Incorporating Advice into Neuroevolution of Adaptive Agents

Chern Han Yong
Institute for Infocomm Research
21 Heng Mui Keng Terrace
Singapore 119613
chyong@i2r.a-star.edu.sg

Kenneth O. Stanley
School of Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL 32816 USA
kstanley@cs.ucf.edu

Risto Miikkulainen
Dept of Computer Sciences
University of Texas, Austin
Austin, TX 78712 USA
risto@cs.utexas.edu

Igor V. Karpov
Dept of Computer Sciences
University of Texas, Austin
Austin, TX 78712 USA
ikarpov@cs.utexas.edu

Abstract

Neuroevolution is a promising learning method in tasks with extremely large state and action spaces and hidden states. Recent advances allow neuroevolution to take place in real time, making it possible to e.g. construct video games with adaptive agents. Often some of the desired behaviors for such agents are known, and it would make sense to prescribe them, rather than requiring evolution to discover them. This paper presents a technique for incorporating human-generated advice in real time into neuroevolution. The advice is given in a formal language and converted to a neural network structure through KBANN. The NEAT neuroevolution method then incorporates the structure into existing networks through evolution of network weights and topology. The method is evaluated in the NERO video game, where it makes learning faster even when the tasks change and novel ways of making use of the advice are required. Such ability to incorporate human knowledge into neuroevolution in real time may prove useful in several interactive adaptive domains in the future.

Introduction

Evolution of neural networks (neuroevolution; NE) is a policy search method for reinforcement learning tasks. The policy is represented as a neural network. Because the correct actions are not known, standard neural network learning methods such as backpropagation cannot be used. However, reinforcement information is available on how effective the actions are, and such information can be utilized as a fitness for the neural network. Based on the fitness evaluations, the networks can then be evolved with e.g. genetic algorithms (GA; (Goldberg, 1989; Mitchell, 1996; Yao, 1999)).

Neuroevolution is particularly effective in control tasks where more traditional value-function-based reinforcement learning methods face difficulties, i.e. those with hidden states and large state-action spaces. Recurrency in the networks serves as memory that allows identifying the state, and the networks generalize well to new states and actions. In several comparisons in standard benchmark tasks (such as balancing two poles simultaneously on a moving cart), neuroevolution has performed significantly better than other reinforcement learning methods (Gomez, 2003; Stanley & Miikkulainen, 2002; Igel, 2003). It has also made novel real-world applications possible, such as controlling finless rockets and mobile robots (Gomez & Miikkulainen, 2003; Floreano & Urzelai, 2000).

In addition to nonlinear control tasks, the approach has proved useful in discovering behavioral strategies for independent agents, such as those that play checkers, robotic soccer, or video games (Chellapilla & Fogel, 1999; Whiteson *et al.*, 2005; Stanley, Bryant, & Miikkulainen, 2005). In such domains, the behavior of agents is usually scripted by hand, which is time consuming and often results in ineffective mechanical behaviors. The behaviors are also difficult to learn because the state-action space is extremely large. With neuroevolution, effective strategies can be discovered automatically, even in real time, as demonstrated e.g. in the NERO video game (Stanley, Bryant, & Miikkulainen, 2005). In NERO, the player acts as a trainer for a team of agents that learn new strategies through neuroevolution.

One problem with the current neuroevolution methods, and traditional reinforcement learning methods as well, is that the agents can learn only through exploration. Often there are behaviors that are obvious for the human observer, yet the agents have to discover them through trial and error. In domains like NERO where the agents interact with humans in real time, exploration slows down learning and makes the agents less believable. More generally, in many cases injecting human knowledge to the learning process would be useful. The human observer might have an idea what works in the domain, formulate it as an advice, and the agents could then take advantage of it in the learning process. Being able to incorporate advice could help in learning difficult tasks, could direct the learning towards the desired kinds of solutions, and could make the human-agent interactions in domains like video games more natural and believable.

This paper shows how advice can be incorporated in the rtNEAT neuroevolution method, and demonstrates that such

advice is useful in interactive domains like NERO. The rtNEAT method is particularly well suited for advice because it evolves both the network weights and its topology (Stanley & Miikkulainen, 2002); using an elaboration of the KBANN technique for converting rules into neural networks (Maclin & Shavlik, 1996), the advice is encoded as a piece of a neural network, which is then included in the rtNEAT networks and refined in further evolution. In NERO, the agents are advised on how to get around an obstacle to a target. Such advice helps evolution discover the desired behavior, and doesn't prevent discovering further behaviors, even those that might conflict with the advice. Therefore, the technique is shown useful in a real-world task, as a mechanism that allows humans to interact with the learning process in real time.

