Evolving Stochastic Controller Networks for Intelligent Game Agents

Bobby D. Bryant
Department of Computer Sciences
The University of Texas at Austin
bdbryant@cs.utexas.edu

Risto Miikkulainen
Department of Computer Sciences
The University of Texas at Austin
risto@cs.utexas.edu

***Abstract*— It is sometimes useful to provide intelligent agents with some degree of stochastic behavior, particularly when used in games and simulators. The less-predictable behavior that results from the randomness can make the agents seem more believable, and would encourage the players or users to address the genuine problems presented by a game or simulator rather than simply learning to exploit the embedded agents' predictability. However, such randomized behavior should not harm performance in the agents' designated tasks. This paper introduces a method, called stochastic sharpening, for training artificial neural networks as stochastic controllers for agents in discrete-state environments. Stochastic sharpening reinforces the representation of confidence values in the outputs of networks with localist encodings, and thus produces networks that recommend alternative actions on the basis of their expected utility. Such networks can be used to introduce stochastic behavior with minimal disruption of task performance, resulting in agents that are more believable and less subject to exploitation based on predictability.**

Keywords: Agents, Multi-Agent Systems, Adaptive Team of Agents, Games, Simulators, *Legion II*, Randomness, Stochastic Behavior, Stochastic Sharpening, Neuroevolution

I. Introduction

A very common problem with the behavior of computerized game opponents – the so-called game AI – is that they are brittle because they are predictable. With repeated play, players learn to predict the behavior of the AI and take advantage of it. Such predictability takes much of the fun out of the game. It becomes, as the saying goes, too much like shooting fish in a barrel.

A similar concern arises with simulations that are used for training humans, or as interactive tools for investigating phenomena such as traffic or water management. If the AI-controlled agents in such systems are fully predictable, the human trainees or investigators may simply learn to beat the system rather than solve the intended problem. It is therefore desirable to have AI-controlled agents that behave stochastically, yet still intelligently, in both games and simulators.

Recently, neuroevolution with fitness determined by game play has been found useful in training artificial neural networks as agent controllers in strategy games and simulators [1], [2], [3], [4]. The designs of these systems provide egocentric agents, that is, agents that decide on their own actions based on their own localized knowledge, rather than being moved around as passive game objects by a simulated player. For egocentric agent behavior, an artificial neural network can be used to map the agent's sensory inputs onto a set of controller outputs, which are interpreted by the game engine or simulator as the agent's choice of action for the current time step.

For discrete-state games and simulators that choice of actions can be implemented in an artificial neural network with *action unit coding*. That is, with a localist encoding that associates each of the network's outputs with a choice of one of the possible discrete actions (figure 1). Whenever a decision must be made, the agent's sensory inputs are propagated through the network and the output unit with the highest resulting activation is taken to be that agent's decision for which action to take. This deterministic winner-take-all decoding of the network's outputs results in fully deterministic behavior for the agent.

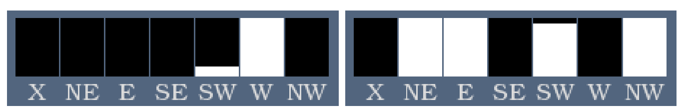

Fig. 1. **Output activations as confidence values.** In these screenshots, a network's output activations are shown graphically, with a vertical white bar showing each neuron's activation value on a scale of $[0, 1]$ and a symbolic code for the action-unit encoded interpretation beneath. *Left:* An output activation pattern plausibly interpretable as confidence values, with a high confidence for output option W, substantially less confidence for option SW, and no discernible level of confidence for any of the other options. *Right:* An output activation pattern only dubiously interpretable as confidence values, since over half the options are fully or near-fully activated. The network that produced these outputs was trained with deterministic winner-take-all decoding, and did not develop activation behaviors suitable for interpretation as confidence values.

The question then arises, can such a system be modified to provide stochastic behavior in an agent, without degrading its performance excessively? A simple and intuitive solution is to interpret the network's outputs stochastically, treating the relative activation levels of the various outputs as confidence values, i.e. indications of how strongly the network evaluates the corresponding action as being appropriate for the current context [5]. If an action is chosen with confidence-weighted decoding, i.e. with probability proportional to the corresponding action unit's activation level, the network will provide stochastic behavior and the stochastic choices will be biased toward the optimal, potentially minimizing any degradation of the agent's performance.

However, it turns out that evolved networks do not automatically learn to represent confidence values in their outputs. When training with the commonly used winner-take-all decoding method described above, there is no penalty for secondary activations at inappropriate action units, so evolution does not learn to suppress them. The network learns only what is useful in solving the problem, and thus its behavior is constrained only by the specific algorithm used to detect the peak output (figure 1). As a result, decoding the network's activations stochastically, as if they were confidence levels, severely degrades the agent's performance (figure 2).

