Some Experiments in Evolutionary Synthesis of Robotic Neurocontrollers *

Karthik Balakrishnan and Vasant Honavar
Artificial Intelligence Research Group
Iowa State University, Ames, IA - 50011
balakris@cs.iastate.edu, honavar@cs.iastate.edu

ABSTRACT

Artificial neural networks provide an attractive approach for design of control mechanisms in robots and autonomous agents. However, designing appropriate networks to realize task-specific behaviors is a difficult task. Evolutionary algorithms offer one approach to automating this task. In this paper, we explore a task requiring a robot to clear an arena by pushing boxes off to the sides. We show how evolution can be successfully used to discover high-fitness designs in task environments of differing properties and constraints. An analysis of the evolved networks sheds light on how the resulting designs are tailored to meet the constraints imposed by the environment.

1. Introduction

The observed behavior of robots and autonomous agents in any given environment, is largely a product of the control mechanisms that they are endowed with. Desirable properties of such mechanisms include efficiency, reliability, and robustness in the presence of faults and noise. Artificial neural networks are therefore offer an attractive paradigm for the design of such behavioral control mechanisms [5, 3, 1].

However, designing a good neurocontroller for a given robotic application is not an easy task, since we need to determine the appropriate network architecture (number of units, connectivity pattern, activation functions etc.), and also the the right connection strengths (weights). This design problem further worsens if one is additionally interested in optimizing other criteria like power consumption, space occupied etc. (if the neurocontroller is to be realized in hardware).

Since *Evolutionary Algorithms* (EAs) are potentially useful procedures for searching large, complex, multi-modal, and deceptive search spaces [6, 4], a number of researchers have employed them for searching the space of neurocontroller designs [8, 3, 7, 1]. EAs offer the benefits of a population-based search (hence the possibility of avoiding local optima) and can potentially search the space of network architectures, in addition to the more common search in the space of weights within *a-priori* fixed network architectures.

In this paper we explore a task in which a robot has to clear a square arena by pushing boxes to the enclosing walls. We use EAs to design good neurocontrollers for this robot. As might be expected, the specific design of the neurocontroller is critically dependent on the properties of the robot-environment interaction, and the contraints imposed by it. For example, if the environment provides no feedback when the robot gets stuck (say attempting to move into a wall), then a good neurocontroller design for that environment would be one that avoids moving into walls. Similarly, if the sensors are faulty, a good design would require a sensing strategy which is immune to such faults (for e.g., through multiple sensors located in key directions), etc. In this paper we evolve neurocontrollers for environments with different properties and constraints, and analyze the evolved networks to show how the constraints and properties of the environment are exploited by the networks in order to attain high fitnesses.

* This research was partially supported by the National Science Foundation through grant NSF IRI-9409580 to Vasant Honavar.

2. The Task

The robotic simulation we consider in this paper was proposed by Teller [9]. Here the robot is placed in a square arena of $N \times N$ cells, in which M boxes are randomly scattered in the inner $(N-2) \times (N-2)$ grid. The arena is enclosed by *impenetrable* walls as shown in Figure 1.

The robot is started off in a random position facing one of four directions — north, south, east, or west; with the task of clearing the arena by moving the boxes to the sides. The robot has eight sensors which provide the robot with specific inputs so that it can detect the presence of boxes, walls, or empty spaces. The sensors are *fixed* to the robot (i.e., turn with the robot), and *sense* one square in each of the eight directions surrounding the robot. The robot determines its output action based on the information provided by the sensors, and possibly the history of its past actions.

The robot is capable of three possible actions — forward move, 90 degree anti-clockwise turn, and 90 degree clockwise turn. Turns are in-place, while move forward actions result in the robot moving ahead by one square. If there happens to be a box immediately in front of the robot when it moves forward, the box gets pushed ahead by one square. This is the primitive operation that the robot needs to clear the arena.

The walls are assumed to be non-reflecting, i.e., the robot does not bounce off the wall if it attempts to move into one. Also, the robot has the capacity to push *at most one* box at a time. If the robot tries to move into a wall or push more than one box, the move forward action fails. The fitness of the robot is measured by the number of boxes it pushes to the walls within an allocated period of time. Our objective then, is to design a neurocontroller for this robotic bulldozer that allows it to attain a high fitness. For reasons mentioned in section 1, we use EAs to search for such designs.

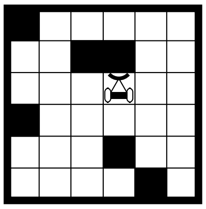

Fig. 1: The operating environment. Shaded squares represent boxes and unshaded ones, empty space.