The rtNEAT neuroevolution method is first described below, followed by the NERO video game. Related work on advice taking in artificial intelligence is then discussed, and the method for encoding advice into rtNEAT in the NERO domain is described. Experiments on advising agents on getting around the obstacle are then presented, concluding that the method can be useful in injecting human knowledge into neuroevolution-based reinforcement learning in general.

Real-time Neuroevolution of Augmenting Topologies (rtNEAT)

The rtNEAT method is based on NEAT, a technique for evolving neural networks for complex reinforcement learning tasks. NEAT combines the usual search for the appropriate network weights with *complexification* of the network structure, allowing the behavior of evolved neural networks to become increasingly sophisticated over generations. While NEAT evolves networks off-line, rtNEAT allows the evolutionary process to take place on-line, i.e. in real time during performance. In this section, the main principles of NEAT and rtNEAT are reviewed; for a detailed description, see e.g. (Stanley & Miikkulainen, 2002; Stanley, Bryant, & Miikkulainen, 2005).

NEAT method is based on three key ideas. First, the genetic encoding of the neural network is designed to make it possible to evolve network structure. Each genome includes a list of *connection genes*, each of which specifies two *node genes* being connected, and the weight on that connection. Through mutation, connection weights can change and nodes and connections can be added. To make crossover possible between different network structures, each gene includes an *innovation number*: this number uniquely identifies genes with the same origin, so that they can be matched up during crossover. Second, NEAT speciates the population, so that individuals compete primarily within their own species instead of with the population at large. Thus, topological innovations are protected and have time to optimize their structure before competing with other species in the population. In addition, individuals must share their fitness with their species (Goldberg & Richardson, 1987), preventing any one species from taking over the population. Third, NEAT begins with a population of simple networks (with no hidden nodes), and gradually makes them more complex. In this way, NEAT searches through a minimal number of weight dimensions and finds the appropriate complexity for the problem.

This approach is highly effective: NEAT outperforms traditional reinforcement learning (RL) methods e.g. on the benchmark double pole balancing task (Stanley & Miikkulainen, 2002; Gomez, 2003). In addition, because NEAT starts with simple networks and expands the search space only when beneficial, it is able to find significantly more complex controllers than fixed-topology evolution, as was demonstrated in a robotic strategy-learning domain (Stanley & Miikkulainen, 2004). These properties make NEAT an attractive method for evolving neural networks in complex tasks such as video games.

However, like most genetic algorithm (GA) methods, NEAT was originally designed to run offline. Individuals are evaluated one at a time, and after the whole population has been evaluated, a new population is created to form the next generation. In other words, in a normal GA it is not possible for a human to interact with the multiple evolving agents *while they are evolving*. The rtNEAT method is an extension of NEAT to on-line evolution, where individuals are evaluated and reproduced continuously, making such interaction possible.

In rtNEAT, a single individual is replaced every few game ticks. The worst individual is removed and replaced with a child of parents chosen probabilistically from among the best, and the division into species is adjusted. This cycle of removal, replacement, and speciation continues throughout evolution. The dynamics of offline NEAT is preserved, i.e. the evolution protects innovation and complexifies the solutions as before. Thus, it is now possible to deploy rtNEAT in a real video game and interact with complexifying agents as they evolve.

Neuroevolving Robotic Operatives (NERO)

NERO is representative of a new genre that is only possible through machine learning (Stanley, Bryant, & Miikkulainen, 2005). The idea is to put the player in the role of a *trainer* or a *drill instructor* who teaches a team of agents by designing a curriculum (Figure 1).

In NERO the learning agents are simulated robots, each controlled by a neural network. The goal is to train such a team of robots for military combat. The robots have several types of sensors, including wall rangefinders, enemy radars, an "on target" sensor, and line-of-fire sensors. The radars activate in proportion to how close an enemy is within the radar's sector, and rangefinders activate according to the distance a ray travels before it hits a wall in the rangefinder's direction. The on-target sensor returns full activation only if a ray projected along the front heading of the robot hits an enemy. The line of fire sensors detect where a bullet stream from the closest enemy is heading. The complete sensor set supplies robots with sufficient information to make intelligent tactical decisions.

