Jack of All Trades, Master of None: Evolving Versatility in Modular Connectionist Robots

Ben Mitchell, & BDan Fairchild

May 19, 2003

Abstract

This paper discusses experiments in using Genetic Algorithms (GAs) to search the weight-space of a fixed architecture neural network with modularized output. Attempts were made to evolve a network which could control a mobile robot to complete several different tasks based on a goal input supplied by an outside source (ie. the human operator). This was done to explore the versatility and scalability of both modularized output and task versatility in neural networks. We determined that, while allowing some interesting behavior, the modules did not allow the network to perform multiple tasks easily; those networks evolved to perform a subset of the total task list were able to do a better job on each of their tasks than networks evolved to do any of the tasks upon request.

1 Introduction and Overview

Many previous experiments have used genetic search algorithms in attempts to produce robot control systems. Many of these have used connectionist networks as their basic architectures, and this has proven to be a useful tool for evolving controllers to perform certain tasks. However, in most cases the robots have been asked to perform only a single, relatively simple task.

This is the case for several reasons. For one thing, at this point in time doing either real or simulated work with complex robot controllers in complex environments is both expensive and time consuming. We are limited in our

ability to explore the possibilities of large networks because we do not have the hardware to do so in a reasonable amount of time. In addition to this, the field is quite new, and there has been little time to explore the possibilities. This presents the problem that it is often difficult to determine how scaleable a given approach will be, including our current approach.

Our experiment was designed to push the limits of what has been done so far in this respect. Rather than attempt to perform a single hard task, we decided to see whether we could evolve a controller to perform several simple tasks. We chose tasks which had already been accomplished individually, but attempted to evolve a robot controller which could do any of them, switching dynamically between them during a single lifetime without modifications to its architecture or parameters.

Compared to what has been previously done with neural networks and GAs, this is a fairly hard problem. Therefore, we decided to make the network as flexible as possible without introducing tremendous computing requirements. For this reason, we chose to use a modularized output system similar to the one used by Nolfi [3].

The idea behind the use of modules is that they will allow extremely different outputs from the robot based on inputs which are not as differentiated. This allows much greater granularity and thresholding in the robot's behavior. In Nolfi's original experiment, an agent was evolved to clean up small 'pucks' in an environment by picking them up and placing them outside the environment's walls. The robot had significant difficulty distinguishing between pucks and walls, having only distance sensors, but the agent with modules did a better job than the agent without modules. Analysis showed that the modules were being used to produce a very different behavior when the robot was approaching a puck, allowing it to differentiate between a puck and a wall despite its limited sensor suite.

Our robot has sensors which should allow it to perform this differentiation without complex behavior. However, we are asking it to do highly differentiated tasks based on differences in only three input nodes out of twenty, which is a difficult problem for a neural network to solve. Neural nets are good at generalizing from similar inputs to produce similar behaviors, but they have a much harder time producing drastically different behaviors on the basis of inputs which differ only slightly [1]. The use of modules should allow the robot

to have a greater range of possible behaviors without drastically increasing the number of nodes and connections, and without having to run the GA for more generations, which would be required if we increased the network size. We gave the network three modules with the reasoning that it could then have an independent module for each of the tasks; however, we did not explicitly set this up, and the robot could very well make use of the modules in entirely different ways. The use of the modules was evolved by the GA along with all the other network weights so that if there was a more efficient way of using the modules, the GA would hopefully find it. Our hope was to show that solutions were being evolved which took advantage of the modules to successfully accomplish all three of the tasks upon demand, but we did not expect that the populations would execute all the tasks competently without evolution over a very large number of generations.

In preparing the experiment we were faced with the choice of whether to run it in the real world or in simulation. There are advantages and disadvantages to each approach, but we decided that simulation was a better tool for our particular experiment given our available resources. First of all, there is the consideration of time. Doing things in the real world means doing them in real time, which would be slow given that we have only one suitable robot on which to run our tests, which also would require additional preparation time between them. Simulations can, in theory, be run faster than real time, allowing experiments involving the evolution of populations of hundreds of individuals over hundreds of generations to be run in a more realistic timeframe. In addition to the issue of speed, there is also an issue of the fitness function for the GA. Since the fitness function requires objective information about the state of the agent and its environment at every timestep, information that the robot itself does not possess, it is difficult to implement a real world system which would allow accurate fitness calculations to be performed. Also, it is much more difficult to automatically reset or randomize a real world environment, as doing so generally requires the human-intensive task of picking things up and moving them. In simulation, the environment may be arbitrarily changed with no need for physical mechanisms and control systems to execute those changes.

