutton, and P. J. Werbos, Editors. (1990) MIT Press: Cambridge, Massachusetts. pp. 179 - 95.

[5] Watkins, C. J. C. H. and Dayan, P., Q-Learning. *Machine Learning*, (1992) **8**(3-4):279-92.

[6] Werbos, P. J., A Menu of Designs for Reinforcement Learning Over Time, in *Neural Networks for Control,* W. T. Miller, R. S. Sutton, and P. J. Werbos, Editors. (1990) MIT Press: Cambridge, Massachusetts. pp. 67 - 96.

[7] Fagg, A. H. and Arbib, M. A., A Model of Primate Visual-Motor Conditional Learning. *Journal of Adaptive Behavior*, (1992) **1**(1):3-37.

[8] Mitz, A. R., Godshalk, M., and Wise, S. P., Learning-dependent Neuronal Activity in the Premotor Cortex. *Journal of Neuroscience*, (1991) **11**(6):1855-72.

[9] Barto, A. G., Anderson, C. W., and Sutton, R. S., Synthesis of Nonlinear Control Surfaces by a Layered Associative Search Network. *Biological Cybernetics*, (1982) **43**:175-85.

[10] Barto, A. G. and Sutton, R. S., Landmark Learning: An Illustration of Associative Search. *Biological Cybernetics*, (1981) **42**:1-8.

[11] Mel, B. W., *Connectionist Robot Motion Planning: A Neurally-Inspired Approach to Visually-Guided Reaching*. *Perspectives in Artificial Intelligence*, B. Chandrasekaran, Editor. Vol. 7. (1990) Boston: Academic Press, Inc.

[12] Klopf, A. H., *The Hedonistic Neuron*. (1982) Washington, D. C.: Hemisphere.

[13] Rumelhart, D. E., Hinton, G. E., and Williams, R. J., Learning Internal Representations by Error Propagation, in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition,* D. Rumelhart and J. McClelland, Editors. (1986) MIT Press: pp. 318-62.

[14] Jacobs, R. A., Jordan, M. I., and Barto, A., Task Decomposition through Competition in a Modular Connectionist Architecture: The What and Where Vision Tasks. *Cognitive Science*, (1991) **15**(2):219-50.

[15] von der Malsburg, C. How are nervous structures organized*? in the Proceedings of the International Symposium on Synergetics*. (1983) Springer. pp. 238-49

[16] von der Malsburg, C., Ordered Retinotectal Projections and Brain Organization, in *Self-Organizing Systems - The Emergence of Order,* F. E. Yates, A. Garfinkel, D. O. Walter, and G. Yates, Editors. (1987) Plenum Press: New York. pp. 265-78.

[17] Gullapalli, V., Grupen, R. A., and Barto, A. G. Learning Reactive Admittance Control. *in the Proceedings of the IEEE International Conference on Robotics and Automation*. (1992) Nice, France: pp. 1475-80

[18] Gullapalli, V., A Stochastic Reinforcement Learning Algorithm for Learning Real-Valued Functions. *Neural Networks*, (1990) **3**(6):671-92.

[19] Lin, L. J., Self-Improving Reactive Agents Based on Reinforcement Learning, Planning, and Teaching. *Machine Learning*, (1992) **8**(3-4):293-321.

[20] Weitzenfeld, A., *NSL - Neural Simulation Language Version 2.1*. (1991) TR 91-05, Center for Neural Engineering, University of Southern California, Los Angeles, CA.