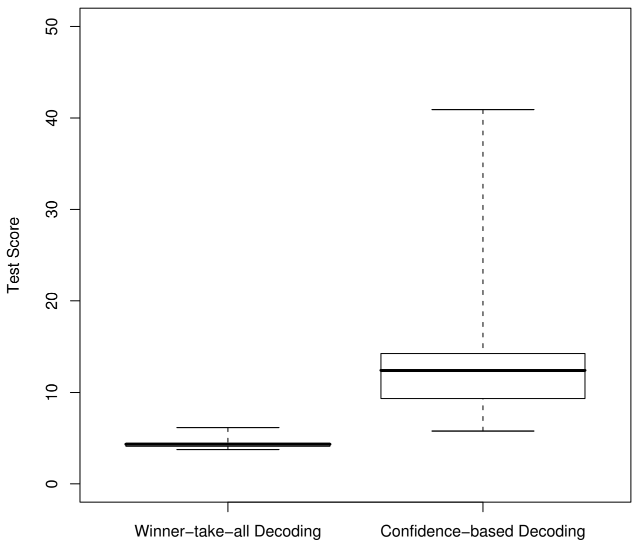

Fig. 2. **Effect of stochastic decoding during testing.** Box-and-whisker plots show the median, quartile, and extremum values obtained by testing 31 networks using deterministic winner-take-all decoding, vs. testing the same networks using confidence-weighted stochastic decoding. (Lower test scores are better.) The stochastic decoding greatly degrades the networks' performance, suggesting that the patterns of output activations do not accurately reflect confidence or expected utility for the various options.

In this paper a method is proposed for evolving networks to produce *utility confidence values* in their outputs. That is, networks learn to produce patterns in their outputs such that the relative activation levels of the various action units reflect the networks' estimates of the relative utility of the associated actions. The training method, called *stochastic sharpening*, discourages spurious output activations by interpreting the network's activation patterns as utility confidence values whenever it is evaluated during training. Networks with inappropriate activations will thus choose inappropriate actions more frequently than others, and consequently perform poorly on the training task. As a result they receive lower fitness scores, and are eventually bred out of the population.

The rest of this paper evaluates the concept of stochastic sharpening experimentally, and examines how sharpened networks can be used to introduce a controlled amount of randomness into the behavior of game agents. Section II introduces the game used as the target problem for testing stochastic sharpening, and describes the underlying neuroevolutionary algorithm. Section III describes the experiments and their results. Section IV evaluates the findings and considers the possibilities for further work.

II. The test environment

Stochastic sharpening was tested in a game/simulator called *Legion II*, which is a slight modification of the *Legion I* game described in [1]. *Legion II* is a discrete-state strategy game designed as a test bed for multi-agent learning problems, with legions controlled by artificial neural networks acting as the intelligent agents in the game.

A. The Legion II game/simulator

The *Legion II* game/simulator is played on a map that represents a province of the Roman empire, complete with several cities and a handful of legions for its garrison (figure 3). Gameplay requires the legions to minimize the pillage inflicted on the province by a steady stream of randomly appearing barbarian warbands. The barbarians collect a small amount of pillage each turn they spend in the open countryside, but a great deal each turn they spend in one of the cities.

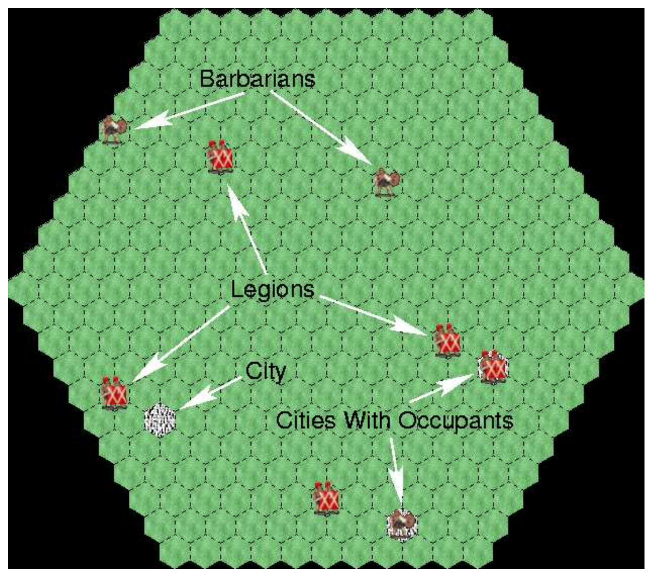

Fig. 3. **The *Legion II* game.** A large hexagonal playing area is tiled with smaller hexagons in order to quantize the positions of the game objects. Legions are shown iconically as close pairs of men ranked behind large rectangular shields, and barbarians as individuals bearing an axe and a smaller round shield. Each icon represents a large body of men, i.e. a legion or a warband. Cities are shown in white, with any occupant superimposed. All non-city hexes are farmland, shown with a mottled pattern. The game is a test bed for multi-agent learning methods, wherein the legions must learn to contest possession of the playing area with the barbarians. (An animation of the *Legion II* game can be viewed at `http://nn.cs.utexas.edu/keyword?ATA`.)