The main drawback of doing experiments in simulation is that there is always a risk that an agent will not behave the same way in the real world, and therefore the results are less useful in terms of creating actual robots to accomplish real

world tasks. We tried to avoid this problem in several ways. First, we used a simulated robot which was based on a real world robot, so that an evolved control system could be put into an actual robot with a reasonable expectation of good transference. We also used a simulation which was artificially noisy to imitate the noisiness of a real world environment, and force the agent to generalize rather than depending on having exact information about the world. Finally, we constructed a fairly simple environment, which minimizes the chance of a real world environment being different in unpredictable ways.

Much of the work done in the field of evolving robot controllers using genetic algorithms has used them to evolve the weights in artificial neural networks. A great deal of work has also been done using machine learning to determine such weights for similar problems. We decided to use only a genetic algorithm, however. The primary reason for this was that coming up with a good teacher function for our network architecture and our task structure would have been exceedingly difficult without forcing the robot to learn our ideas of good strategy rather than developing its own, due to the complexity of both the inputs and expected behavior. There was the option of combining a GA with learning, as described by Nolfi and Parisi [4], but this would have required vastly more run time to simulate, since each controller would have had to be given time to learn before being evaluated.

As demonstrated by Meeden [5], both learning and GAs can solve problems like the individual tasks we attempted. However, GAs are better suited to tasks in which the agent is rewarded for the achievement of long term goals, without knowledge of the best way to achieve them in the short term. Furthermore, Meeden demonstrated that the addition of explicit goal nodes helped both the learning and the GA to achieve better results. This was even more important in our experiment, in which the order of the goals to achieve was not predictable. Since we wanted to create an agent which could perform tasks based on the desires of some outside agency, those tasks could be given in an arbitrary order, whereas the tasks in Meeden's experiment were given in a predictable pattern. This meant that a controller could learn to predict the change of goals in advance; in fact, this was the goal. In fact, our controllers could not even predict when the task would change, since they lacked recurrence, and thus memory. Additionally, Meeden had only two very similar goals: go toward the light, and go away from the light; similarly, Baldassarre [1] had the two similar goals of

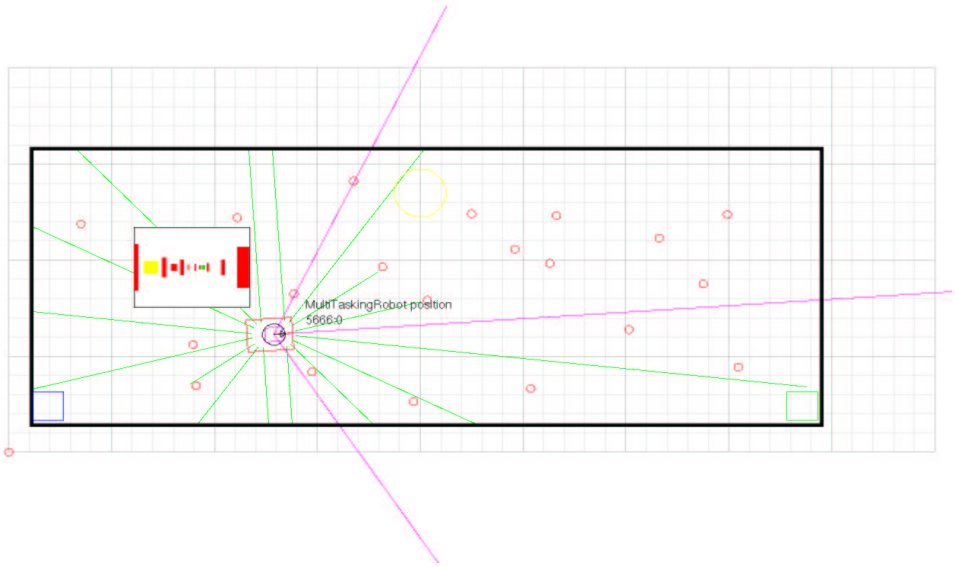
eating food when hungry and drinking water when thirsty, simulated by moving over red and blue colored circles, respectively. We asked our agents to perform three explicit tasks, one of which was very different from the other two, making the agent's job more difficult, as well as an implicit fourth task, that of keeping a simulated battery charged; this was implemented in a fashion similar to Floreano & Mondada [6]. Their original experiment involved only a homing behavior, and used a simple recurrent network, with the weights evolved by a GA. In that experiment, nearly all individuals in the population eventually evolved to take full advantage of the recharging station, charging their batteries precisely when they were about to run out.

