# An incremental approach to developing intelligent neural network controllers for robots[*]

Lisa A. Meeden
Computer Science Program
Swarthmore College
500 College Ave
Swarthmore, PA 19081 USA
meeden@cs.swarthmore.edu

**Abstract**

By beginning with simple reactive behaviors and gradually building up to more memory-dependent behaviors, it may be possible for connectionist systems to eventually achieve the level of planning. This paper focuses on an intermediate step in this incremental process, where the appropriate means of providing guidance to adapting controllers is explored. A local and a global method of reinforcement learning are contrasted—a special form of back-propagation and an evolutionary algorithm. These methods are applied to a neural network controller for a simple robot. A number of experiments are described where the presence of explicit goals and the immediacy of reinforcement are varied. These experiments reveal how various types of guidance can affect the final control behavior. The results show that the respective advantages and disadvantages of these two adaptation methods are complementary, suggesting that some hybrid of the two may be the most effective method. Concluding remarks discuss the next incremental steps towards more complex control behaviors.

[*] Appeared in *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, June 1996, Volume 26, Number 3, pages 474–485

# I. INTRODUCTION

Neural networks offer one of the most promising means for investigating robot control because connectionist methodology allows the task demands rather than the designer's biases to be the primary force in shaping the system's development. By actively limiting the effect of our presuppositions on the implementation, we may be able to discover radically different forms of control.

At the initial design phase, connectionist methodology provides useful constraints by encouraging bottom-up construction. Input can come directly from the sensors and output can feed directly into the actuators creating a close coupling of perception and action. This immediate interplay between sensing and acting is crucial to producing the real-time reactive behavior necessary in robotics. Because adaptation is fundamental to the connectionist paradigm, the designer need not determine what form the internal knowledge will take or what specific function it will serve. Instead, based on the training task, the system will construct its own internal representations built directly from the sensor readings to achieve the desired behavior. However, the designer still has an important role to play in this development process which includes determining: the network topology, the network parameters, the adaptation scheme, the input and output representations, the robot's physical characteristics, and the robot's environment.

After this development process has been set in motion and the system has reached an adequate level of performance at the task, its method can be dissected and a high-level understanding of its control principles can be described. As Clark has noted, connectionism "inverts the official classical ordering, in which a high-level understanding comes first and closely guides the search for algorithms" [9]. Through adaptation, the system's solution to the task emerges as it discovers the key features of the problem space rather than simply being the product of the designer's understanding of the domain.

By employing this basic technique of constructing a neural network controller, adapting it for an agent and an environment, and then analyzing the emergent behavior, a number of researchers have found quite interesting control mechanisms within their systems [3, 14, 22, 28]. However, there are limitations to what a fundamentally reactive system can accomplish. Many tasks we would like a robot to perform require some kind of planning. If the connectionist approach cannot go beyond the level of reactive behavior to planned behavior its usefulness in robotics will probably be restricted to the front-end processing of perceptual information. This paper will argue that an incremental approach, beginning with simple reactive behavior and gradually building up to more memory-dependent behavior, is a possible avenue for connectionist systems to eventually achieve the level of planning.

In recent years there have been a number of different proposals for incrementally constructing intelligent agents [6, 7, 30, 31, 33]. This paper will focus on the suggestions made by Waltz in [31] because they address high-level distinctions that can be implemented in neural networks. He offers eight guiding principles to building an intelligent architecture; the first five of these principles lead up to planning, and the final three go beyond it. This research adopts the five principles relevant to planning, with slight modifications to better fit them into a connectionist framework.

1. Use associative memory as the overall mechanism.
   $state \rightarrow action$

2. Populate the associative memory system with sequenced rote experiences.

3. Include mechanisms to automatically generalize across rote memories.

4. Include innate drive and evaluation systems to provide the robot with guidance for its actions.
   $state + goal \rightarrow action + evaluation$

5. Include control structures to allow planning.
   $state + goal + plan \rightarrow action + evaluation$

These principles provide a foundation for an intelligent robot controller, but key issues remain unresolved. With respect to the first, second, and third principles, what sort of associative memory should be employed to ensure recognition of sequences and ease of generalization? With respect to the fourth principle, how should goals be specified to produce the appropriate type of motivation and what kind of evaluation should be given to best direct the adaptation? Finally, with respect to the fifth principle, what will constitute a plan and how will plans develop?

This paper will address all of these questions to some extent but will focus on the questions raised by the fourth principle. In terms of the first three principles, it has already been well demonstrated that connectionist networks are fundamentally associative engines that naturally perform generalizations [24, 9]. However it is not yet clear, in terms of the fourth principle, what are the best ways to incorporate goals or to provide adaptation evaluation for robotics domains. Since this approach is incremental, the planning issues of the fifth principle cannot be addressed until these previous problems of the fourth principle are explored.

In this paper, a local and a global reinforcement method of adaptation for neural network controllers are contrasted—a special form of back-propagation and an evolutionary algorithm. These methods are applied to a simple robot, called carbot. A number of experiments are described where the presence of explicit goals and the immediacy of reinforcement are varied. These experiments are meant to shed light on the questions, raised by the fourth principle, of the most effective means for providing goals and evaluation to an adapting robot controller.

The paper is organized as follows. In Section II, the robot, the task, and the control network are described. In addition, the advantages and disadvantages of real-world versus simulation training are discussed. In Section III, a brief background is given for both the local and global adaptation methods as well as an explanation of the specific implementation details used here. In Section IV, the experiments and results are described. Finally in Section V, there is a discussion of the results with respect to the overarching aim of incrementally moving towards connectionist planning.

## II. THE ROBOT

Carbot is a modified toy car (approximately 15 cm wide, 23 cm long, and 10 cm high) controlled by a programmable mini-board (designed at MIT [23]). During operation, carbot was tethered to a PC and was controlled by a remote connectionist network that communicated with the mini-board through the serial port. Carbot was inexpensive to build, primarily because it makes use of primitive sensors—no lasers or video. Instead it has just two types

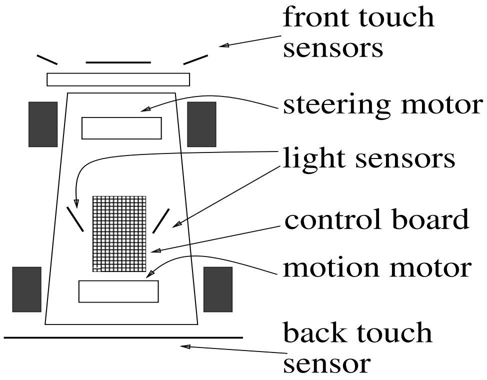

Figure 1: Positioning of sensors, motors, and control board on carbot. Note the 30 degree angle of the light sensors.

of sensors: digital touch sensors on the front and back bumpers, and analog light sensors on stalks near the back which are directed 30 degrees to each side of carbot. For movement it has two servo-motors; one controls forward and backward motion and the other steering. See Figure 1 for a schematic drawing of carbot that shows the position of the sensors and motors.

### *A. The Environment and Task*

The environment is a rectangular box (approximately 1.2 m by 0.6 m) with a light in one corner. Carbot's task within this sparse environment is multi-dimensional: it must learn low-level reactive responses as well as goal-guided behavior. At the reactive level, carbot receives negative reinforcement any time it contacts one of the walls or is not moving. Conceptually this could be considered rudimentary navigation—carbot must learn to continually keep moving while avoiding the boundaries of its environment. See [26] for a detailed description of factors that affect a network's performance when learning this type of simple reactive task.

At the goal level, carbot must either seek or avoid the light depending on the current goal. A positive value for the goal indicates that carbot should seek out the light until a maximum light reading is obtained. Once this is accomplished, the goal automatically switches to a negative value, indicating that carbot should avoid the light until a minimum light reading is obtained. Successful avoidance switches the goal back to seek-mode again. The goal varies in this periodic manner throughout the task, seeking is immediately followed by avoiding and so on. Carbot receives negative reinforcement any time it does not follow the light gradient correctly for the given goal. Otherwise it receives positive reinforcement.

At first glance this task seems quite simple; however there are a number of aspects of carbot's physical characteristics that make the task problematic. First, carbot, as its name suggests, is built like a car and its turning radius is larger than the smallest dimension of the environment. Second, carbot cannot control its speed, but must always keep moving. Each action is executed for one second, and on average straight motion propels carbot 17.8 centimeters (almost 30% of the smallest environment dimension) and turning motion results in a linear change in position of 11.4 centimeters (almost 20% of the smalled environment dimension) as well as a 24 degree change in heading. Third, carbot's touch sensors provide

Table 1: Example sequences

| | |
|---|---|
| A | 1 2 3 4 5 1 2 3 4 5 ... |
| B | 1 2 3 4 5 1 3 5 2 4 ... |
| C | 1 2 3 1 2 4 1 2 5 5 ... |

no advance warning of upcoming obstacles, as sonar sensors would. Finally, because of the position of the light sensors, carbot strongly senses light in only a 60 degree range centered on its front end. In summary, carbot's maneuverability is limited to a few gross actions and its perceptual abilities are quite primitive, making this seemingly simple task much more difficult.