The game is parameterized to provide enough legions to garrison all the cities and have a few left over, which can be used to disperse any warbands they find prowling the countryside. The original purpose of this parameterization was to require the legions to learn an on-line division of labor between garrisoning the cities and patrolling the countryside, in a multi-agent cooperative architecture called an *Adaptive Team of Agents* [1]. The game is used to test stochastic sharpening because it is a challenging learning task where the

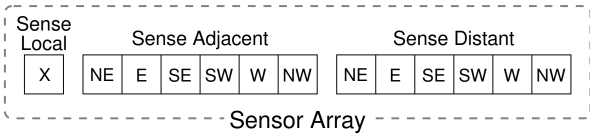

Fig. 4. **A legion's sensor architecture.** Each sensor array for a legion consists of three sub-arrays. A single-element sub-array (left) detects objects colocated in the map cell that the legion occupies. Two six-element sub-arrays detect objects in the six radial fields of view; one only detects adjacent objects, and the other only detects objects farther away. The legions are equipped with three complete sensor arrays with this structure, one each for detecting cities, barbarians, and other legions. The three 13-element arrays are concatenated to serve as a 39-element input layer for an artificial neural network that controls the legion's behavior (figure 5).

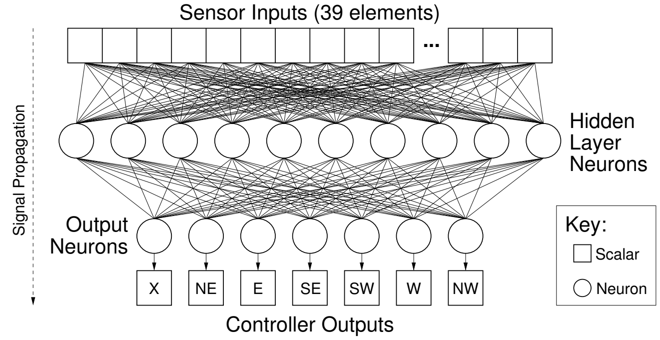

Fig. 5. **A legion's controller network.** During play the values obtained by a legion's sensors are propagated through an artificial neural network to create an activation pattern at the network's output. This pattern is then interpreted as a choice of one of the discrete actions available to the legion. When properly trained, the network serves as the controller for the legion as an intelligent agent.

varying utility of the legions' choices of action can be put to good use.

The *Legion II* map is in the shape of a large hexagon, divided into small hexagonal cells to discretize the placement of game objects such as legions and cities (figure 3). Moves are taken in sequential turns. During a turn each legion makes a move, and then each barbarian makes a move. All moves are atomic, i.e. during a game agent's move it can either elect to remain stationary for that turn or else move into one of the six hexagons of the map tiling adjacent to its current position.

Only one agent, whether legion or barbarian, can occupy any map cell at a time. A legion can bump off a barbarian by moving into its cell as if it were a chess piece; the barbarian is then removed from play. Barbarians cannot bump off legions: they can only hurt the legions by running up the pillage score. Neither legions nor barbarians can move into a cell occupied by one of their own kind, nor can they move off the edge of the map.

A game is started with the legions and cities placed at random positions on the map; the combinatorics allow a vast number of distinct game setups. The barbarians enter play at random unoccupied locations, one per turn. If the roving legions do not eliminate them they will accumulate over time until the map is almost entirely filled with barbarians, costing the province a fortune in goods lost to pillage.

Play continues for 200 turns, with the losses to pillage accumulated from turn to turn. At the end of the game the legions' score is the amount of pillage lost to the barbarians, rescaled to the range $[0, 100]$ so that the worst possible score is 100. Lower scores are better for the legions, because they represent less pillage. The learning method described in this paper allows the legions to learn behaviors that reduce the score to around 4 when tested on a random game setup never seen during training (i.e. to reduce pillage to about 4% of what the province would suffer if they had sat idle for the entire game).

The barbarians are programmed to follow a simple strategy of approaching cities and fleeing legions, with a slight preference for the approaching. The are not very bright, which suits the needs of the game and perhaps approximates the behavior of barbarians keen on pillage.

The legions must be trained to acquire appropriate behaviors. They are provided with sensors that divide the map up into six pie slices centered on their own location. All the relevant objects i in a pie slice are sensed as a single scalar value, calculated as $\sum_i 1/d_i$. This design provides only a fuzzy, alias-prone sense of what is in each sector of the legion's field of view, but it works well as a threat/opportunity indicator: a few barbarians nearby will be seen as a sensory signal similar to what would be seen of a large group of barbarians further away.

There is a separate sensor array for each type of object in play: cities, barbarians, and other legions. There are sensors within each array to provide more detail about what is in the map cells adjacent to the sensing legion, or colocated in the legion's own cell (figure 4). In practice only a city can be in the legion's own cell, but for simplicity the same sensor architecture is used for all three object types.

The scalar sensor values, 39 in all, are fed into a feed-forward neural network with a single hidden layer of ten neurons and an output layer of seven neurons (figure 5). A fixed-value bias unit is also fed into each of the neurons in the network. The size of the hidden layer was chosen by experimentation. The output neurons are associated with the seven possible actions a legion can take in its turn: remain stationary, or move into one of the six adjacent map cells. This localist *action unit coding* is decoded by selecting the action associated with the output neuron that has the highest activation level after the sensor signals have been propagated through the network.