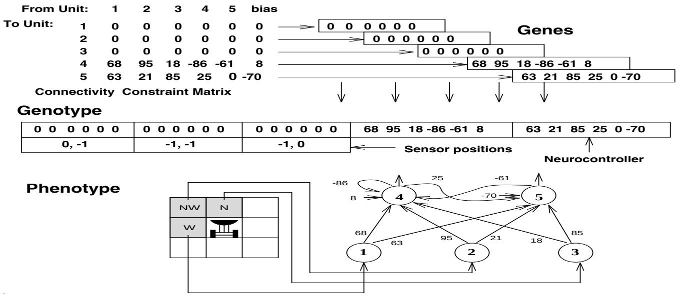

Fig. 2: Genetic representation used.

3. Implementation Details

In our simulations we use *Genetic Algorithms* (GAs) [6, 4] to evolve sensory and neurocontroller designs. Our *genetic representation* had two parts, one encoding the placement and ranges of the sensors, and the other encoding the input connectivities of the units of the neurocontroller, as shown in Figure 2. (For most of the experiments in this paper only the neurocontroller part of the encoding was used. Later work on sensor evolution makes use of the encoding for sensor positions.) The sensors were encoded using a 2-tuple (i, j), with the understanding that the corresponding sensor is placed to detect the square i cells *behind* and j cells to the *right* of the cell occupied by the robot. Thus, negative values of i mean the sensor is *frontal*, i.e., in front of the robot, while negative $j's$ indicate that the sensor is positioned to the *left* of the robot. For example, a value of (-1,0) encodes a sensor that senses the square one cell to the *north* and zero cells to the *east* of the current position of a *north-facing* robot (labeled **N** in Figure 2). This sensor would detect the cell to the east if the robot turns clockwise through 90 degrees. (Henceforth, sensors will be labeled assuming a north-facing robot, with the understanding that the sensors turn with the robot.)

The genetic operator *crossover* was defined to swap *logical blocks* (genes), where each block is the specification of either one sensor or one unit. *Mutation* was defined to randomly change values in the blocks, thereby producing changes in either the placement/range of sensors or in the input connections of a unit (depending on what the corresponding block encoded). Our simulations used populations of size 100, and the evolutionary runs lasted 100 generations. We used *uniform-crossover* with crossover probability of 0.5, and mutation with probability 0.1 per gene. The weights in the network were constrained to be *integers* in the range [-100 $\cdots$ +100], while sensor positions and ranges were bounded by the size of the arena.

Each sensor provides one of three input values to the corresponding input unit of the neurocontroller: 1 for a *box*, 0 for *empty space*, and -1 when it detects a *wall*. At each time-step, the network updates its inputs to reflect the sensor readings and each neuron in the network computes its output (by applying the threshold function to its summed input) and passes it on to each of the neurons to which it is connected. Thus, at each time step, signals are propagated through a depth of one link through the network. The output of the network at each time-step results in an (instantaneous) action by the robot. The neurocontrollers have two output units, one codes for the *action* (move vs turn), while the other determines the *direction of turn*. When the action unit produces a +1, the robot moves forward, while a -1 forces it to turn. On a turn action, the actual direction of turn is given by the output of the turn unit (-1 for clockwise and +1 for anti-clockwise).

The general framework for the simulation experiments was borrowed from [9] which describes Teller's attempts at evolving control programs using *genetic programming*. Thus we used 6×6 arenas, with 6 boxes randomly distributed in the inner 4×4 grid. In each trial, the robot was placed in a randomly chosen empty cell, facing a random direction and allowed to perform $O(N^3) = 216$ actions (where each action takes one time step). For reasons mentioned in [1], Teller's simulation time estimate of $O(N^2) = 80$ was not used.

At the end of 216 time steps each box against the wall was awarded one point, while boxes in the corners fetched one extra point. This was deemed the fitness of the robot in the current trial. The overall fitness of the robot was evaluated by averaging its fitness over 100 random trials, where each trial involved placing it in a random environment and simulating it through 216 time steps as described above. This fitness value was used by evolution in the *fitness-proportionate selection* of parents.

4. Simulation Results

Although we also evolved neurocontrollers of feedforward and recurrent types, we only report results for a third class, labeled *initially feedforward*. Here evolution starts off with feedforward networks, but through low probability mutation, retains the ability to add recurrent links (if need be). We have found this approach to work well in designing networks with few recurrent links [1]. The absence of a link is represented by a zero value at the corresponding weight position (see Figure 2). For the purposes of keeping the analysis simple and the discussion focused, we have chosen to explore networks without any hidden units, although the addition of hidden units is seen to improve performance considerably [1]. In addition, only one kind of sensor was simulated in our experiments.