2 Methods

We performed all of our experiments in simulation due both the slowness of real time and the difficulty of building a real world environment in which we could evaluate the robot controllers objectively. We constructed our neural networks and our genetic algorithm with the Pyro (Python Robotics) package (`http://emergent.brynmawr.edu/wiki/?Pyro`), which includes an interface to the Player/Stage package (`http://playerstage.sourceforge.net/`) that we used for simulation of robots and environments. The robots were simulated Pioneer IIs, and the environment was a large rectangle with objects in it (see figure 1).

The basic Genetic Search Algorithm was fairly standard, using an initial population with randomized genes and generating mutated offspring of selected individuals. The genomes were arrays of floating point values in the range (-1.0, 1.0), with each number being directly mapped to a single weight in a fixed architecture connectionist network. Each individual was initialized by creating a network using these weights, and was then evaluated by being allowed to interact with an environment for a period of time. The individual was scored on its performance while in the environment.

After all individuals in a population had been evaluated, those with the highest fitness scores (see section 2.4) were allowed to reproduce. Each individual was allowed to produce several offspring with mutated weights; in addition, the best members of the population were allowed to live on in the next generation.



A captured image from the Stage simulator. Green lines represent sonar sensors, magenta lines denote the center and extrema of the camera view, and the box contains the output of the blobfinder.

Figure 1: Environment and robot sensors

2.1 Environment:

The environment consisted of a large rectangle, roughly 7.5 meters by 3 meters, populated with several types of objects (see figure 1). Two of the corners were painted blue and green respectively, and a cylindrical region of space was painted yellow to indicate the battery charging area. (There was no physical object to support the yellow region, so that the robot could move through it. In a real environment, this would be replaced with a light source, or something similar.) The corner areas and the battery charging area were invisible to the sonar sensors. In addition to these areas, the environment was also at times populated with ten small red pucks which the robot could pick up by running over them; these pucks could also be detected by the sonar sensors. The walls (which were also visible to the sonars), floor, and ceiling were all white.

2.2 Network Architecture:

We used a fully connected, three layer feed forward network with emergent modules. The input layer was made up of 20 nodes, divided as follows: 8 nodes for sonar sensor values, 8 nodes for blobfinder sensor values, 1 node for hunger, and 3 nodes for goal description. The robot had 16 sonar sensors, which we condensed by taking pairs of adjacent sensors and using the lesser (i.e. closer) of the two values as the input for each sonar input node. The blobfinder inputs were based on simulations of images taken with the on board camera, which were processed by simple image recognition software to parse the world into areas of different colors. There were four color channels used: red, green, blue, and yellow, for the four different colors of the objects in the environment; the walls, floor, and ceiling, being white, were simply ignored by the blobfinder. The robot was given the X-axis offset (from the left side of the forward-pointing camera view) of the largest blob of each color, and also the percentage of the view taken up by that blob. The hunger node functioned as a simulated battery level sensor. It was set at each timestep to a value inversely proportional to the number of timesteps the robot had left to live before dying due to lack of power. The battery could be recharged, resetting this value (see section 2.1). The goal nodes were used to indicate which of three tasks the robot was being asked to do using a localist representation. All input values were scaled to be between 0.0 and 1.0.