Typically, 50 such robots are in the field at the same time. The robots begin the game with no skills and only the ability to learn. The player sets up training exercises by placing objects on the field and specifying goals through several sliders. The objects include static enemies, enemy turrets,

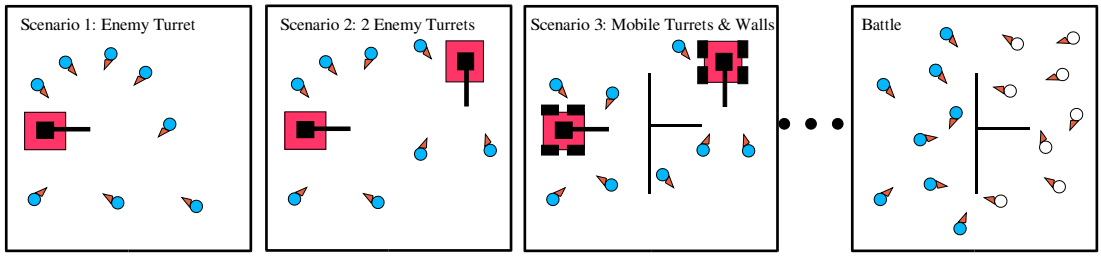


Figure 1: **Training NERO agents for battle.** The figure depicts a sequence of increasingly difficult and complicated training exercises in which the agents attempt to attack turrets without getting hit. In the first exercise there is only a single turret but more turrets are added by the player as the team improves. Eventually walls are added and the turrets are given wheels so they can move. Finally, after the team has mastered the hardest exercise, it is deployed in a real battle against another team. To download a playable demo and animations of various training and battle scenarios, see http://nerogame.org.

rovers (i.e. turrets that move), and walls. To the player, the sliders serve as an interface for describing ideal behavior. To rtNEAT, they represent coefficients for fitness components. For example, the sliders specify how much to reward or punish approaching enemies, hitting targets, getting hit, following friends, dispersing, etc. Fitness is computed as the sum of all these normalized components multiplied by their slider levels. Thus, the player has a natural interface for setting up a training exercise and specifying desired behavior.

Over the course of training, the player sets up increasingly difficult training exercises so that the team can begin by learning a foundation of basic skills and then gradually building on them. Such skills range from attacking an enemy, dodging enemy fire, and navigating a maze, to more complex ones such as approaching an enemy from behind to fire on it and then quickly backing off when it turns around to shoot back. When the player is satisfied that the team is prepared, the team is deployed in a battle against another team trained by another player. The robots are no longer learning and they are eliminated from the game after being shot. The battle ends when one team is completely eliminated, or when the remaining robots will no longer fight. The winner is the team with the most robots left standing.

The challenge of the game for the player is to anticipate the kinds of skills that might be necessary for battle and build training exercises to hone those skills. Behaviors evolve very quickly in NERO, fast enough so that the player can be watching and interacting with the system in real time. However, there are times when evolution seems to take longer than necessary to discover a desired behavior, or when that behavior would be easier to specify verbally rather than through a training regimen. In such cases, it would be useful to be able to incorporate the desired behavior directly into the population. Such human-injected knowledge is very difficult to take into account in evolution in general, but the structure of the rtNEAT method makes it possible, as described in the next section.

Incorporating Advice into Learning

Advice taking has long been recognized as a useful component in artificial intelligence (AI) systems (Huffman, Miller, & Laird, 1993; McCarthy, 1958; Hayes-Roth, Klahr, & Mostow, 1981). Methods have been developed also for integrating advice into systems that learn, either by supervised learning or through reinforcement. For example, Noelle (1997) showed how advice could be entered through separate inputs of a recurrent neural network that then drive the network into an attractor state incorporating the advice. In Gordon and Subramanian's (1994) approach, high-level advice is encoded in rules that are then refined using genetic algorithms. In Maclin and Shavlik's (1996) RATLE system, advice is used to direct reinforcement learning. A neural network is trained to represent the utility function in Q-learning based on examples. At the same time, advice is expressed in a formal language and converted with the KBANN (Knowledge-based artificial neural network) algorithm into nodes and connections added to the neural network.

The method described in this paper is based on Maclin and Shavlik's approach, extended to the rtNEAT neuroevolution learning method. Instead of the utility function, the neural networks represent the controller itself, i.e. a direct mapping from the state to the actions. Instead of backpropagation based on Q-value examples, rtNEAT is used to learn the network weights and the topology. With this interpretation, the advice structures that KBANN produces are exactly like the structural mutations at the core of rtNEAT. The mechanisms for incorporating, evaluating and further refining the advice already exist, making this combination of knowledge-based neural networks and the topology-evolving neuroevolution a natural approach to incorporating advice into neuroevolution.

In the next section, this method will be described in detail, using the NERO video game as the context.