### *B. The Control Network*

One of the most basic connectionist architectures is a feedforward network where an input layer of units feeds into a hidden layer which then feeds into an output layer. By interpreting the inputs as perceptions and the outputs as actions a typical feedforward network can learn to react appropriately for a given robotics task. One difficulty with the standard feedforward architecture is that when time is an important aspect of the problem to be modeled, it is not clear how to represent time in an efficient and useful manner. Certainly, for the ultimate goal of planning, timing information is crucial since plans involve temporal sequences of actions.

For example, suppose we wanted a neural network to learn one of the three sequences shown in Table 1. If the network were presented with only one element at a time, then to learn such a sequence it must master the transitions between the elements. These transitions may depend on one or more elements that appeared an arbitrary number of steps back in time. The difficulty of a sequence to be learned can be measured by the complexity of its transitions. Sequence A is the easiest to learn because each transition depends only on the current element—2 is always followed by 3. No memory is necessary and a feedforward network could master this sequence. However, sequence B is harder to learn because each transition depends on the the previous and current element—2 is followed by 3 or 4. To accomplish this transition, memory of the previous element is necessary and a feedforward network could not master this sequence. Finally, sequence C is the most difficult to learn because of the repeating sub-sequence 1 2. Now the transition from 2 may be either 3, 4, or 5 depending on the previous two elements. By relaxing the feedforward restriction and allowing feedback connections, a network can learn to be sensitive to these kinds of temporal changes in the input.

Recurrent networks, which allow lateral and backward connections between units, offer a way to implicitly represent time by the effect it has on processing. One form of simple recurrent network (these are termed *simple* because the location of recurrent connections is restricted to a single layer) was developed by Elman [12]. In this architecture (shown in Figure 2), each unit in the hidden layer is connected to all the other hidden units including itself. With this type of recurrence, the network can develop a short-term memory of its own encodings of the past input states and can therefore learn to respond appropriately to sequences such as B and C in Table 1.

This control network gathers input data from the sensors and determines how to set the

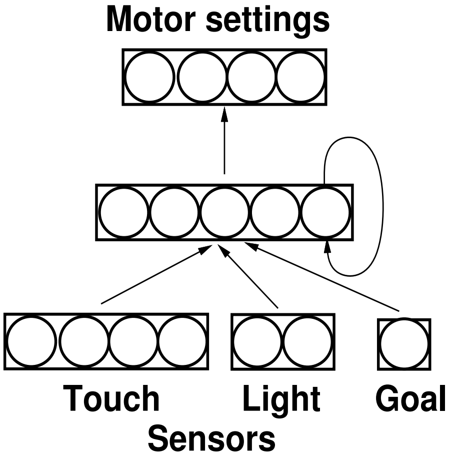

Figure 2: The control network. The arrows indicate which layers are fully connected. The hidden layer is fully connected to itself.

motors for the next time step. There are three sets of input units; two sets are for sensors, and one is for goals. The first set of units in the input layer represent the state of the digital touch sensors—three in front and one in back. Two of the front sensors are out to either side so that carbot can sense side collisions when moving forward. When the digital sensors are triggered by contact with an obstacle they take on the value 1, otherwise they are 0. The next set of two units represent the state of the analog light sensors, whose values can range from 0.0 to 1.0 (due to ambient light, their values rarely fall below 0.25). The final set of one unit encodes the network's goals. Currently only two goals are used: +1 for seeking the light and -1 for avoiding the light.

The four output units of the network determine how to set the motors for the next time step. Carbot is propelled by two motors each requiring two units to specify its state. The first output unit represents the spin direction of the rear motor which determines direction of motion—forward or backward. The second unit designates the state of the motor as on or off. The third unit represents the spin direction of the front motor which determines the direction of turning—right or left. The fourth unit designates the state of the front motor as on or off. Note that in order to turn, carbot must have both motors running; the back motor provides movement while the front motor steers. In order to go straight, carbot must have only the back motor on.

### *C. Real-World versus Simulation Training*

Since carbot has to physically move in the world, it takes a long time to train and test a particular controller network. Each real-world action is executed for one second, so 5,000 actions require approximately 1.5 hours to be completed. To alleviate this time limitation, a software simulation was implemented.

Although simulators are usually not very much like reality, when the simulator is based on an actual robot, the simulator's behavior can be programmed to closely correspond with the real-world behavior. Harvey, Husbands, and Cliff [20] suggest some ways to ensure that a simulator stays in close step with reality:

1. Simulations of the inputs to sensors and the outputs to actuators should be based on carefully collected empirical data.
2. Noise must be taken into account at all levels.
3. A range of unstructured, dynamic environments should be used to ensure robustness.
4. The simulation should be calibrated by testing adapted architectures in the real robot.

In constructing carbot's simulator, all of these suggestions were followed. As per the first suggestion, carbot's sensor readings, average turning radius and distance traveled for each action in the real world were empirically determined and used as the basis for the simulator. With respect to the second suggestion, small amounts of random noise were added to carbot's integer heading (-4 to +4), real position (-2.0 to +2.0), and real light sensor readings (0 to 0.11) after each action taken within the simulator. For the third suggestion of exploring a range of environments, although only enclosed, rectangular spaces were used, a number of different dimensions were tried. The chosen dimensions proved to be hard to maneuver within, but solutions were still readily found.

For the final suggestion, a number of experiments were done to verify that the sorts of controllers that worked well in the simulator would also work well in the real robot. Controller networks trained in the simulator were transplanted to the actual robot and tested without further training. The behaviors produced by the same network on the simulator and on the robot were compared in terms of percent time punished. These transplant tests provided a very encouraging result: the robot's performance was always superior to the simulator's performance. This result is probably due to the additional noise provided in the simulator. The actual robot's movements are only occasionally noisy while the simulator's movements are systematically noisy and this added noise enhances learning by exposing the controller to a wider range of environmental conditions[1].

Thus the simulated robot's behavior was determined to be close enough to the actual behavior of carbot to warrant use of the simulator for training the robot controllers. The simulator is used to test hypotheses and to develop useful architectures that are then applied back to the real robot. This saves a significant amount of time since 5,000 simulated actions can be executed in less than a minute (a speedup factor of about 90). Another benefit of using the simulator is that the behavior tests can be more precisely monitored and compared.

## III. REINFORCEMENT LEARNING METHODS

Robotics problems are typically defined in terms of abstract goals (i.e. remain viable for as long as possible) rather than specific perception to action pairs, so explicit, moment-to-moment guidance is not usually available. Reinforcement procedures are well suited for this type of learning because they only require scalar values to encode the relative desirability

[1] As another example of this phenomenon, Pomerleau found that in training a neural network to drive a large vehicle, it was beneficial to train the network not only on good data provided by a human driver, but also on shifted versions of this data which forced the controller to learn how to recover from errors [29].

of particular states. The magnitude of this value can be used to reflect the degree of the state's goodness. Typically, positive values are thought of as rewards and negative values as punishment. For example, suppose that carbot has just bumped into a wall triggering its front sensors. Any action that moves it away from the wall and clears its sensors should be rewarded, while any action that persists in bumping into the wall should be punished. There is not necessarily one right or wrong action for a given situation, and even if there were, it may not be known *a priori*. Reinforcement procedures allow the learning to be guided at the abstract, goal level.

In this paper, two very different kinds of reinforcement procedures have been used to adapt the weights of the control networks—one is a local method while the other is a global method. The local method is a special back-propagation algorithm that updates the weights immediately upon receiving reinforcement. The global method is an evolutionary algorithm which tests out a collection of networks in the environment for an extended period of time. From this test it obtains a global fitness measure that is then used to bias the subsequent adaptation.

### *A. Local Method: Back-Propagation*

In half of the experiments, the control networks were trained with a modified version of the complementary reinforcement back-propagation (CRBP) learning algorithm [1]. Back-propagation learning requires precise error measures for each output produced by a network so that gradient descent on the error can be performed. CRBP provides these exact error measures from the abstract reward and punishment signals as follows.