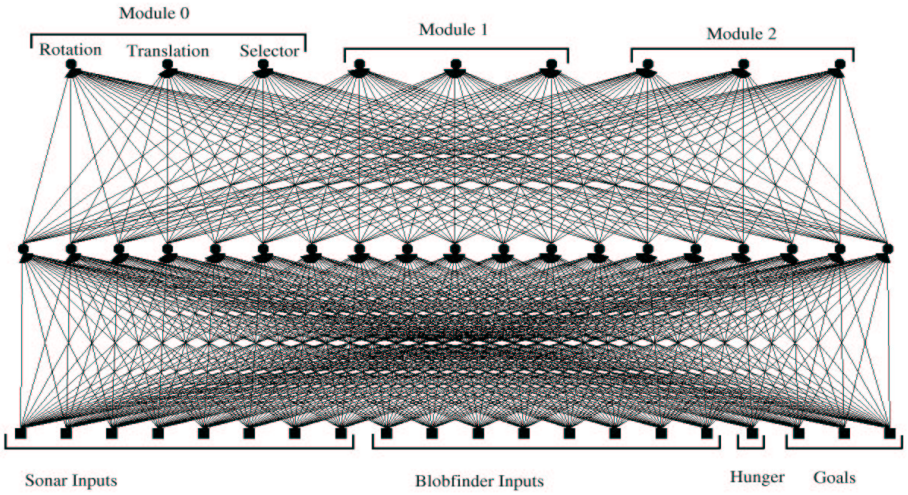


Figure 2: Basic Network Architecture

This input layer was fully connected to a hidden layer made up of 20 nodes, which were fully connected to an output layer. The output layer comprised 9 nodes split into two parts: the modules, and the selectors. There were three separate output models, each of which contained two output nodes. One node corresponded to the robot's speed, and the other corresponded to the robot's rotation; each of these could take on values from -0.5 to 0.5, corresponding to forward and backward travel and left and right turning, respectively. Each output module corresponded to one selector node, which was used to determine which module was allowed to control the robot's movement. At each timestep, whichever selector node had the highest value was considered to have won, and the module associated with that selector was allowed to set the robot's rotation and translation for that timestep. A new module could potentially be selected at every timestep. In total, the network contained 609 connections and 49 nodes; see figure 2.

In a separate run, we used a similar network with one additional output module, but only 16 hidden nodes, with the additional modification that all of the inputs were also connected directly to the selectors; all other connections

were left in place. This network did not perform quite as well as the first one.

2.3 GA parameters:

For the genetic algorithm, we represented the weights of the network as a genome with a floating point number for each of the 609 connections. Values were allowed to vary between -1.0 and 1.0. In each generation, all the members of the population were evaluated, and each individual was assigned a probability of reproducing equal to its own fitness divided by the sum fitness of the population.

$$Probability\ of\ an\ individual\ reproducing = \frac{Fitness\ of\ the\ individual}{Total\ fitness\ of\ all\ individuals}$$

For each individual in the new population, a single old individual from which to inherit the weights was randomly selected based on these probabilities, at which point the weights were randomly mutated. Each value in the genome had a 30% chance of being mutated, where a mutation was an addition of a random value in the range [-1.0, 1.0]. In addition, the top 10% of the population by fitness score were copied into the next generation without mutation, in order to prevent good solutions from being accidentally lost. Once the new population had been created, all individuals were evaluated, and the process began again. This process was iterated for 120 generations.

The fitness function for the GA was based on allowing each individual to control a simulated robot in a simulated environment for a period of time. The simulation lasted for either 400 timesteps, until the robot's battery ran out, or until the robot crashed into a wall and became stuck, whichever came first. Each time a new individual was to be evaluated, the robot was placed at a random location in the middle of the environment (at least 1 meter away from the long walls, and at least 2 meters away from the short walls), and a connectionist network was created using the weights from the individual's genome. The robot's battery initially had 250 timesteps worth of power, and could be instantly recharged by the robot's moving over the battery recharging area. The robot was never given any explicit direction to recharge it's battery, and no fitness was given for recharging the battery. However, since recharging the battery allowed the robot to live longer (and thereby acquire more fitness), it was an implicit goal. Similarly, the robot had an implicit task of avoiding

crashing into walls, as the robot would be killed (with no chance to accumulate additional fitness) if it crashed and stopped moving for ten timesteps.

The three explicit tasks the robot could be asked to perform were to go to the blue corner, to go to the green corner, and to clean up the pucks. At the start of each simulation, one of the three tasks was chosen at random, and the corresponding goal node in the network was set to 1.0. If the goal was puck cleanup, then 10 pucks were randomly scattered around the environment; otherwise, the pucks were not put into the environment at all. The robot was allowed to work on the task until either it's life was ended, it completed the task, or 150 timesteps had elapsed since it started that task, whichever came first. For the tasks which required reaching a corner, the robot was defined to have completed the task when 90% of its visual field was filled with the painted portion of the corner, which could only happen if the robot was very close to the corner and facing directly toward it. For the puck cleanup task, completion was defined as the robot having removed all the pucks in the environment.