4.1. No Feedback and No Faulty Sensors

In this set of experiments, we followed the task specification of section 2. Hence the robot has limited pushing capacity, limited sensor range, and obtains absolutely no feedback from the environment. Thus the robot could possibly get stuck forever — attempting to push two (or more) boxes, trying to move into a wall, or simply spinning in place.

Given this understanding, we performed five evolutionary runs, each starting off with a different random seed (hence a different initial random population). The average fitness of the best network evolved in the five runs was 3.74, while the best network ever found had a *labeled fitness* of 5.03.

Since the fitness evaluations during evolution are based on averaging the fitness over 100 random environments, the fitness figures reported by different evolutionary runs are not truly comparable (as the 100 random environments might have been different in each run). Hence we took the best network found in each run, and reevaluated it over a chosen set of 500 environments. This fitness figure, which we call *corrected fitness*, is 4.07 for the best network. This also happens to be the network with the best labeled fitness (5.03), and is shown in Figure 3.

As observed in [1], the evolved neurocontrollers have a large negative self-loop at the output unit coding for action (labeled **A** in the figure). Such an arrangement biases the robots to interleave actions of moving forward and turning, thereby preventing them from getting into states in which they might be stuck till the end of simulation. This feature allows such robots to achieve high fitnesses on this simulation task [1].

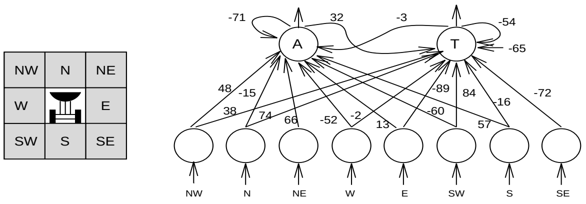

Fig. 3: Evolution of neurocontroller alone. The figure on the left shows the eight active sensors. Also notice the large negative self-loop at the action unit A.

4.2. With a Simple Feedback Mechanism

It is natural to wonder what would happen to the neurocontroller design if the environmental constraints of the preceding section were somehow relaxed. Though one would suspect that the design would be different, how different would it be? In this set of experiments, we introduced a simple feedback mechanism that the robot to could use to detect (and recover) from stuck states, called the *four-time-step countdown*. Here, if the robot spends four *consecutive* time-steps in the same square, the feedback mechanism comes into action and randomly changes the intended action of the robot. Thus, if the robot was stuck trying to move forward, it would be forced to turn now, and vice-versa. Although this strategy is extremely simple, it has interesting consequences on the evolved networks.

Figure 4 shows the best network found by any of the evolutionary runs. This has a labeled fitness 5.63 (corrected fitness 5.23). The average fitness of the best network produced in the five evolutionary runs was 5.09 (as against 3.74 in the previous experiment).

Notice that the network does not have any self-loops. In addition, the network biases the robot to move forward (indicated by the positive threshold value of the action unit **A**, and also the positive weights from six of the eight sensors to the action unit). Indeed the behavior of this robot can be characterized as follows. It moves to locate a box (turning, if the SW or SE sensors detect a box), then it blindly keeps pushing the box until it gets stuck. At this point, either the box is against the wall or against another box. Unlike the robots in the previous experiment, this situation does not spell doom — this robot simply waits for four time-steps and performs a random turn (aided by the feedback mechanism). It repeats this process till the end of simulation.

This behavior (and hence the design of the network) was consistently observed across the different evolutionary runs. As an aside, it was also observed that the three classes of networks — feedforward, recurrent, and initially feedforward, all performed equally well in this environment. This suggests that feedforward networks are sufficient to realize high performing robots in an environment with such a feedback mechanism (Figure 4 being an example of evolution choosing to ignore recurrent links).

The feedback mechanism could be more complex than a four-time-step countdown. Furthermore, once the robot detects it is stuck, it could react in a variety of ways, not necessarily in a random fashion as above. This suggests the possibility of a range of built-in or evolved control strategies that evolution can exploit to produce networks of specific designs in each of the cases.