The output is determined by a two-step process. First, a forward propagation of the input values produces a real-valued vector on the output layer called the *search vector* $S$. Each real value of the search vector is interpreted as the probability that the associated random bit takes on the value 1. Then from these probabilities a binary *output vector* $O$ is stochastically produced (this non-determinism provides some exploration for the learning). If $O$ is rewarded (i.e. the reinforcement value is positive), then learning should push the network towards this vector, so the error measure $(O - S)$ is back-propagated. If $O$ is punished (i.e. the reinforcement value is negative), then learning should push the network away from this vector, but the appropriate direction is not clear. CRBP chooses to push the network directly toward the complement of $O$, using the error measure $((1 - O) - S)$.

Using this algorithm, rewarded outputs will be more likely to occur again and punished outputs will tend to produce the complement output vector in similar situations. Let's consider the effect of using the complement as the target for punished actions in the carbot domain. The four output bits can be summarized as follows: forward/backward, motion motor on/off, right/left, turning motor on/off. For most of these bits, taking the complement of a punished action makes intuitive sense: the direction of motion will be switched (left versus right or forward versus backward) and the type of motion will be switched (linear versus curved). However, for the second bit, which determines whether carbot is moving, the complement is not a good choice. Recall that one aspect of the reinforcement task is that carbot must keep moving at all times. For this to occur this particular output bit must always be on. So whenever carbot is moving and receives punishment, taking the complement will result in a non-moving target. This is an unfortunate effect, but the controller networks quickly learn to overcome this aspect of the target by developing a high positive bias value for this output node.

Another feature of CRBP is that there are different learning rates for reward and punishment. When the network is rewarded, we can be confident that we have good information to learn from because the current state led to positive reinforcement. Therefore, we should use a high learning rate. In contrast, when the network is punished, we know that the current state is not desirable, but we can only arbitrarily pick the complement state as our target. There is no guarantee that the complement is a good choice, so our learning rate should be lower.

*Implementation Details*

Adaptation begins by initializing all the weights in a control network to random values between $-1.0$ and $+1.0$. Unlike the evolutionary algorithm to be described next, CRBP adapts a single control network throughout one long, connected series of actions. So starting from a random situation, carbot's future situations are determined by the actions produced by the constantly changing control network. With CRBP all types of punishment are treated equally. Regardless of whether the punishment came from not moving, bumping into a wall, or not following the light gradient correctly, the reinforcement value is -1 and the consequences are the same: the complement of the punished action is back-propagated. Similarly all types of reward, keeping moving, following the light gradient correctly, and achieving a light goal, are treated equally—each receives $+1$ as the reinforcement value[2].

The following set of steps are considered one *cycle* of processing and were iterated until a learning criterion was achieved or until a maximum of $750,000$ cycles was completed (a cycle equates with performing one robot action). The learning criterion was met whenever the percent of actions punished for $5,000$ consecutive cycles was less than or equal to 35%.

1. Get input from the sensors
2. Forward propagate input to produce real-valued $S$
3. Stochastically determine binary-valued $O$ from $S$
4. Execute action $O$ on carbot
5. Determine reinforcement
6. Determine error
   if rewarded then $E = O - S$
   if punished then $E = ((1 - O) - S)$
   if no reinforcement then $E = 0$
7. If $E \neq 0$ then back-propagate $E$
   reward learning rate $= 0.3$
   punish learning rate $= 0.1$

[2] It would be possible to differentiate between types of punishment or types of reward by varying the learning rate by type. For example, following the light gradient correctly could result in a 0.3 learning rate, while actually achieving the final light goal could result in a 1.0 learning rate. This type of variation has been tried, but the results were not significantly different than those obtained using the standard (and simpler) method.

In CRBP, the results of the reinforcement are immediately used to update the network weights, thus changing the controller after every action.

### *B. Global Method: Genetic Algorithm*

Genetic algorithms (GAs) were originally used for function optimization. However with the arrival of classifier systems, which use GAs for rule discovery, they became linked with a reinforcement procedure [21, 5, 17, 11, 10]. But as will be seen below, a GA was, in a sense, already a reinforcement procedure prior to the advent of classifier systems because it operates on information about the relative performance of potential solutions [32]. Goldberg provides a thorough introduction to both the optimization and reinforcement applications of GAs [16].

Genetic algorithms are based on the theory of natural selection in evolution. GAs work on a *population* of individuals, where each individual represents a possible solution to the given problem. After the initial population is randomly generated, the algorithm evolves the population through the use of three operators: selection, crossover, and mutation. *Selection* determines which individuals from the current generation to copy to the next generation based on each individual's fitness, where *fitness* is some measure of the goodness of an individual that the GA is trying to maximize. A certain portion of the individuals chosen by selection will be mated through crossover. *Crossover* creates a new offspring by combining complementary portions of the genetic strings of two parents. Finally, through *mutation* some portion of the chosen individuals will have random pieces of their genetic material altered.

The selection process ensures that individuals with above average fitness are more likely to be chosen for inclusion in the next generation than individuals with below average fitness. Therefore "genetic algorithms are capable of performing a global search of a space because they can rely on hyperplane sampling to guide the search instead of searching along the gradient of a function" as back-propagation does [32].

#### *Combining Genetic Algorithms and Neural Networks*

There are many interesting possibilities for applying genetic algorithms to neural networks. GAs have been used to find good initial network weights, to tune network learning parameters, to determine network structure, to evolve network learning algorithms, and to learn network weights [4, 18, 20, 8, 32]. It is the last option—learning weights—that will be used here: for the carbot problem, the network architecture is fixed (as shown in Figure 2) and the GA works to adapt an appropriate set of weights.

Applying genetic algorithms to networks has not been as straightforward as other types of GA applications. Traditionally individuals in GA populations have been represented as bit strings. Weights in connectionist networks are real-valued, and converting them into a binary encoding would entail arbitrarily discretizing them to a particular precision. So for many GA applications to networks, individuals are represented as real-coded vectors of weights (for example see [15, 32]), and this was done for the carbot domain as well.

Another tradition in GAs is that the crossover operator is used much more frequently than the mutation operator. In the carbot domain, crossover would create a new set of weights by taking some weights from one successful network and the rest from another successful network. However, these two networks could be using very different strategies for solving the problem. Creating an offspring by recombining portions of their weights may result in an extremely poor alternative solution. In fact even if the two networks are employing the same strategy, it is probably instantiated in the weights in very different ways. So again, recombination may be quite unsuccessful. Networks tend to solve problems in a distributed, holistic fashion and thus may not even have useful building blocks to contribute to a recombined solution.

In initial experiments with the GA, crossover was used in adapting network controllers for carbot, but this operator did not improve and often hurt performance. As a first attempt, standard crossover was applied, where one random point on the genetic string was picked and a swap was done. In a second attempt, all the incoming weights to a unit and its bias were treated as a building block and located together on the genetic string. Crossover was restricted to occurring between such blocks. Even with this more sophisticated form of crossover, performance suffered. Therefore due to the problematic nature of crossover for network representations, this form of recombination was not used in the experiments described here (although crossover has been used in other GA applications to networks—for a good discussion of this topic see [4]).

Instead, new individuals were created solely by mutation. Harvey notes that there has been "surprising success (in some circumstances) of what has come to be called *naive evolution*; i.e. mutation only, contrary to normal GA folklore which emphasizes the significance of crossover" [19]. Further, he found that the optimal mutation rates were between one and two mutations per individual and this was nearly invariant over the length of the individual's representation. In some instances, even a much more brute-force form of mutation has been successful for GA applications to networks. Rather than just altering a small number of the weights, every weight on the entire string is modified by a random amount [15] from a range as large as -10 to +10 [32]. This form of mutation essentially creates a new random point located within a specific radius from the parent.

Although using real-encoded strings and depending on mutation for the creation of new genetic material are not standard GA choices, some research has shown that they are crucial to obtaining positive results for evolving neural networks [27, 32].

*Implementation Details*

Each individual in the GA's population is a possible control network for carbot. Therefore the fitness function should measure the effectiveness of a network for controlling carbot in its task of doing rudimentary navigation and periodically seeking and avoiding the light. The fitness was determined as follows. A random situation was selected for carbot, where a situation consisted of an initial goal (either seek or avoid), an $(x, y)$ position in the simulator, and a heading. The particular network then controlled carbot for 20 actions starting from that point. Each action received either a reward or punishment and the sum of this reinforcement over all 20 actions was calculated. Five more random situations were chosen and the same process was done again. The fitness was the sum of the reinforcement received over all six random starts[3].