B. Neuroevolution with enforced sub-populations (ESP)

The legions in *Legion II* are trained with the ESP neuroevolutionary algorithm [6], [7], using gameplay for the fitness evaluations. The distinctive feature of ESP is that the "chromosome" representations manipulated by the genetic algorithm represent individual neurons rather than entire networks. Moreover, a separate breeding population is maintained for the position of each neuron in the complete network (figure 6). Breeding is only done within each population,

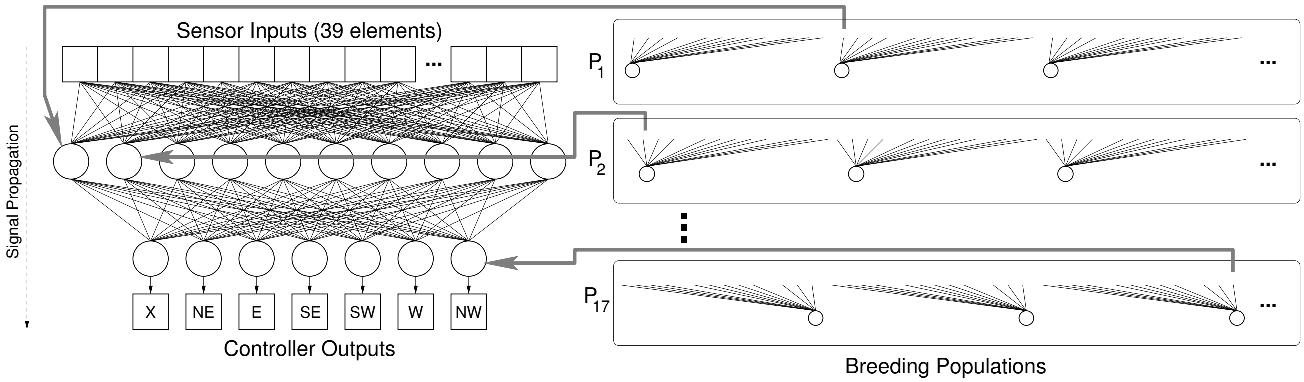

Fig. 6. **Neuroevolution with ESP.** In *neuroevolution with enforced sub-populations* (ESP), a separate breeding population is maintained for each neuron in a network. For the *Legion II* controller network there are 17 such populations, shown in this figure as $\{P_1, P_2, ..., P_{17}\}$. Networks are assembled by drawing one chromosome at random from the population associated with each position in the network and instantiating the neurons the chromosomes represent (gray arrows). The resulting network is tested in the environment and its fitness is ascribed back to each of the chromosomes that specified its neurons. The process is repeated until a fitness score has been determined for every chromosome in all of the sub-populations, at which time an evolutionary generation is complete. Each population is then updated by selective breeding with random crossovers and mutations, independently of the other populations. As generations pass, the sub-populations co-evolve to produce chromosomes describing neurons that work well with the others in the network.

so that each will evolve neurons specific to one position in the network. During training, the populations co-evolve functionality complementary to one another; as a result the algorithm is able to converge quickly on solutions to problems that were formerly considered difficult [7].

For fitness evaluations, one chromosome is drawn at random from each population and the neurons represented by the selected chromosomes are assembled into a network. The network is then evaluated at its assigned task, i.e. as the controller for the legions during one complete game. The fitness value that is measured for the network – the game score – is recorded for each neuron that participated. This scoring method is somewhat noisy because the "real" fitness of a neuron can be brought down by the bad luck of being chosen to participate in a network with other incompetent neurons, or it can be brought up by being chosen for network with superior neurons. To minimize such evaluation noise, each neuron is tested three times in each generation, participating in a network with a different random selection of peers each time. The three scores are then averaged to approximate the neuron's unknown "real" fitness.

Within each sub-population, chromosomes are selected for breeding by a method that favors the ones with the best fitness scores but still allows selection of less-fit chromosomes with low probability. The chromosomes are first sorted from best to worst. Then each is replaced by breeding, starting from the end of the list and working back to the front. An index keeps track of the current position; the position is filled by selecting two chromosomes at random from anywhere on the list ahead of the current neuron and breeding them. I.e., both the parents are at least as good as the one being replaced. The index is then moved forward to point to the next better chromosome. Since replacement is done from worst to best, the better neurons have more more opportunities to be selected than the worse, so selection probabilistically favors the best neurons. However, the probabilities are based on fitness rank rather than on fitness value.

Each neuronal chromosome lists a neuron's input weights as floating point numbers in a flat array. During breeding, 1-point and 2-point crossover are used with equal probability, and point mutations are applied with a low probability at each position in the resulting chromosome. Point mutations add a delta to the weight stored at that position in the chromosome; the deltas are drawn from the exponential distribution so that very small deltas occur with high probability and large deltas occur with low probability. The delta is flipped to be negative with a 50% chance.

ESP has previously been used for training continuous-state controllers for pole balancing and other standard benchmark tasks [8]. It was also effective on an earlier version of the *Legion II* problem, and therefore was used to test stochastic sharpening as well.

III. Experimental evaluation

In evaluating stochastic sharpening experimentally two issues need to be studied: (a) its effect on learning performance, and (b) its effect on interpreting output patterns as utility confidence values. These requirements are covered by training several networks with and without stochastic sharpening and applying appropriate metrics to their performance during testing.