Advising NEAT in NERO

The advice-incorporation method consists of four components: first, the advice is specified according to a formal advice language and converted into a neural-network structure. Second, the advice is added to an agent by splicing the advice structure onto the agent's neural network. Third, the advice is inserted into the population of agents in NERO. Fourth, the advice is refined in rtNEAT evolution in the domain.

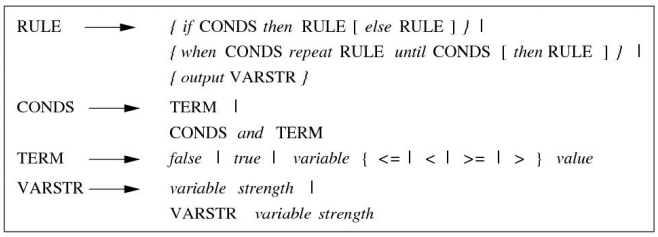


Figure 2: **Advice grammar.** The grammar allows specifying if-then rules and nested loops. The if-then rules are the most useful in the NERO domain and will be evaluated in this paper.

```
{ if wall_ahead > 0.1 then { output move_forward 1.0 turn 0.5 } }
  if wall is some distance in front, then move forward and turn right
{ if wall_45deg_left > 0.5 then { output move_forward 1.0 turn 0.1 } }
  if wall is near 45 degrees to the left, then move forward and turn right slightly
```

Figure 3: **Advice on how to go around a wall.** This sample advice, expressed in the language specified in Figure 2, tells the agent to move forward and to the right, going around the right side of the wall.

Specifying the Advice

The advice language is specified as a grammar, given in Figure 2. There are 3 different types of rules: if-then rules, repeat rules, and output rules. The first two are self-explanatory; the third, output rule, states that an output variable should be either excited or inhibited. A wide range of advice is possible in this language, including state memories (using repeat loops) and nested statements (loops within loops). The experiments reported in this paper focus on the if-then statements because they are the most common type of advice in NERO, and also because their effect can be readily quantified.

For example, if the agents have been trained to go to a flag but have no experience with walls, they will run directly towards the flag even if there is a wall in the way. Through exploration, they will eventually learn to head away from the flag and go around the wall. Although such learning does not take very long in terms of game time — about one minute on average — it still seems like such a simple solution is not worth the time to be discovered automatically, since it is obvious to the human player what the agents should do.

Such knowledge can be easily encoded as advice, as shown in Figure 3. It tells the agent how to go around the right side of a wall by turning right and moving along the wall. The next step is to convert it to the neural network structure.

Converting the Advice into Network Structure

The advice is converted into its equivalent neural-network structure via a recursive algorithm similar to RATLE (Maclin & Shavlik, 1996). This algorithm builds one of three structures depending on whether the rule is an if-then, repeat, or output rule (Figure 4):

For an if-then rule:
1. Create the CONDS structure.
2. Build the rule in the "then" action by calling the conversion algorithm recursively.
3. If there is an "else" clause, create the negation node, and build the rule in the "else" action by calling the conversion algorithm recursively.
4. Connect the nodes with the weight values given in the rule.

For a repeat rule:
1. Create the WHEN-CONDS and UNTIL-CONDS structures.
2. Create the remember-repeat-state and negation nodes.
3. Build the rule in the "repeat" action by calling the conversion algorithm recursively.
4. If there is a "then" clause, create the node for it, and build the rule in the "then" action by calling the algorithm recursively.
5. Connect the nodes with the weight values given in the rule.

For an output rule:
1. Connect the current node to the output units with weight values specified in the rule.

The main difference from RATLE is that learnable recurrent connections are used to maintain the state of the loop in repeat rules, instead of copies of previous activations. Figure 5a shows the sample advice converted into a network structure. This structure is then added to the existing neural network.