Allocating only 120 actions to evaluate each control network results in a fairly noisy measure. Yet, according to Fitzpatrick and Grefenstette, it is better to obtain quick, rough estimates and to allow the GA to consider many candidate solutions than it is to attempt to obtain highly accurate evaluations with a smaller population size [13]. Baluja found evidence

[3] Other fitness options were also explored, such as averaging the results of the random starts or using the minimum of the random starts. Neither of these options improved performance.

to support Fitpatrick's and Grefenstette's claim [2].

Unlike the case for CRBP, in the GA, each type of reinforcement was given a different reinforcement value:

- Accomplished a light goal: +50
- Not moving: −4
- Any touch sensor triggered: −2
- Not following light gradient correctly for goal: −1
- Following light gradient correctly for goal: +5

Note that each action received only one of these possible values and the evaluation of an action was checked in the order given above. So if a goal was accomplished by an action that also caused carbot to hit a wall, the reinforcement for that action would be +50 and not −2 or the combination of the two, +48.

Based on these reinforcement values we can determine the range of possible fitness values. Recall that the sum of the reinforcement received over six sequences of 20 actions is used to determine a network's fitness. To calculate the minimum fitness value, assume that every action results in the minimum reinforcement value (-4 for not moving).

$min(fitness) = actions \times -4 = 120 \times -4 = -480$

To calculate the maximum fitness value, we need to know the optimal strategy for seeking and avoiding the light given this particular domain and carbot's capabilities. However, it is not clear what the optimal strategy would be. In practice, carbot's most successful strategies have required at least five actions. Let's assume the optimal strategy can accomplish a goal every five actions and is never punished. So a fifth of the actions will receive the accomplished goal reinforcement of +50 while the remaining actions will receive a reward of +5 for correctly following the light gradient.

$max(fitness) = (0.2actions \times 50) + (0.8actions \times 5) = (24 \times 50) + (96 \times 5) = +1680$

In the experiments described below, the genetic algorithm adapted a population of 50 networks. Each network was initialized with random weights between −1.0 and +1.0 and then its fitness was measured as just described. Typically this initial population had an average fitness of approximately −200 where the worst individual had a fitness of about −450 and the best had a fitness of about +200. After training, these measures improved to approximately +600 for the average fitness, +200 for the worst individual, and +1150 for the best individual.

Processing in GAs is measured in terms of *generations.* A generation is completed when a new population has been created through selection and mutation within the old population. The carbot experiments used a technique called *tournament selection* to choose the parents for the next generation. Two individuals were randomly chosen from the population and competed in a tournament. The one with the higher fitness was declared the winner. By forcing both robots to attempt the same tasks, the evaluation of the robots is reduced to the simple question: which performed the best across these tasks [19]?

Each GA run consisted of at most 2500 generations. After each generation, the current best individual was determined. If the best individual had improved since the previous generation, then it was tested for $5,000$ actions to determine if it had achieved the same learning criterion used for CRBP: a percent punishment less than or equal to 35%. If so, then the adaptation process was ended.

Iterating the following set of steps 50 times (the population size) constituted a generation. At the end of this process the current best individual (in terms of fitness) was returned as the solution.

1. Use tournament selection to determine a parent
2. Replace the loser by a mutation of the winner
   Create mutation by updating all the weights in the winner
   by random values between $-1$ and $+1$
3. Determine the fitness of the new individual

Individuals in the GA for this carbot domain had length 89—the networks each had 80 weights and 9 biases. Referring back to Figure 2, note that the layers are fully connected. There are $7 \times 5 = 35$ weights from the input to hidden layer, $5 \times 5 = 25$ weights from the hidden to hidden layer, and $5 \times 4 = 20$ weights from the hidden to output layer. Only units in the hidden and output layers have biases: $5 + 4 = 9$.

### *C. Comparison of Methods*

GAs tend to be a very robust method because they operate on a population of possible solutions rather than just a single solution as is done in CRBP. Using CRBP, there is much more potential for getting stuck in local minima. For example, if CRBP begins with a poor set of initial weights it may never be able to converge on a solution. Whereas a GA, given enough processing time, can more reliably find at least a reasonable solution.

In terms of carbot actions executed, however, this processing price can be steep. For the GA, the total number of actions performed in a single run is:
$actions = popsize \times generations \times starts \times steps = 1.5 \times 10^7$
where the population size is 50, the maximum number of generations is 2500, the number of random starts is 6, and the number of steps per start is 20. In contrast, the CRBP runs performed at most $7.5 \times 10^5$ actions (20 times fewer actions).

Perhaps the most significant difference between these two methods is in the immediacy of the reinforcement used. CRBP was designed to learn from direct reinforcement—an action is executed and an evaluation of its goodness is immediately expected. GAs learn from indirect reinforcement—a sequence of actions is executed and some overall measure of fitness is returned upon completion. Thus GAs are an inherently delayed reinforcement procedure. The experiments in section IV, subsection E will examine this difference in the adaptation methods and test whether CRBP can learn from delayed reinforcement.

## IV. EXPERIMENTS

For each type of experiment described below, ten trials were done with CRBP and ten trials were done with the GA. However, if a particular controller network did not converge to a

Table 2: Experimental setup

| | Information used | |
|---|---|---|
| Variation | Explicit goal | Light gradient |
| *control* | yes | yes |
| *nogoal* | no | yes |
| *delay* | yes | no |

reasonable level of performance within the limit of 750,000 cycles for CRBP or the limit of 2,500 generations for the GA, it was excluded from subsequent analysis. All the experiments were done with the simulator and three types of experiments were conducted.

In the first set, called the *controls*, the adaptation methods had access to the most information—both goals and reinforcement about the light gradient were provided. On the input layer an explicit goal was given: the sign of this value determined which mode the controller should be in (either seek or avoid). This goal is explicit in the sense that it is unambiguous, however the controller must learn what this goal means and how to act based on its value. The light gradient reinforcement was given through the reinforcement procedure. When in seek-mode, a controller was rewarded if the sum of carbot's light sensor readings increased relative to the previous time step; in avoid-mode, a controller was rewarded if the sum of the readings decreased relative to the previous time step; otherwise it was punished.

In the second set of experiments, called the *nogoals*, the goal unit remained in the input layer but its value was always zero. Even though an explicit goal was not provided the robot still had to periodically seek and avoid the light just as before. The difference in this case, is that the control network only gets implicit information about the current goal through reinforcement about the light gradient. For instance when carbot should be seeking the light, the control network will be punished for choosing actions that do not increase its light readings. By varying the presence of an explicit goal we can investigate whether having goals as input enhances the adaptation method's ability to learn a given task (as we would expect).

In the final set of experiments, called the *delays*, the goal unit's activation was returned and the light gradient reinforcement was removed. Under this variation, both the GA and CRBP still received reinforcement about contacting walls and not moving, but only received information about the light upon achieving the current goal. By varying the immediacy of the light reinforcement we can investigate whether the adaptation method can adequately learn a task from delayed reinforcement. Table 2 summarizes the experimental setup.

## *A. Evaluating Behavior*

Two methods were used to evaluate and compare each network's performance after adaptation was completed. The overall goal of a reinforcement learner is to maximize reward and to minimize punishment. So one way to evaluate a controller is to examine how well its action decisions minimize the punishment received over an extended period of behavior. In addition, it is important that a controller's behavior be robust. So another means of evaluation is to measure how well a controller reacts to a wide variety of situations.

In the first method, carbot was positioned in the center of the environment, with a heading of 180 degrees, and with an initial goal of seeking the light. Each trained network