A. Experimental methodology

Fitness scores are obtained by playing the learners against randomly generated game setups; the set of possible game setups is so large that none ever have to be reused. However, for fairness of evaluation every neuron in ESP needs to be evaluated against the same game setup before moving on to the next game. Therefore, the internal state of the random number generator that generates the training games is saved just before generating a new setup, and restored whenever the same game is required again.

Each neuron is evaluated on three different games per generation, and the three resulting fitness scores are averaged. The associations of the neurons into networks are re-randomized before each of the three games so that the averaged fitness scores will reflect the quality of a given neuron *per se* more than the quality of the other neurons it happened to be associated with in the network. Each of the three evaluations uses a different game setup, and all of the neurons are evaluated on the same three game setups during the generation.

Since the training game setups differ continually from generation to generation, learning progresses somewhat noisily: a neuron that performs well on the training games in one generation may not perform well on the new training games of the following generation. Therefore the learning algorithm uses a validation set mechanism to decide what network to deliver at the end of the run, rather than just returning whatever is produced at the end of the final generation of learning. The validation set is simply another set of randomized game setups on which the learner is tested at the end of each generation. It is created independently for each run of the learning algorithm, but once created the same set of games is used each generation throughout the run. A 10-game validation set is deemed large enough to provide sufficient variety of game setups in order to compare fitnesses between generations fairly; excessively large validation sets greatly increase the run time of the learning algorithm and contribute little to the quality of the result.

The combinatorics between the populations of neurons make it infeasible to test every possible network obtainable from the populations at the end of a generation, so a *nominal best network* is defined as the network composed of the highest-fitness neuron from each population, and only the nominal best network is tested. If that network scores better on the validation set than the nominal best network of every preceding generation it is saved as the provisional output of the learning algorithm. At the end of the run, the most recently saved network is returned as the actual output of the algorithm, regardless of which generation produced it.

When testing the trained networks the same test set is used for all the training runs, regardless of training method, so that any differences in the test scores will be the result of variations in training rather than variations in the test set. A separate seed is used to generate the test set games, different from any of the seeds used to generate the training and validation games. For convenience, the progress of learning during a run is examined by testing the learner on the test set at each generation where an improved score is obtained on the validation set. However, to ensure that the learner does not become biased toward the test set, the learning algorithm is not allowed to make any decisions on the basis of the resulting scores; they are merely spilled to a data file for later examination.

Parametric statistical tests such as the Student t-test require sufficiently many samples (i.e. 30) so that the distribution of the t-statistic is approximately normal [9]. Thirty-one independent runs are used in the experiments to satisfy that requirement, plus one extra to give an odd number, so that there is always a clearly defined median performer if ever a

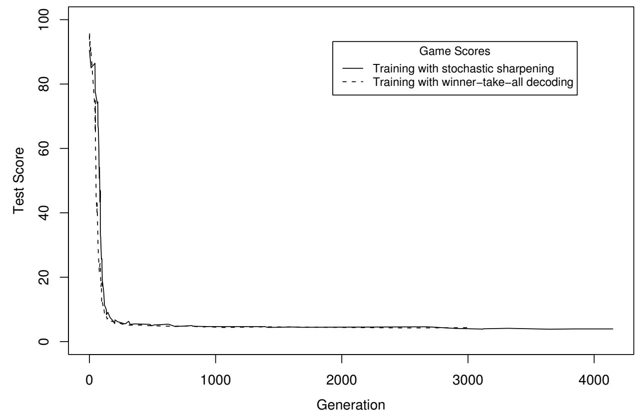

Fig. 7. **Typical learning curves for neuroevolution with ordinary deterministic training and stochastic sharpening.** Learning progress is shown for the median performer of 31 runs for training with deterministic winner-take-all decoding and with stochastic sharpening. The plotted points are the average scores on the test set for the nominal best network at various times during training. Since learning is noisy, points are plotted only for the generations in which progress was made on the validation set, and then connected to make the plot easier to interpret. Other than some noisiness early on, learning with stochastic sharpening progresses similarly to learning with the deterministic winner-take-all method.

single run needs to be singled out as "typical" for plotting or analysis.

The 31 training runs for each learning method need to be independent, so a different training seed is used for each. The stream of random numbers resulting from each choice of seed controls all the non-deterministic learning decisions, such as initializing the values for the input weights of the neurons in the initial populations, generating training and validation game setups during the run, and randomizing crossovers and mutations during the breeding step of each generation. As a result, the 31 runs represent independent random samples from the space of all possible runs of the training algorithm with a given parameterization (i.e., all possible sequences of random decisions during a run), and statistical tests applied to the results of those runs can be used to infer the distribution of results for that universe of possible runs of the algorithm. Due to the very long streams of random numbers required for evolutionary learning, the Mersenne Twister [10] is used to avoid repeats in the streams of generated numbers.

When 31 networks have been trained for each method to be evaluated, a program is run that uses the networks to play the games in the test set and spill various run-time metrics to data files, for analysis and plotting with the R statistical computing environment [11]. Those results are presented in the following sections.