Inserting the Advice into Agent Networks

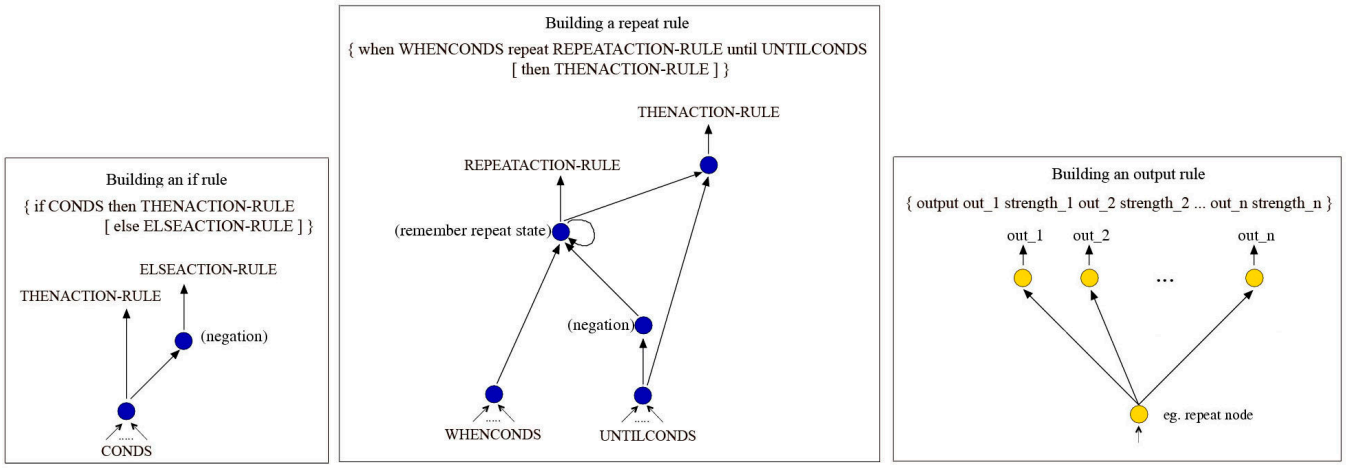
Advice is incorporated into an agent by splicing the advice network structure onto the agent's neural network. This step is similar the KBANN algorithm, except connections with low weights are not included to make the resulting network fully connected; instead, the rtNEAT algorithm adds connections later as needed. Figure 5b shows the original network and Figure 5c the same network with the sample advice added.

In NERO, the advice is added to the best individual in the top k species of the population. If there are fewer than k species, then advice is also added to the lower-ranked individuals in each species until k individuals have received it. Each of these individuals is copied n times, slightly mutating their weights. These kn individuals are then used to replace the worst kn agents in the population. In the experiments in this paper, $k = 5$ and $n = 4$, to give a total of 20 agents with the advice. Slight variations of these values lead to similar results.

Refining the Advice

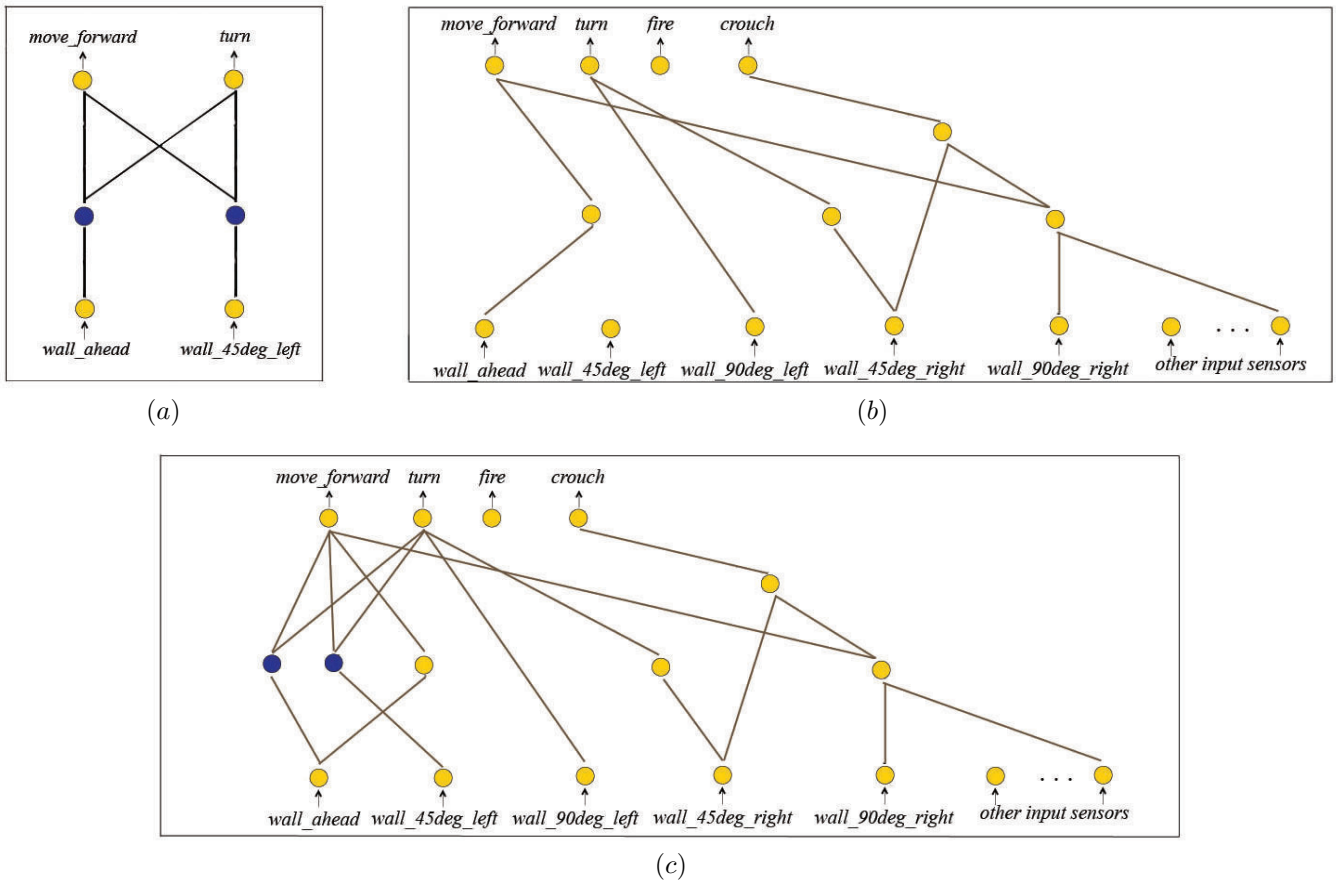
The inserted advice is refined during further evolution by allowing the advice portion of the network to evolve along with the original network. Its weights can be mutated, new links can be added between nodes, and new nodes can be added.