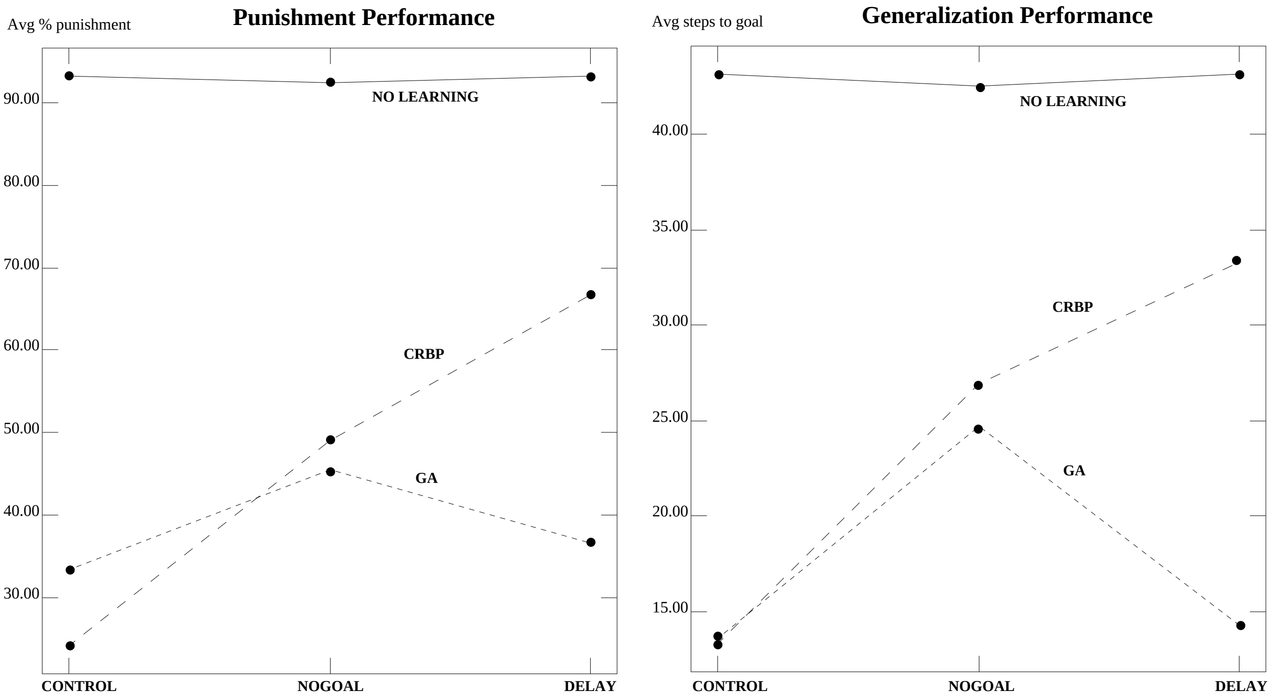

Figure 3: Summary of mean performance across the experimental variations for the two different methods of evaluation.

then controlled carbot for 300 consecutive actions starting from this point. The goals changed periodically just as during the adaptation phase. The percent of these actions that led to punishment was compared. Clearly, a low value for this testing measure indicates that a controller is successful at minimizing punishment.

In the second method, 100 random situations were selected for generalization testing. A situation consisted of a goal, an $(x, y)$ position, and a heading. There are $9 \times 10^5$ possible situations if the environment is discretized into 50 by 25 positional units and the heading into one-degree units (2 goals, 50 $x$ positions, 25 $y$ positions, and 360 headings). Thirty of these 100 random situations proved to be too simple—either no actions were needed or a single action was required to achieve the particular goal. These simple situations were removed, and the remaining 70 random situations served as the generalization test base. Every controller network was started at each of these situations and the number of steps needed to achieve the given goal was calculated. If a controller was unable to reach the goal within 50 steps, it received a score of 51 for that particular situation. The average number of steps taken over all 70 situations was compared. A low value for this measure indicates that a controller's behavior is quite robust.

In comparing the experimental results, classical analysis of variance testing was used. Several types of comparisons were done: across an adaptation type (e.g. GA *control* versus GA *nogoal*)) and between adaptation types for a particular variation (e.g. CRBP *control* versus GA *control*).

Figure 3 summarizes the quantitative differences in behavior that resulted as the exper-

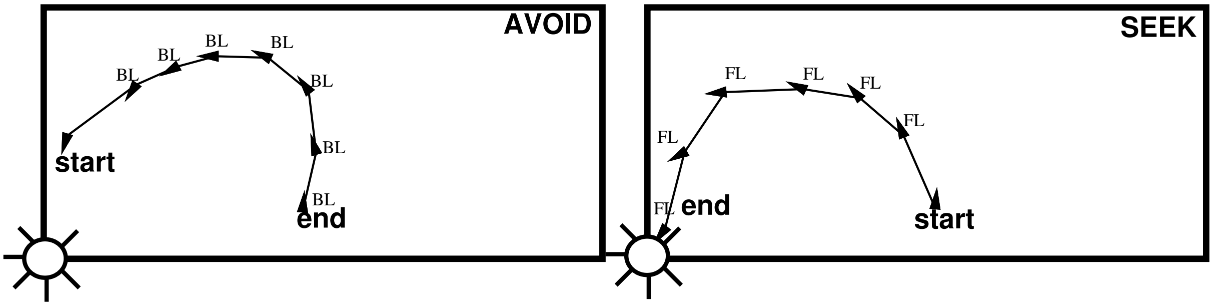

Figure 4: Semi-circle strategy: Executes a backward-left series for avoiding the light and a forward-left series for seeking the light. This strategy was found by 56% of the CRBP *controls*, 12.5% of the GA *controls*, and 12.5% of the GA *delays*.

imental conditions were varied. Both methods of evaluation produced very similar results. Prior to learning, the behavior was almost completely ineffective receiving punishment over 90% of the time and rarely achieving a goal within the 50 step limit. In the *control* condition, both CRBP and the GA were quite successful, with CRBP outperforming the GA in minimizing punishment. In the *nogoal* condition, the performance of both CRBP and the GA suffered relative to the *control* condition, but the GA was slightly less affected. Finally in the *delay* condition, the GA was drastically more effective than CRBP and almost equaled its performance in the *control* condition. These quantitative results will be discussed more fully in the following subsections. In addition, the interesting qualitative differences in behavior will be described.

### *B. Baselines*

As a baseline for measuring the effects of the adaptation methods, the performance of the networks prior to any learning was examined. Ten networks with the architecture shown in Figure 2 were initialized with random weights and both evaluation methods were executed. When provided with an explicit goal (the *control* setup), on average these networks received a punishment score of 93.22% and a generalization score of 43.11 steps. When the goal unit's activation was removed (the *nogoal* setup), the averages were similar with a punishment score of 92.46% and a generalization score of 42.50 steps. Note that the baseline behavior for the *delay* experiments is the same as that for the *control* experiments because the change between these two setups occurs in the how the learning is executed and not in the input values provided. These values are depicted in the "NO LEARNING" curves of Figure 3.

### *C. Controls*

For CRBP controllers trained under the *control* conditions, nine of the ten networks reached the learning criterion. The remaining network was not included in the analysis. The majority of the viable controllers (five of nine) developed a semi-circle strategy to solve the task. Figure 4 shows one example of this strategy. The environment is depicted from a birdseye view where the light is positioned at the lower-left corner. Carbot's path is shown as a curve with the starting and ending locations indicated. Carbot's position and heading over time

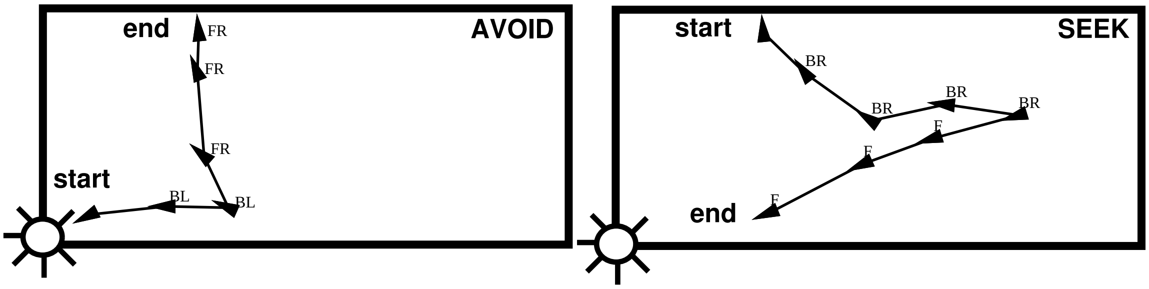

Figure 5: One-point turn strategy: Executes a backward-left series followed by a forward-right series to avoid the light. Executes a backward-right series followed by a forward series to seek the light. This strategy was found by 44% of the CRBP *controls*.

along this path are given by an arrowhead. The arrowhead marks the location of the center of the robot and does not reflect its true size (which is larger). Next to each arrowhead is a shorthand description of the action taken to reach that position (where FL means forward-left, BL means backward-left). As can be seen in Figure 4, the semi-circle strategy effectively follows the light gradient. When avoiding the light, successive backward-left actions gradually adjust carbot's heading away from the light. Similarly when seeking the light, successive forward-left actions have the opposite effect.

This semi-circle strategy is successful even though carbot's position at the completion of the avoid phase is still relatively near the light. Because the reinforcement task requires carbot to achieve certain minimum and maximum light readings, heading becomes more relevant than position in solving the task. Suppose that instead of using a backward-left semi-circle for avoiding the light, carbot backed straight away from the light. This would result in positioning carbot in the top, right corner of the environment. Although this position is as far from the light as is possible, carbot's heading would remain towards the light, causing the light readings to be well above the minimum required for achieving the avoid goal. In addition, since the two opposing goals of seek and avoid are always presented periodically, the two strategies must work well in succession (as the two semi-circles do).