B. Learning with stochastic sharpening

Stochastic sharpening is implemented in neuroevolution by using confidence-weighted decoding during training. That is, at each move during the training games a legion's sensory inputs are propagated through the controller network being evaluated, to obtain a pattern of activations at the seven action-unit outputs. Those activations are normalized so that their

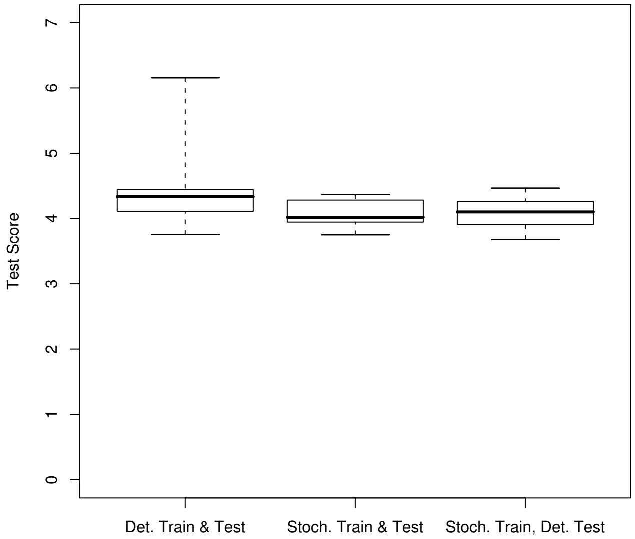

Fig. 8. **Effect of stochastic sharpening on performance.** Box-and-whisker plots show the performance when using deterministic winner-take-all decoding for both training and testing (left) vs. using stochastic sharpening for training and either stochastic or deterministic decoding during testing (center and right, respectively). Stochastic sharpening improved the performance of the resulting networks and reduced the variability of the result as well. It also caused the networks to suppress most secondary activations, so there is little difference in the networks' behavior whether stochastic or deterministic decoding is used during testing (cf. figure 2; note the different scale on the y-axis).

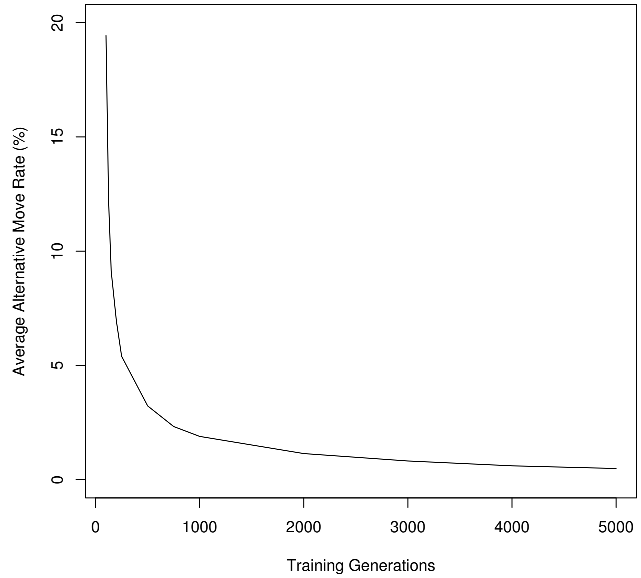

Fig. 9. **Effect of training time on randomness.** A behavioral randomness metric, averaged over the 31 training runs with stochastic sharpening, is plotted against training time. The randomness with confidence-weighted stochastic decoding decreases as training progresses. The effect continues even after task-learning has stalled out (figure 7).

sum is 1.0, and the normalized value of each is treated as the probability that its associated action should be selected for the legion's current move. Thus the behavior of the legion – and ultimately the game score – depends on the *pattern* of activations that the controller network produces, rather than on the peak activation alone. Since a network's evolutionary fitness is derived from the game score, evolution ultimately rewards networks that produce "good" patterns and punishes networks that produce "bad" patterns, where good patterns assign high probabilities to contextually appropriate moves and bad patterns assign high probabilities to contextually inappropriate moves.

When stochastic sharpening is used with neuroevolution the fitness values are initially more random due to the stochastic decoding of the poorly trained networks during evaluations, so learning initially progressed with slightly more variation. However, neuroevolution learns well under noisy fitness evaluations, and in the experiments training with stochastic sharpening rapidly converged onto a learning curve very similar to what was seen for deterministic winner-take-all decoding (figure 7).

The networks produced with stochastic sharpening ultimately converged to a performance that was better by a small but statistically significant amount. The 31 networks trained with deterministic winner-take-all decoding gave a mean score of 4.439 on the test set; those trained with stochastic sharpening gave a mean score of 4.086 when tested with stochastic decoding and 4.092 when tested with deterministic winner-take all decoding (figure 8). In both cases the improvement over the deterministic training method was statistically significant at the 95% confidence value ($p = 0.002$ and $p = 0.003$, respectively). The minimal variety in the performance of the sharpened networks under the two decoding methods suggests that stochastic sharpening greatly suppressed the secondary output activations, so that confidence-weighted stochastic decoding almost always picks the same action that the deterministic decoding does.