At the end of a task, if the robot was still alive it was assigned another task from the list of remaining tasks. If the list was empty, it was reset with all the goals in random order again.

2.4 Fitness Function:

Each task had its own fitness function which would be applied while the robot was being asked to perform that task. The fitness functions were as follows.

Go to the Green (Blue) Corner: The fitness function for this task gave the agent positive fitness at each timestep based on it's forward speed, it's direction with respect to the correctly colored blob, and the change in the subjective size of the blob (which is related to how close it is). Specifically, the function was a product of three factors: the first factor was equal to the speed of the robot if it was positive, and zero if it was negative (s); the second factor ranged from 0.1 to 0.5 depending on the distance of the blob from the horizontal center of the visual field (d); and the third factor was 1.0 if the blob had gotten larger since the previous timestep, and zero if it had not (l: $Fitness = s \times d \times l$

Puck Cleanup: The agent was able to accumulate fitness for moving towards a puck, using an identical version of the function above with the color

changed to red. This was done primarily to bootstrap the agent into moving toward pucks. The agent would also a receive an increase in fitness of 1.0 for actually consuming a puck.

For each of the three tasks, the agent would receive a bonus of 10.0 fitness for completing the task.

The overall fitness function is fairly crude, due to the fact that robots and pucks are placed into the environment at random locations. However, it has been shown that it is better to have a noisy fitness function and be able to run for more generations [7]. Our simulation requires enough processor time that doing several fitness runs for each individual is not computationally feasible, and we determined that the noisiness in the fitness function was preferable to allowing the robot to have a fixed starting point. This was done to force the agents to generalize better.

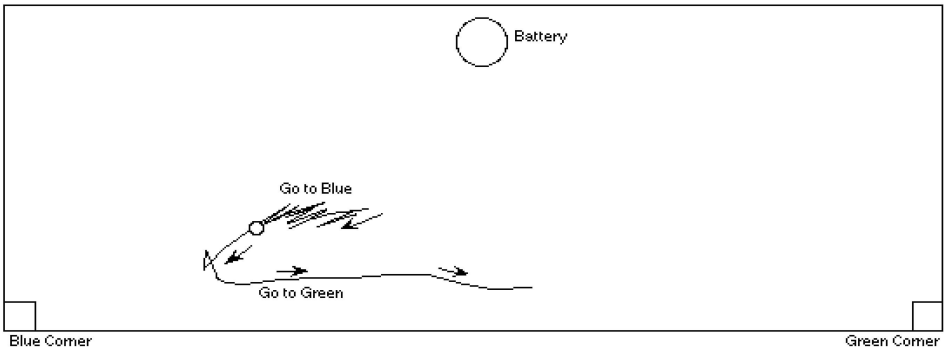
The GA was allowed to run for 120 generations with a population size of 50 for each experimental run, these numbers being chosen on the basis of available computational time. Runs were done in which the robot was asked to do one task in isolation, two tasks, and all three simultaneously.

3 Results

The GA showed a slow but steady increase in average fitness over the course of 120 generations; the average fitness increased by a factor of 115% from the first 10 generations to the last 10. However, the low fitness values showed that the networks were not getting the "bonus" for completing tasks. It is difficult to quantify the progress that was made, due to the difficulty of evaluating the controlers in a non-arbitrary fashion (the fitness function being ultimately arbitrary). We can, however, graph the path taken by a robot during its lifetime, which gives a good general idea of how qualitatively well a controller is performing. See figure 3 for captures of some of the more interesting behaviors in a single controller's life.

4 Discussion

After the same number of generations, the controllers evolved to perform all three tasks had a significantly more difficult time performing any given task



The line shows the path of a robot as it attempted the corner-seeking tasks, with its position measured every five timesteps. In the first part, the goal was to reach the blue corner, and the robot moved quickly back and forth to keep the corner in the center of its visual field. In the second part (after the small circle, the goal was to reach the green corner, and the robot turned around and began traveling towards it, although it died of a low battery before it could get there.