Rather than employing alternating semi-circles, the other four CRBP networks trained under the *control* conditions developed a one-point turn strategy, an example of which is shown in Figure 5. Like the semi-circle strategy, the one-point turn strategy effectively follows the light gradient and employs a distinct set of actions for each goal (backward-left and forward-right for avoiding; backward-right and forward for seeking). The qualitative behavior of the GA controllers trained under the *control* conditions was quite different from that of the CRBP controllers. In GA-trained controllers, the light gradient was not as faithfully followed and typically the same set of actions was used to achieve both goals (i.e. more global solutions).

Only six of the ten GA runs trained under the *control* conditions reached the learning criterion. Two of the remaining four runs were quite close to achieving it and were thus included in the analysis; the other two were excluded. Of the eight viable controllers produced, half developed a two-point turn strategy as shown in Figure 6. Here we see that the GA has discovered a single set of actions that is able to accomplish both goals, rather than a unique

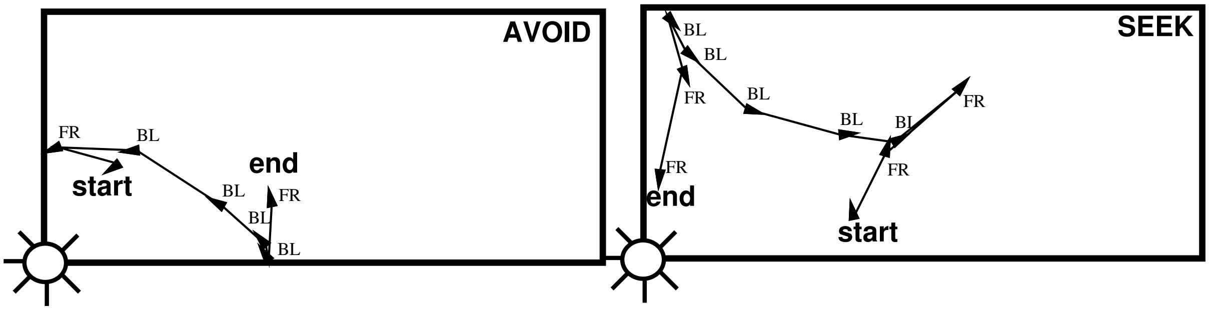

Figure 6: Two-point turn strategy: Executes a short forward-right series, then a longer backward-left series, and finally ends with another short forward-right series for both goals. This strategy was found by 50% of the GA *controls*, 20% of the GA *nogoals*, and 37.5% of the GA *delays*.

response to each goal. The switch in turning directions for this strategy was usually triggered by a wall, since the environment is so small, but in some cases the switch occurred without this environmental cue, as is seen at the start of the seek phase in Figure 6. Notice in the initial steps of this strategy the light gradient is actually ignored—to seek the light the controllers begin by moving carbot further away from the light, decreasing the light sensor readings—but once the backward-left portion of the strategy is entered, the sum of light readings steadily increases as carbot's heading gradually turns more towards the light.

Three of the remaining eight GA-trained controllers developed a many-pointed turn pattern of behavior shown in Figure 7. This strategy allows the robot to remain in the vicinity of the light while its heading is systematically adjusted in the appropriate direction. Like the two-point turn behavior, this star pattern uses one set of actions to accomplish both goals. Only the last GA-trained network developed a strategy tuned to each goal: the semi-circle behavior discussed previously.

Because the GA-trained controllers typically employed a single strategy regardless of the goal, they were less successful at following the light gradient correctly and received significantly more punishment during testing than CRBP-trained controllers (33.29% versus 24.18% [$p < .05$]). However, the GA-trained controllers performed as well as the CRBP-trained controllers in the generalization testing (13.61 steps versus 13.35 steps). Thus the simpler global strategies developed by the GA were as robust as the more finely tuned local strategies developed by CRBP. From these results we should expect that the GA's performance in the next set of experiments, where the goal is removed, will not diminish as much as CRBP's performance.

### *D. Nogoals*

For this set of experiments, the explicit goal was removed. Since the controller has no indication of which of the two phases (either seek or avoid) it should be in at any particular time, the optimal strategy must be able to achieve both goals by systematically moving towards and then away from the light. During learning the networks are provided with reinforcement about their actions relative to the unknown goal, but during testing this reinforcement is no

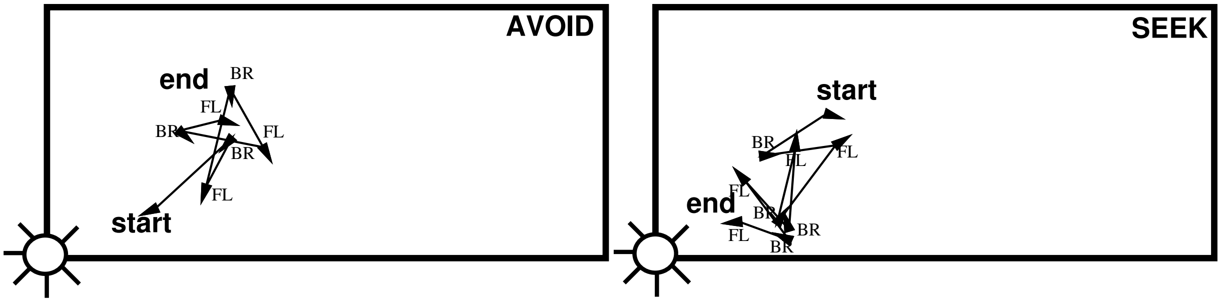

Figure 7: Many-point turn strategy: Executes alternating backward-right and forward-left actions for both goals. The resulting pattern sometimes looks like a star. This strategy was found by 37.5% of the GA *controls* and 60% of the GA *nogoals*.

longer present. Why should we expect that controllers without goals could even succeed at the light task during testing? The recurrent memory allows the network to maintain information about which phase of the task it is in, enabling it to complete an entire seek or avoid phase before switching to the next phase and thus succeed at this hard task.

Under the *nogoal* condition, only one of the ten CRBP-trained networks was able to reach the learning criterion. However all the of the other nine networks converged to reasonable levels of performance, so all ten networks were included in the analysis. For the majority of these controllers, there were no clearly evident strategies being used. During testing the CRBP-trained *nogoal* controllers were punished 49.04% of the time and on average required 26.95 steps to reach the goals. Both performance measures have doubled with respect to the CRBP *control* condition, and this difference is statistically significant [$p < .01$]. The local learning method seems to need the explicit goals to develop distinct patterns of behavior.

Perhaps the most telling evidence that the CRBP-trained networks benefit from explicit goals is that in three of the ten CRBP *nogoal* controllers, the learning actually produced hidden units that served a goal-like function. These three networks performed better than the remaining seven, receiving less punishment (45.85% versus 52.23%) and requiring many fewer steps on average to achieve goals (20.21 versus 29.84). Figure 8 shows the activation of one such goal-like hidden unit during the course of 300 actions along with the sum of carbot's light readings. The sum of the light readings can range from 0 to 2 while the activation of the hidden unit can only range from 0 to 1. To achieve the seek goal the sum of the light readings must exceed 1.7, and to achieve the avoid goal the sum must fall below 0.6. The hidden unit activation remains at the maximum level of 1 most of the time, and then just as a seek goal is being achieved the level drops dramatically for a few steps. This goal-like unit is not perfect though; it is falsely triggered several times (around cycle 175 and 225).

In the *control* experiments, the goal unit's activation in the input layer automatically switched sign to mark the achievement of a goal. In these *nogoal* experiments, this external cue was removed, and in response, some of the CRBP-trained *nogoal* networks developed their own internal cue for marking the achievement of seek goals.