Stochastic sharpening also reduced the variance in the performance of the networks by an order of magnitude (figure 8). The performance of the 31 networks trained with deterministic winner-take-all decoding had a variance of 0.378; those trained with stochastic sharpening had a variance of 0.039 when tested with stochastic decoding and 0.051 when tested with deterministic winner-take-all decoding. Reduced variance in the result of a learning algorithm is useful because it increases the probability that a single run will perform near the expected value. For commercial application to difficult problems, a large number of independent runs may not be deemed feasible.

The *randomness* of a network tested with stochastic decoding can be defined as the percentage of the time that an output other than the peak activation is chosen. For networks trained with stochastic sharpening and tested with confidence-weighted decoding, the randomness was continually reduced as training continued, even beyond the point where progress at learning the task had flattened out (figure 9). This fact also suggests that stochastic sharpening suppresses secondary activations in the networks, at least for the current application.

C. Inducing stochastic behavior

As a baseline for comparison, a strawman method was devised for inducing random behavior into the deterministically trained networks. In the strawman method a parameter p specifies the amount of randomness desired, with randomness defined as above. With a $100-p$ percent chance the network's activations are decoded according to the standard deterministic winner-take-all method; otherwise one of the other outputs is chosen instead. Since the secondary activations of the deterministically trained networks are not a useful guide for choosing among them (figure 2), the alternative output is chosen at random, with an equal probability for each option.

Testing on the networks trained with deterministic winner-take-all decoding revealed that task performance degraded at an approximately constant rate as p was increased over the range $0 \leq p \leq 15$. When the same method was used for testing the networks trained with stochastic sharpening the same pattern was seen, though the improved learning obtained by stochastic sharpening (figure 8) provided a constantly better performance for all p (figure 10).

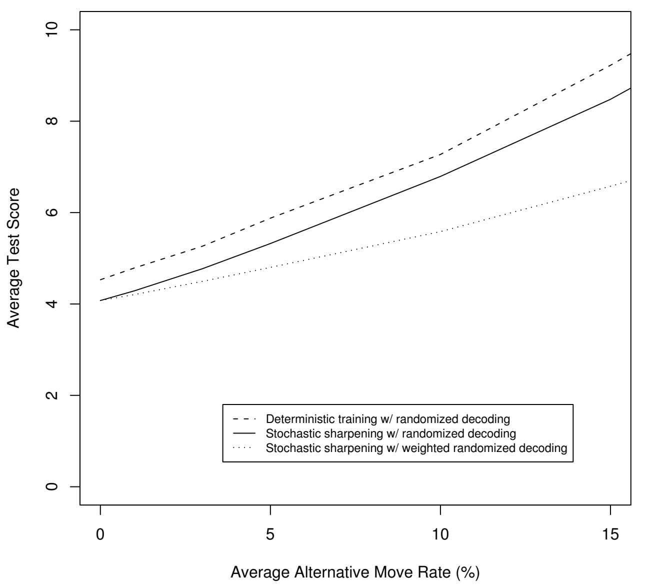

Fig. 10. **Tradeoff between randomness and task performance.** Coerced randomness degrades game scores almost linearly as the amount of randomness increases, regardless of which training method was used (solid and dashed lines). However, for networks trained with stochastic sharpening, if the alternative move is selected by a confidence-weighted choice between the possibilities, the task performance score degrades far more slowly (dotted line).

However, the hypothesis that stochastic sharpening produces networks with useful utility confidence values in their output activations suggests an alternative method of obtaining random behavior. This method works like the strawman method, except that whenever an alternative action must be selected it is chosen based on a confidence-weighted interpretation of the secondary output activations. That is, the activation levels other than the peak are normalized so that they total to 1.0, and then one is selected with probability proportional to the normalized values. When using this method task performance degraded at only about half the rate of the strawman method, as randomness was increased. The method allows significantly more randomness to be introduced into the agents' behavior before their game performance suffers excessively (figure 10). An observer notices the legions making sub-optimal moves slightly more often as the level of induced randomness is increased, but there is no qualitative change in their behavior. (Animations of the randomized behavior can be viewed at `http://nn.cs.utexas.edu/keyword?ATA`.)

IV. Discussion and Future Work

The experiments show that it is possible to introduce stochastic behavior into game agents without degrading their task performance excessively. Moreover, the stochastic sharpening used to train the agents for stochastic behavior provided an absolute improvement in learning performance over the conventional deterministic method. The combination of stochastic sharpening with managed randomization allows agents in the *Legion II* game to choose a non-optimal move about 5% of the time and still perform as well as the fully deterministic agents trained by the conventional method. Greater or lesser amounts of randomness can be obtained by a direct trade-off against the agents' ability to perform their task well (figure 10).

The coerced-randomness method, with stochastic sharpening and confidence-weighted choice among alternative moves, provides the best trade-off options of the methods examined here. It is also flexible, since the randomness parameter is set at run time; no re-training is required for changing the amount of randomness displayed by the agents. Indeed, the amount of randomness can be adjusted during the course of a game or simulation, by simply changing the current value of p.