Figure 3: Example Robot Path

than the ones which were evolved to perform only the two similar tasks, or the ones which had to perform only one task in isolation. None of the evolved controllers used the modules to select between behaviors for different tasks, or even for different parts of the same task; instead, the modules seemed to be used primarily to select between different modes of movement, such as moving forward versus moving backward. This allowed the controller to easily switch between types of movement in certain situations. However, the robot could also display significantly different behaviors while using a single module, depending on its situation.

The most useful comparison to made seems to be between the controllers evolved to perform all three tasks, and those evolved only to perform the corner-seeking tasks. After 120 generations, many of the latter developed several useful behaviors, including moving forward toward the appropriate corner, turning from side to side to keep the corner in the center of the visual field, wandering around randomly when the corner was not in sight, and backing up and turning when they were about to run into a wall; other behaviors included circling, two- and three-point turns, and occasionally "go forward till I hit a wall." Most notably, however, many were able to distinguish between which corner they were supposed to be attempting to find. On the other hand, after the same number of generations the three-task "jacks of all trades" only managed to wander around, apparently at random, during all the tasts, and did not even develop very good wall avoidance. This was of some use in the cleanup task, because it allowed the robot to cover a reasonably sized area of floor, but it was of little help in the other tasks, and many better strategies could be imagined even for the cleanup task. There was also little sign of behavior specifically designed to keep the batery charged, indicating that the network had not yet learned the implicit task, either.

For all tasks, we saw modules used for alternating between forward and backward movement; this seemed to be the main use for modules. However, the controllers also made use of modules to turn in different directions, and in the case of the two-task networks, to go in a relatively straight line. Some of the two-task networks used two modules, one of which arced gently to the left and one of which turned more sharply to the right, in alternation when they spotted the correct color. Whenever the color moved to one side or the other of the viewfield, they would switch behaviors, creating a lopsided zig-zag path

toward the appropriate corner. While this behavior was not terribly reliable (there were many individuals which never displayed it due to a combination of differing genomes and differing initial placements in the evironment), it showed that the population as a whole had discovered a strategy which, had it been refined, would have been quite effective.

The three-task networks did not develop an equivalent capability for any of the three tasks after the same number of generations. While it was expected that adding tasks would make the problem harder, it was initially hoped that the addition of modularized outputs to the controller architecture would allow multitasking with a minimal performance penalty. It is possible that, given enough time, the GA would come up with a solution to the three-task problem as well as the two-task problem; however, the three-task networks were much less capable after 120 generations.

While the networks did seem to be taking advantage of the output modules, they did not appear to be using them in a way that facilitated multi-tasking as we had originally hoped. In fact, the pattern of use is very reminiscent of their use in Nolfi's experiments [3]; this may indicate that output modules are only useful for selecting among low-level behaviors, without application towards selecting between high-level behaviors. It is possible that a more complex modular system, perhaps making use of a modularized hidden layer, might be better suited to such an application; this may be an area for future investigation.

Several possible methods spring to mind for improving the controllers' performance. The fitness functions could undoubtedly be improved: the controllers could garner significantly more fitness on the puck-cleaning-up task than on the others, owing to the bonuses from eating pucks. We had thought that this would be offset by the fact that the cleanup task only occurred one third of the time; instead, it might very well be beneficial to multiply the fitness by some constant on the corner-seeking tasks. The environment could also use some work: occasionally, when the controllers trained only on corner-seeking actually reached a corner, they did not receive the bonus because from some angles the colored area was not big enough to take up 90% of the visual field.

Due to the great computational demands of the simulator, we were unable to run the GA with as large a population as would have been ideal. We were also unable to run it for as long as we would have liked. In the future, we would like to allow a long (the equivalent of several months on a high end PC) experimental

run in the hopes of getting more complex, precise behavior. But beyond simply running our experiments for longer, there are several novel experiments which are of interest in relation to this work.

An obvious extension would be the inclusion of a wider range of tasks. This could take the form of either a larger number of tasks, or tasks which were more varied in terms of the required behavior. The initial experiments were done with a small number of simple tasks for the reason that even that had never been done before. It remains to be seen just how scaleable this approach will be, especially given that the controllers from this experiment did not succeed very well at performing the tasks they were given, but it seems important to attempt to continue to push the limits of robot versatility.