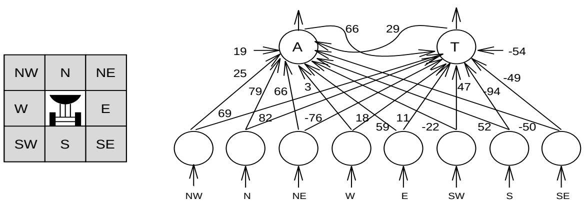

Fig. 4: Environment provides feedback to a stuck robot. If robot spends over four time-steps in a cell, its action is randomly flipped. Notice the absence of any recurrent links whatsoever.

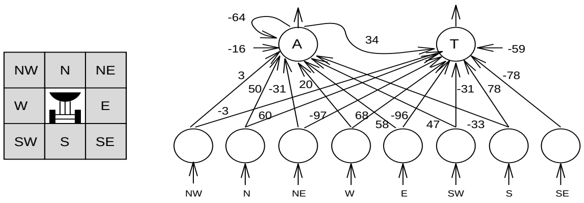

Fig. 5: Environment does not provide any feedback to a stuck robot. The sensors are faulty with probability 20%. Hence at each time-step each sensor (with probability 0.2) either faults (reads 0), or senses the correct value.

4.3. No Feedback but Faulty Sensors

In this study we consider an environment which provides no feedback to a stuck robot, but where the sensors are *inherently faulty*. These faults could be modeled in a variety of ways, for e.g., the sensors could *fail* for the entire simulation run of the robot, or they might simply be *noisy*, sensing wrong values at various time-steps of the simulation run. This study only considers the effect of *noisy* sensors.

Noisy sensors can be modeled in a number of ways. The noise could be *random*, *white Gaussian*, or characterized by a probability distribution that captures the properties of the operating domain. In our simulations, we assume a simple noise model. We assume that *each* sensor has a certain *a-priori* probability of being faulty, where a fault is modeled as the sensor reading a value of 0 instead of what it currently senses. This is tantamount to saying that each sensor (with a pre-specified probability) confuses boxes and walls with empty spaces. Our simulations were performed with a 20% probability of each sensor being faulty. Thus, at each time-step, each sensor *independently* determines (with probability 0.2), whether to provide a 0 or the sensed value to the neurocontroller.

Again, five random evolutionary runs were performed, and the average fitness of the best networks produced in the runs was found to be 6.03. The best network found had a labeled fitness of 6.5 (and corrected fitness 5.97). This network is shown in Figure 5.

Though this network has a negative self-loop at the action unit, it is not very large (particularly considering the offset provided by the threshold value of -16). Considering the fact that the environment does not provide any feedback in case the robot gets stuck, this neurocontroller design should typically result in low a fitness. However, this network attains a very high fitness by exploiting the faulty sensors. For example, suppose the N and E sensors detect a box while the other sensors only detect empty spaces. Then the robot would choose to move forward, pushing the box detected by the N sensor. Now, if for some reason the box cannot be pushed (it is either against a wall or another box), then the robot is effectively doomed to this state for life (since we observe that $50 + 58 > 64 + 16$ and hence the action unit would always produce a 1 making the robot move forward). However, with faulty sensors there is a possibility that one (or both) sensors read 0 (empty space) instead of a 1 (box) when they fail. When this happens, the action unit produces a -1, making the robot turn and allowing it to escape from this doomed state. This exploitation of the sensor faults results in high fitnesses.

However, the fitness of this robot is extremely sensitive to the presence of faults. For e.g., if a robot with the network of Figure 5 is placed in an environment where the sensors do not have faults, the corrected fitness drops from 5.97 to 0.81. In addition, with either an increase or decrease in the fault probability, the fitness of the robot decreases (10%, 30%, 90% fault probabilities resulting in fitnesses of 5.32, 5.79, and 1.83 respectively). Thus this neurocontroller design is seemingly tailored for a 20% sensor fault probability, with both an increase as well as a decrease in the fault probability leading to a drop in performance.

5. Discussion and Conclusions

In this paper we have explored a an approach to evolving neurocontrollers for guiding the behavior of a robot in a relatively simple task environment. Since the environment constrains the design of the controller, we have explored a few different environments and evolved neurocontrollers for robots operating in each of them. By analyzing the evolved networks, we are able to determine how particular environmental pressures lead to specific neural network structures.

In an environment where the robot obtains no feedback from the environment regarding the failure of its actions, evolution produces neurocontroller designs that avoid repeated actions — since repeated actions are what lead to potentially stuck robots. This is achieved through the emergence of a large, negative self-loop at the output unit coding for action, which biases the robot to interleave move and turn actions.