The degree of refinement is controlled by a parameter associated with each advice, representing confidence. This parameter ranges from 0 to 1; a value of x means that the advice portion has a $1-x$ chance of being mutated during each



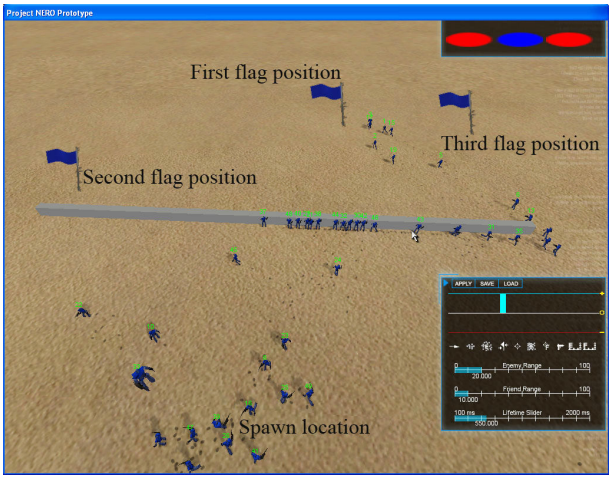
(a) (b) (c)

Figure 4: **Converting advice into neural-network structures.** (a) An if-then rule is constructed by linking the conditions to the "then" action, and their negation to the "else" action. (b) A repeat rule is constructed by creating a neuron that remembers the repeat state through a recurrent connection, and linking the "when" conditions to activate this node and the "until" conditions to deactivate it. (c) An output rule is constructed by linking the rule node to each of the output nodes, weighted as specified in the rule. A blue (i.e. dark) node indicates a node to be added to the network, a yellow (i.e. light) node indicates an existing node in the network.

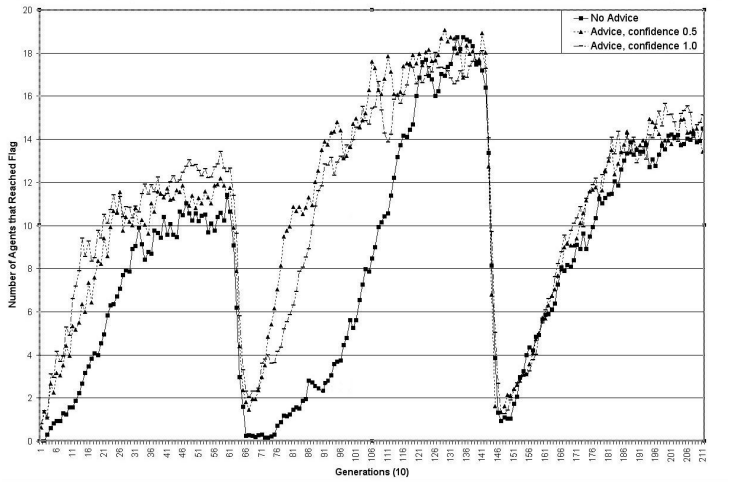


(a) (b) (c)

Figure 5: **Incorporating advice into neural networks in NERO.** (a) The network structure that implements the advice in Figure 3. The thresholds and weights of each hidden unit is set so that they implement the two rules. (b) A network controlling the agents before the advice. (c) The advice is added by connecting the hidden nodes of the advice structure to the appropriate inputs and outputs.