Somewhat surprisingly, the GA's performance was also quite affected by the removal of the explicit goals. None of the ten runs reached the learning criterion, but all exhibited clear strategies. Six of the ten GA-trained *nogoal* controllers developed the many point turn

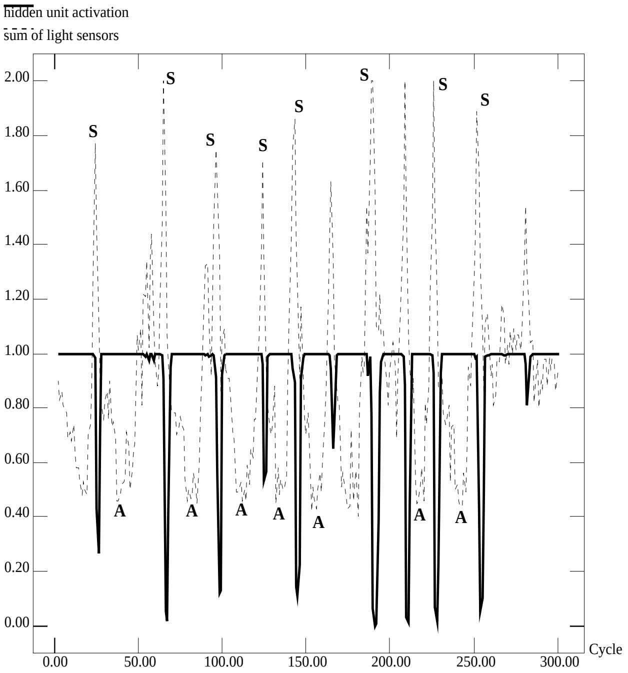

Figure 8: Development of a goal-like hidden unit when no explicit goal was present. An **S** indicates when a seek goal was accomplished and an **A** indicates when an avoid goal was accomplished.

strategy again. Another two developed the two-point turn strategy again. The final two discovered a slight modification to the many-point turn behavior, shown in Figure 9. Here the avoid pattern is exactly the same as before, but the seek pattern has an additional feature. First a number of alternating turns are used to orient the robot towards the light. Then a partial semi-circle is used to approach the light. In terms of the performance measures, the GA *nogoal* networks were significantly worse than the GA *control* networks getting punished 45.52% of the time and needing 24.71 steps on average to achieve the goals [$p < .01$].

In comparing the performance of the GA to CRBP in the *nogoal* condition, there are no significant quantitative differences between the two. However, the examination of the qualitative behavior reveals that without explicit goals CRBP's behavior patterns became much more random while the GA's behavior remained fairly stable. Indeed, the GA was even able to discover a new strategy that combined aspects of two previously successful strategies.

### *E. Delays*

One small change in the CRBP learning procedure was made for this set of experiments where reinforcement about the light was delayed. Because reward was obtained much less frequently (less than 1% of the time even after 750,000 cycles of training), the reward learning rate was increased from 0.3 to 1.0. Despite this change, when the immediate reinforcement about the light gradient was removed, none of the ten CRBP-trained networks reached the

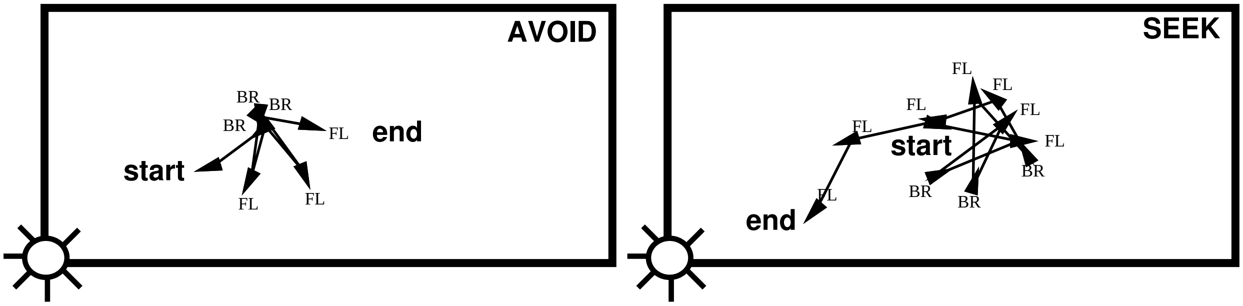

Figure 9: Many-point turn plus semi-circle strategy: Executes alternating backward-right and forward-left actions for both goals, but for seeking the light an additional series of forward-left actions are used to approach the light. This strategy was found by 20% of the GA *nogoals* and 50% of the GA *delays*.

learning criterion. They were punished 66.71% of the time and needed 33.27 steps on average to achieve the goals. Both of these performance levels are significantly worse than CRBP *control* and *nogoal* cases [$p < .01$].

The GA fared much better than CRBP in the *delay* condition. Two of the ten GA-trained *delay* networks reached the learning criterion and six others nearly achieved it (the remaining two were excluded from the analysis). Four of the eight viable networks developed the many-point turn plus semi-circle strategy. Three others settled on the two-point turn strategy. The last used the semi-circle strategy. Once again we see that the GA's qualitative behavior remained quite stable despite large variations in the adaptation conditions. These GA-trained *delay* networks were only punished 36.71% of the time and needed only 14.29 steps on average to reach the goals. These performance marks are significantly better than the GA *nogoal* performance [$p < .01$] and equivalent to the GA *control* performance. In addition, these marks are significantly better than CRBP's *delay* marks [$p < .01$].

## V. DISCUSSION

Figure 3 summarized the mean punishment and generalization scores for all the experimental variations. The statistical analyses revealed several significant quantitative differences between the two learning methods. First, CRBP out-performed the GA in the *control* condition in terms of punishment received. Second the GA out-performed CRBP in the *delay* condition in terms of both punishment received and average number of steps to the goals. Perhaps even more interesting were the qualitative differences in the behaviors produced by the two methods. Being a local method, CRBP is much more sensitive to the moment-to-moment changes in the environment and can thus use the explicit goals to develop unique strategies tuned to each goal. In fact when no explicit goal is present, CRBP-trained networks will sometimes create their own goal-like units in the hidden layer. However this sensitivity to the environment can also be limiting. We saw that in each successive experimental variation, CRBP's performance significantly degraded to the point that it could not succeed with only delayed reinforcement about the light.

In comparison, the GA, as a global method, tends to develop a single overall strategy that is applicable to both goals. More importantly, the GA's ability to find good strategies was quite robust across the experimental variations. But this insensitivity to conditions also has a minor cost: the GA was less attentive to the reinforcement schedule and so received more punishment.

The respective strengths and weaknesses of these two adaptation methods are clearly complementary, suggesting that some hybrid of the two could be the most effective method. Because the GA globally samples the entire space of alternative solutions while CRBP locally searches the immediate neighborhood of a particular solution, the most straight-forward form of hybrid would be to allow the GA to find a good starting point in the weight space and then use CRBP to do the fine-tuning. Belew, McInerney, and Schraudolph did a number of experiments to test the feasibility of using a GA as a source of initial weights for gradient descent learning and found that this technique is effective [4].

To return to the incremental program described in the introduction, combining global and local adaptation methods such as the GA and CRBP is a promising answer to the questions raised about how to properly guide the adaptation process of a network controller. As in nature, the global, evolutionary method can determine a good gross solution which the local learning method can then appropriately adjust to the current environmental conditions. But, there is a caveat: the computational complexity of these hybrids can be extremely high. However, if such hybrid models can produce controllers that are both robust across large environmental changes and yet sensitive to subtle features, then the additional computational effort may be well worth it.

Armed with these insights about how to approach the fourth principle, we can now speculate about the next incremental step towards planning described in the fifth principle. Consider again the basic control architecture shown in Figure 2. As a side effect of learning how to react to the environment and the goals, this network may build up a consolidated record of its past states in the hidden layer. There is no guarantee that this will be the case, but the capacity to do so is available in the recurrent connections. Thus immediately after a goal is achieved, the contents of the hidden layer could conceivably reflect a generalized history of the environmental situations encountered while achieving the goal. Initial analyses of these hidden layer representations show that for the most successful controllers, this kind of history is indeed retained. These observations lead to the following hypothesis: Given a robot controller based on this type of recurrent network, if it is provided with explicit, abstract goals as input and is adapted with a combined global/local reinforcement method, then upon goal achievement, the hidden layer will contain information that can be used to plan for that goal.

This hypothesis has begun to be tested in subsequent work [25]. The hidden layer representations at the time of goal achievement were termed *protoplans*. To investigate whether protoplans could actually help to guide behavior, a transfer of learning experiment was done. The protoplans learned in one controller network were used to guide a second network as it learned the same light seeking and avoiding task from scratch[4]. In this way the strategies discovered in one controller could bias the strategies developed in a new controller so that another agent, rather than a human designer, could direct the learning process.