When a primitive feedback mechanism is incorporated in the environment — enabling the robots to determine when they have been stuck in the same position for four consecutive time-steps, evolution produces radically different designs. Highly fit, feedforward networks are sufficient for this domain, as the best behavior seems to be one in which the robot latches onto a box, pushes it all the way to the end, gets stuck, and escapes using the four-time-step feedback mechanism.

When sensors are modeled as faulty (possibly induced by hazardous environments like nuclear waste cleanup, toxic chemical plants, space exploration, etc.), the robots attain very high fitnesses by exploiting the sensor failures to escape from stuck states. The model of sensor faults used in this paper is equivalent to introducing noise into the system, which helps the robots break out of fixed cycles.

Recently we have also been looking at sensor evolution [2] — where the sensory systems of robots are also designed in addition to the usual practice of designing just their controllers. Our results indicate that such approaches lead to extremely efficient, yet highly fit designs. For example, when we evolved the sensory system for the robots of section 4.2, among the best networks discovered by evolution was the one shown in Figure 6. As can be observed, this neurocontroller *only requires two sensors* (N and NE), yet attains a labeled fitness of 6.66 (corrected fitness of 5.96). This network is *smaller* yet *fitter* than the best network found in the experiments of section 4.2 (labeled fitness 5.63, corrected fitness 5.23).

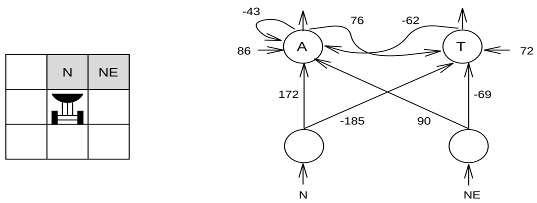

Fig. 6: Environment provides a four-time-step countdown feedback. Sensors are also evolved. Evolution turns off four sensors, while placing the rest in just two sensor positions.

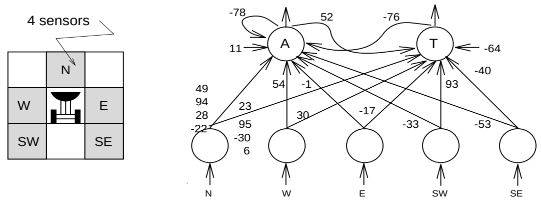

Fig. 7: Environment provides no feedback, sensors are faulty, and sensors are also evolved. Notice the redundant placement of sensors to counter the effects of faulty sensors.

Similar sensor evolution studies performed for robots of section 4.3, also produced highly fit controllers. In particular, the controller of Figure 7 was the best that was found by evolution, with a labeled fitness of 6.69 and a corrected fitness of 6.54. Notice the placement of *four* sensors directly ahead of the robot (at the position marked N). Thus, evolution uses redundancy of sensors in the critical frontal sensor position to help overcome the limitations imposed by sensor faults. This design is radically different from (and more fit than) the one in Figure 5.

Our current research extends the work in this paper, exploring the use of EAs and fitness functions in the design of robust and flexible neurocontrollers under a variety of design, performance, and environmental constraints. Evaluation of evolved controllers embedded in actual robots is in progress.

References

[1] K. Balakrishnan and V. Honavar, "Analysis of neurocontrollers designed by simulated evolution," in *Proceedings of IEEE International Conference on Neural Networks ICNN'96*, 1996. To appear.

[2] K. Balakrishnan and V. Honavar, "On sensor evolution in robotics," in *Proceedings of Genetic Programming Conference – GP-96*, 1996.

[3] 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: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, (D. Cliff, P. Husbands, J.-A. Meyer, and S. Wilson, eds.), (Cambridge, MA), pp. 421–430, MIT Press, 1994.

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

[5] I. Harvey, P. Husbands, and D. Cliff, "Issues in evolutionary robotics," in *From Animals to Animats II: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, (J. Meyer, H. Roitblat, and S. Wilson, eds.), (Cambridge, MA), pp. 364–373, MIT Press-Bradford Books, 1992.

[6] J. Holland, *Adaptation in Natural and Artificial Systems.* Ann Arbor: The University of Michigan Press, 1975.

[7] O. Miglino, K. Nafasi, and C. Taylor, "Selection for wandering behavior in a small robot," *Artificial Life*, vol. 2, no. 1, pp. 101–116, 1994.

[8] S. Nolfi, J. Elman, and D. Parisi, "Learning and evolution in neural networks," *Adaptive Behavior*, vol. 3, no. 1, pp. 5–28, 1994.

[9] A. Teller, "The evolution of mental models," in *Advances in Genetic Programming*, (K. Kinnear, ed.), pp. 199–219, Cambridge, MA: MIT Press, 1994.