(a) The three phases of the experiment



(b) Performance over generations

Figure 6: **Experiment on advice in NERO.** The agents in the NERO game were first (during generations 0 to 600) trained to go to the flag by going around the right side of the wall. The advised agents learned significantly faster and reached significantly higher performance. The task then changed (during generations 601 to 1400), requiring them to go to the left. Surprisingly, the difference is even greater: it was easier for evolution to discover mutations that changed the bad advice into behaviors that worked in the new situation, than change the behaviors it had discovered without advice. In the third phase (generations 1401 to 2100), the right-path behavior was required again. All systems learned equally: the advice had become encoded the same way as the behaviors discovered without advice. The control window on the bottom right in (a) indicates that the agents are trained only on one objective: going to the flag (indicated by the single cyan (i.e. light) slider). The ovals on top right of (a) indicate that the snapshot is taken during the second (middle) task. The generations in (b) are listed as multiples of ten.

reproduction cycle. When it is 1, the human advice giver expresses full confidence in the advice, and the advice portion is not changed during evolution. When it is 0, the advice evolves just as if it were the original part of the network. In this manner, hard rules can be implemented, but it is also possible to give uncertain advice whose value will be tested in further evolution.

Experiments

Experiments were conducted to test three hypotheses: (1) Giving advice helps the agents learn the task; (2) even if the task changes into one that conflicts with the advice, agents can modify the advice to solve the new task; and (3) the advice eventually becomes incorporated into the network the same way as behaviors discovered through exploration.

These hypotheses were tested in learning to move around a wall to get to a flag that is placed in various locations behind it. The advice used in all these experiments is the sample depicted in Figure 3a. Three versions are compared: learning without advice, and learning with 0.5 and 1.0 confidence on the advice.

The performance of each approach is measured as the number of agents out of the population of 50 who reach the flag at each timestep. The agents' lifetime is 5.5 seconds; depending on the position of the flag, the shortest path to it takes 3 to 5 seconds. Thus each agent has to behave nearly optimally to be successful. To smooth out the performance values, at each time step the windowed average of the performance of the past 30 time steps is taken, and the results are averaged over 3 runs.

Before the experiment, the agents were pretrained to go to the flag, but they did not have any experience with walls. In each run, they start in front of a wall with the flag placed behind it (Figure 6a), and have to learn to circumvent the wall around the right side to reach the flag before their lifetime expires (in about 5 seconds). The evolution continues until a stable, successful behavior has been discovered in all three versions; this happened in 600 generations in all runs.

After the agents have learned to go to the flag, it is moved close behind the wall on the left side (Figure 6a). To reach the flag in their lifetime, agents now have to circumvent the wall on the left side; there is not enough time if they follow the advice and go to the right. The three versions are compared to determine whether evolution can recover from such bad advice and learn the new task. Evolution is run for 800 further generations to achieve the new task.

In the third phase, the flag is moved further back behind the wall to near where it originally was, but slightly to its right (Figure 6b). The agents now have to circumvent the wall on its right side again, i.e. to make use of the old advice. The three versions are compared to see if whether any differences remain, or whether the advice has been incorporated into the network like the behaviors learned by exploration.

Results

Figure 6b plots the average performance of each version over time. The left third of the graph (generations 0 to 600) corresponds to the first phase of the experiment.

Without advice, the agents reach a performance level of 11 at around generation 450. With 0.5 confidence, the performance improves significantly: The agents learn faster and reach a higher level of performance. With 1.0 confidence,

they improve even more, reaching performance 13 at generation 450. These results support the first hypothesis that giving pertinent advice makes learning the task easier.

The middle third of the graph (generations 601 to 1400) corresponds to the second phase where agents have to learn to circumvent the wall around the opposite side. Initially, the agents either try to make a long detour around the right side of the wall, or go straight towards the flag and get stuck behind the wall, resulting in a drastic drop in performance. Gradually, they begin to learn to circumvent the wall around its left. Surprisingly, the agents without the advice were the slowest to learn this behavior, taking over 200 generations longer than the agents with advice. This result was unexpected because the advice is in direct conflict with the necessary behavior. Analysis of the successful networks showed that the same advice portion of the network was still used, but evolution had discovered mutations that turned a right circumvention into a left one. In other words, the original advice was still useful, but the network had discovered a refinement that applied to the changed task. The networks with 0.5 confidence learned faster than those with 1.0 confidence because it was easier for the advice to be modified. Hypothesis 2 was therefore confirmed: evolution can modify advice to fit the new task. Furthermore, if the advice can be exploited to give an advantageous bias in the new task, evolution can modify it to do so.