Widening the range of tasks would make this an even harder problem than it already is, which would in turn mean that even more run-time would be required to produce a good solution. However, there are several other methods which we considered for improving the adaptability of the population. The most obvious method would be to simply increase the network size, either by increasing the number of hidden nodes, increasing the number of output modules, or both; the resulting increased complexity of the network should permit more complex behavior. However, this would also increase the search space, making it that much harder for the GA to find a good solution quickly [6].

Many other evolved robot controlers [3][6][5] have instead used recurrence in their neural networks. This is usually implemented as a hidden layer which is fully connected to itself in a recursive fashion, allowing the values of the hidden nodes to be influenced by the previous values of those same nodes, a la Elman [2]. This "memory" allows an agent to keep track of what it has done, and also keep better track of where it is in its environment. This addition requires no extra nodes, but it does require extra connections, and therefore connection weights, which means it also increases the genome size. However, it adds novel capability to the network which could not be achieved through the addition of any number of hidden nodes in a strictly feed-forward perceptron. How much this capability would benefit an agent attempting our particular task is unknown, but it would be interesting to find out.

Finally, one way to increase the potential of each individual without increasing the genome size would be to attempt some form of machine learning. There have been many robot controllers which have used some form of neural network

learning [5][1], and even some which combined learning with evolution. By allowing the GA to evolve a teaching function for neural network learning, rather than evolving the weights directly, controllers tend to find better solutions to problems, though the population takes a somewhat larger number of generations to reach its highest fitness [4]. The price, however, is the fact that it takes a great deal more time to allow each individual to learn before it can be evaluated.

As suggested by Meeden [5], since a GA finds global solutions and learning finds local solutions, a combination of the two could plausibly find both, making for an overall more robust agent. In the future, we would like to run experiments similar to this one using each of the different methods described here to determine which are most useful and what are the specific benefits of each for solving the multiple task problem. As we move on to more complex, varied, and challenging tasks, this comparison will help us to decide which methods will best allow us to scale our neural net controllers up to deal with them.

In general, we feel that it is important to do research in the field of developmental robotics aimed at exploring the precise advantages and disadvantages of different techniques in different situations. While many experiments have been done, there is as yet very little basis for making informed decisions as to what sort of network architecture, GA parameters, or specifications in general will be required to solve a given problem. We regret the lack of precision in our own work, both in the reasons for making decisions about implementation details, and in the reporting of results. Due to the arbirtariness of the fitness functions, there is little in the way of quantitative conclusions we can offer other than the bare fact of the network's improvement. Qualitative conclusions are interesting, but are less useful as a resource for doing future work, in that they do not allow formulaic predictions of the outcomes of even simple tests. It is difficult to come to any truly meaningful conclusion, because the current state of the field does not allow it. This is due to many factors, including the youth of the field and the limitations of current computer hardware, but in order for the field to make true progress in any scientific fashion, we will need a much firmer groundwork than currently exists.

References

[1] Gianluca Baldassarre "Needs and motivations as mechanisms of learning and control of behaviour: interference problems with multiple tasks." *Cybernetics and Systems 2000 - Proceedings of the Fifteenth European Meeting on Cybernetics and Systems Research*, pp. 677-682, 2000.

[2] Jeffrey Elman "Finding structure in time." *Cognitive Science*, 14:179-211, 1990.

[3] Stefano Nolfi "Using emergent modularity to develop control systems for mobile robots." *Adaptive Behavior*, 5(3-4):343-364, 1997.

[4] Stefano Nolfi and Domenico Parisi "Auto-teaching: networks that develop their own teaching input." *Proceedings of the Second European Conference on Artificial Life*, pp. 845-862, 1993

[5] Lisa Meeden "An Incremental Approach to Developing Intelligent Neural Network Controllers for Robots." *IEEE Transactions on Systems, Man, and Cybernetics - Part B: Cybernetics*, 26(3):474-485, 1996.

[6] D. Floreano & F. Mondada "Evolution of Homing Navigation in a Real Mobile Robot." *IEEE Transactions on Systems, Man, and Cybernetics - Part B: Cybernetics*, 26(3):396-407, 1996.

[7] M.J. Fitzpatrick & J.J. Grefenstette "Genetic Algorithms in Noisy Environments." *Machine Learning*, 3(2-3):101-120, 1998.