This transfer of learning experiment was conducted as follows. First, a control network

[4] One difference was that the robot's touch sensors were replaced with sonar sensors allowing the system to anticipate obstacles. This change led to a much more varied repertoire of behaviors.

was trained with reinforcement learning until it was highly successful at the light task, receiving punishment only 11% of the time. After training, this successful network was tested for 1,000 actions. Each time it accomplished a goal, the hidden layer activations (constituting the protoplan) were saved. In addition, the five input states that preceded the goal achievement were saved. These preceding input states provide the cues as to when a particular protoplan is appropriate (when in situation $X1$, $X2$, $X3$, $X4$, or $X5$, do protoplan $Y$). Next an associative memory was constructed that mapped these preceding input states to the ultimate protoplan ($X1 \rightarrow Y$, $X2 \rightarrow Y$, ..., $X5 \rightarrow Y$). Finally a new controller was trained from scratch without access to goals but with access to this protoplan memory. Using its current input state $X'$, this new control network was able to retrieve an appropriate protoplan $Y'$ out of the associative memory built from the original network's solution.

The results of this experiment show that controllers trained with protoplans as inputs, instead of goals, converged more quickly on good solutions than the original controllers with goals. Protoplans were able to guide the robot's behavior by marking the important moments in the interaction with the environment when a switch in behavior should occur. This kind of timing information was indirect—no specific action was indicated—but knowing when to change from a particular strategy to a new one can be very useful information.

In future work, rather than transferring the information contained in a protoplan between controllers, the protoplan should affect the system in which it was created. Specifically, a controller should be able to save and generalize over its own protoplans. In this way, the protoplan memory could be updated on every time step to reflect the system's ever changing summary of the current situation. In addition, this summary could be used to help determine the next action. Ultimately protoplans could serve as communication links between separate modules of a much larger network controller. By being grounded in the environment, protoplans can provide information about the dynamics of the world to higher order modules not directly connected to perception. Furthermore, these higher order modules could create higher order protoplans leading to more complex levels of behavior. Through a series of such incremental steps connectionist controllers may eventually be able reach the level of planning.

## VI. ACKNOWLEDGEMENTS

Thanks to Doug Blank and Gary McGraw for building carbot, to Mike Gasser for commenting on earlier versions of this paper, to Robert Dufour for his help with the statistical analyses, and to the reviewers whose suggestions greatly improved the paper.

## References

[1] D. H. Ackley and M. L. Littman, "Generalization and scaling in reinforcement learning," in *Advances in Neural Information Processing Systems 2* (D. S. Touretsky, ed.), pp. 550–557, Morgan Kaufmann, San Mateo, CA, 1990.

[2] S. Baluja, "Evolution of an artificial neural network based autonomous land vehicle controller," this issue.

[3] R. D. Beer and J. C. Gallagher, “Evolving dynamical neural networks for adaptive behavior,” *Adaptive Behavior*, vol. 1, no. 1, pp. 91–122, 1992.

[4] R. K. Belew, J. McInerney, and N. N. Schraudolph, “Evolving networks: Using genetic algorithms with connectionist learning,” in *Artificial Life II* (C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, eds.), pp. 511–547, Redwood City, CA, 1992, Addison-Welsley.

[5] L. Booker, D. Goldberg, and J. Holland, “Classifier systems and genetic algorithms,” *Artificial Intelligence*, vol. 40, no. 1-3, pp. 235–282, 1989.

[6] V. Braitenberg, *Vehicles: Experiments in Synthetic Psychology.* MIT Press, Cambridge, MA, 1984.

[7] R. A. Brooks, “A robust layered control system for a mobile robot,” *IEEE Journal of Robotics and Automation*, vol. 2, no. 1, pp. 14–23, March 1986.

[8] D. J. Chalmers, “The evolution of learning: An experiment in genetic connectionism,” in *Proceedings of the 1990 Connectionist Summer School*, pp. 81–90, Palo Alto, CA, 1990, Morgan Kaufmann.

[9] A. Clark, *Associative Engines: Conectionism, Concepts, and Representational Change.* MIT Press, Cambridge, MA, 1993.

[10] M. Dorigo and H. Bersini, “A comparison of Q-learning and classifier systems,” in *From animals to animats 3* (D. Cliff, P. Husbands, J. Meyer, and S. Wilson, eds.), pp. 248–255, Cambridge, MA, 1994, MIT Press.

[11] M. Dorigo and U. Schnepf, “Genetics-based machine learning and behavior-based robotics: A new synthesis,” *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 23, no. 1, pp. 141–154, 1993.

[12] J. L. Elman, “Finding structure in time,” *Cognitive Science*, vol. 14, pp. 179–212, 1990.

[13] M. J. Fitzpatrick and J. J. Grefenstette, “Genetic algorithms in noisy environments,” *Machine Learning*, vol. 3, no. 2/3, pp. 101–120, 1988.

[14] D. Floreano and F. Mondada, “Automatic creation of an autonomous agent: Genetic evolution of a neural-network driven robot,” in *From animals to animats 3* (D. Cliff, P. Husbands, J. Meyer, and S. Wilson, eds.), pp. 421–430, Cambridge, MA, 1994, MIT Press.

[15] D. B. Fogel, L. J. Fogel, and P. V. W., “Evolving neural networks,” *Biological Cybernetics*, vol. 63, pp. 487–493, 1990.

[16] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading, MA, 1989.

[17] J. J. Grefenstette, “The evolution of strategies for multiagent environments,” *Adaptive Behavior*, vol. 1, no. 1, pp. 65–90, 1992.

[18] S. A. Harp, T. Samad, and A. Guha, "Towards the genetic synthesis of neural networks," in *Proceedings of the Third International Conference on Genetic Algorithms* (J. D. Schaffer, ed.), pp. 360–369, Palo Alto, CA, 1989, Morgan Kaufmann.

[19] I. Harvey, "Evolutionary robotics and SAGA: The case for hill crawling and tournament selection," in *Artificial Life III* (C. G. Langton, ed.), pp. 299–326, Redwood City, CA, 1993, Addison-Welsley.

[20] I. Harvey, P. Husbands, and D. Cliff, "Issues in evolutionary robotics," in *From animals to animats 2* (J.-A. Meyer, H. Roitblat, and S. Wilson, eds.), pp. 364–373, Cambridge, MA, 1993, MIT Press.

[21] J. Holland, "Escaping brittleness: The possibilities of general purpose learning algorithms applied to parallel rule-based systems," in *Machine Learning: An Artificial Intelligence Approach* (R. Michalski, J. Carbonell, and M. T., eds.), vol. 2, pp. 593–623, Morgan Kaufmann, San Mateo, CA, 1986.

[22] P. Husbands, I. Harvey, and D. Cliff, "Analysing recurrent dynamical networks evolved for robot control," in *Proceedings of the Third IEE International Conference on Artificial Neural Networks*, pp. 158–162. IEE Press, 1993.

[23] F. Martin, "Mini board 2.0 technical reference," MIT Media Lab, Cambridge MA, July 1992.

[24] J. McClelland and D. Rumelhart, eds., *Parallel Distributed Processing*, vol. 1. MIT Press, Cambridge, MA, 1986.

[25] L. A. Meeden, *Towards planning: Incremental investigations into adaptive robot control.* PhD dissertation, Indiana University, 1994.

[26] L. A. Meeden, G. McGraw, and D. Blank, "Emergence of control and planning in an autonomous vehicle," in *Proceedings of the Fifteenth Annual Meeting of the Cognitive Science Society*, pp. 735–740, Hillsdale, NJ, 1993, Lawrence Erlbaum Associates.

[27] D. Montana and L. Davis, "Training feedforward neural networks using genetic algorithms," in *Proceedings of the 1989 International Joint Conference on Artificial Intelligence*, pp. 762–767, 1989.

[28] R. Pfeifer and P. F. Verschure, "Designing efficiently navigating non-goal-directed robots," in *From Animals to Animats 2* (J. Meyer, H. Roitblat, and W. S., eds.), pp. 31–39, Cambridge, MA, 1993, MIT Press.

[29] D. A. Pomerlearu, *Neural Network Perception for Mobile Robot Guidance.* Kluwer Academic Publishing, 1993.

[30] U. Schnepf, "Robot ethology: A proposal for the research into intelligent autonomous systems," in *From Animals to Animats* (J.-A. Meyer and S. W. Wilson, eds.), pp. 465–474, Cambridge, MA, 1991, MIT Press.

[31] D. L. Waltz, "Eight principles for building an intelligent robot," in *From Animals to Animats* (J.-A. Meyer and S. W. Wilson, eds.), pp. 462–464, Cambridge, MA, 1991, MIT Press.

[32] C. Whitley, S. Dominic, R. Das, and C. W. Anderson, "Genetic reinforcement learning for neurocontrol problems," *Machine Learning*, vol. 13, no. 2/3, pp. 259–284, 1993.

[33] S. W. Wilson, "The animat path to AI," in *From Animals to Animats* (J.-A. Meyer and S. W. Wilson, eds.), pp. 15–21, Cambridge, MA, 1991, MIT Press.