The right third of the graph (generations 1401 to 2100) corresponds to the third phase of the experiment, where the agents had to relearn to circumvent the wall on the right side, a behavior they had learned in phase one. All three approaches relearned this behavior equally well, whether the agents originally learned it with advice or on their own, showing that the behavior encoded through advice was as well embedded as when it was discovered from scratch. Hypothesis 3 was therefore confirmed as well: advice had become fully integrated into the network.

Conclusion

The experiments presented in this paper demonstrate how human-generated advice can be incorporated in real time in neuroevolution. Such advice makes learning easier, and even when the advice is conflicting, evolution may discover ways to utilize it to its advantage. The structures generated from the advice gradually become incorporated into the network like structures discovered by evolution. The advice technique makes it possible to inject human knowledge into neuroevolution, which should turn out useful in several domains in the future, including video games, adaptive user interfaces, intelligent assistants, and adaptive training environments.

References

Chellapilla, K., and Fogel, D. B. 1999. Evolution, neural networks, games, and intelligence. *Proceedings of the IEEE* 87:1471–1496.

Floreano, D., and Urzelai, J. 2000. Evolutionary robots with on-line self-organization and behavioral fitness. *Neural Networks* 13:431–4434.

Goldberg, D. E., and Richardson, J. 1987. Genetic algorithms with sharing for multimodal function optimization. In Grefenstette, J. J., ed., *Proceedings of the Second International Conference on Genetic Algorithms*, 148–154. San Francisco: Kaufmann.

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

Gomez, F., and Miikkulainen, R. 2003. Active guidance for a finless rocket using neuroevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 2084–2095. San Francisco: Kaufmann.

Gomez, F. 2003. *Robust Non-Linear Control Through Neuroevolution*. Ph.D. Dissertation, Department of Computer Sciences, The University of Texas at Austin.

Gordon, D., and Subramanian, D. 1994. A multistrategy learning scheme for agent knowledge acquisition. *Informatica* 331–346.

Hayes-Roth, F.; Klahr, P.; and Mostow, D. J. 1981. Advice-taking and knowledge refinement: An iterative view of skill acquisition. In Anderson, J., ed., *Cognitive Skills and Their Acquisition*. Hillsdale, NJ: Erlbaum. 231–253.

Huffman, S. B.; Miller, C. S.; and Laird, J. E. 1993. Learning from instruction: A knowledge-level capability within a unified theory of cognition. In *Proceedings of the 15th Annual Conference of the Cognitive Sceince Society*, 114–119. Hillsdale, NJ: Erlbaum.

Igel, C. 2003. Neuroevolution for reinforcement learning using evolution strategies. In *Proceedings of the 2003 Congress on Evolutionary Computation*, 2588–2595.

Maclin, R., and Shavlik, J. 1996. Creating advice-taking reinforcement learners. *Machine Learning* 22:251–281.

McCarthy, J. 1958. Programs with common sense. In *Proceedings of the Symposium on the Mechanization of Thought Processes*, volume I, 77–84.

Mitchell, M. 1996. *An Introduction to Genetic Algorithms*. Cambridge, MA: MIT Press.

Noelle, D. C. 1997. *A Connectionist Model of Instructed Learning*. Ph.D. Dissertation, Departments of Cognitive Science and Computer Science, University of California, San Diego.

Stanley, K. O., and Miikkulainen, R. 2002. Evolving neural networks through augmenting topologies. *Evolutionary Computation* 10:99–127.

Stanley, K. O., and Miikkulainen, R. 2004. Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence Research* 21:63–100.

Stanley, K. O.; Bryant, B.; and Miikkulainen, R. 2005. Real-time neuroevolution in the NERO video game. *IEEE Transactions on Evolutionary Computation* 9:653–668.

Whiteson, S.; Kohl, N.; Miikkulainen, R.; and Stone, P. 2005. Evolving keepaway soccer players through task decomposition. *Machine Learning* 59:5–30.

Yao, X. 1999. Evolving artificial neural networks. *Proceedings of the IEEE* 87(9):1423–1447.