The improved learning performance for stochastic sharpening was an unexpected side benefit. There are two reasons for this effect. First, when stochastic sharpening is used, even the peak output activations are somewhat low, whereas the deterministic method tends to produce networks that saturate their outputs. Saturated outputs tend to generate a race condition among a network's weights during training, which is generally detrimental to learning. Stochastic sharpening pushes the network's activations back away from the saturation point, and thus avoids the race condition in tuning the weights. Second, the stochastic decoding used during training serves as a self-guiding shaping mechanism [12], [13]. That is, a partially trained network does not have to maximize the correct output in order to perform well; it merely needs to activate it enough to be selected with some probability. A partially trained network that does the right thing *sometimes* may show a higher evolutionary fitness than a network that stakes everything on being able to pick the right choice *always*.

It must be noted that stochastic sharpening does not depend on the choice of neuroevolutionary algorithm; it relies only on the use of action-unit output codings. Thus it should be possible to deploy stochastic sharpening as an add-on to neuroevolutionary methods other than ESP, such as CMA-ES [14] and NEAT [15]. Indeed, it may be applicable even apart from the use of artificial neural networks, so long as the decision engine can represent a choice between discrete options with scalar utility confidence values.

In the future, methods must be developed for inducing stochastic behavior in games and simulators that operate in continuous space and time, such as the NERO machine-learning strategy game [4]. Since controllers for those environments generally select by degree – what angle or how fast – rather than selecting from among distinct options, operations in those environments will provide a different sort of challenge for randomizing behavior intelligently.

V. Conclusions

It is sometimes desirable to introduce randomness into the behavior of AI-controlled agents in games and simulators, in order to increase the challenge to players, learners, and investigators. When those agents are controlled by artificial neural networks, randomized behavior can be induced by means of stochastic interpretation of the networks' outputs. This paper offers a simple method for stochastic decoding in action-unit coded networks, and provides a training mechanism to improve the performance of that method. The mechanism, called stochastic sharpening, allows partially randomizing an agent's behavior with minimal harm to its task performance. Agents with such behavior will make games and simulators more believable, and will challenge players, trainees, and investigators to find solutions to the intrinsic challenges of the games and simulators, rather than merely exploiting the predictability of AI-controlled agents.

Acknowledgments

This research was supported in part by the Digital Media Collaboratory at the IC^2 Institute at the University of Texas at Austin. The images used in *Legion II*'s animated display are derived from graphics supplied with the game *Freeciv*, `http://www.freeciv.org/`.

References

[1] B. D. Bryant and R. Miikkulainen, "Neuroevolution for adaptive teams," in *Proceeedings of the 2003 Congress on Evolutionary Computation (CEC 2003)*, vol. 3. Piscataway, NJ: IEEE, 2003, pp. 2194–2201. [Online]. Available: http://nn.cs.utexas.edu/keyword?bryant:cec03

[2] S. M. Lucas, "Cellz: a simple dynamic game for testing evolutionary algorithms," in *Proceeedings of the 2004 Congress on Evolutionary Computation (CEC 2004)*. Piscataway, NJ: IEEE, 2004, pp. 1007–1014. [Online]. Available: http://algoval.essex.ac.uk/rep/cellz/PortlandCellz.pdf

[3] K. O. Stanley and R. Miikkulainen, "Competitive coevolution through evolutionary complexification," *Journal of Artificial Intelligence Research*, vol. 21, pp. 63–100, 2004.

[4] K. O. Stanley, B. D. Bryant, and R. Miikkulainen, "Real-time neuroevolution in the NERO video game," *IEEE Transactions on Evolutionary Computation Special Issue on Evolutionary Computation and Games*, vol. 9, no. 6, pp. 653–668, 2005. [Online]. Available: http://nn.cs.utexas.edu/keyword?stanley:ieeetec05

[5] S. Renals and N. Morgan, "Connectionist probability estimation in HMM speech recognition," International Computer Science Institute, Berkeley, CA, Tech. Rep. TR-92-081, 1992.

[6] F. Gomez and R. Miikkulainen, "Solving non-Markovian control tasks with neuroevolution," in *Proceedings of the 16th International Joint Conference on Artificial Intelligence*. Denver, CO: Morgan Kaufmann, 1999. [Online]. Available: http://nn.cs.utexas.edu/keyword?gomez:ijcai99

[7] F. Gomez, "Learning robust nonlinear control with neuroevolution," Ph.D. dissertation, Department of Computer Sciences, The University of Texas at Austin, 2003. [Online]. Available: http://nn.cs.utexas.edu/keyword?gomez:phd03

[8] F. Gomez and R. Miikkulainen, "Active guidance for a finless rocket using neuroevolution," in *Proceeedings of the Genetic and Evolutionary Computation Conference (GECCO-2003)*. San Francisco, CA: Morgan Kaufmann, 2003.

[9] R. R. Pagano, *Understanding Statistics in the Behavioral Sciences*, 2nd ed. St. Paul, MN: West Publishing, 1986.

[10] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, 1998. [Online]. Available: http://www.math.keio.ac.jp/~matumoto/emt.html

[11] R Development Core Team, *R: A language and environment for statistical computing*, R Foundation for Statistical Computing, Vienna, Austria, 2004. [Online]. Available: http://www.R-project.org

[12] B. F. Skinner, *The Behavior of Organisms: An Experimental Analysis*. New York: Appleton-Century-Crofts, 1938.

[13] T. S. Peterson, N. E. Owens, and J. L. Carroll, "Towards automatic shaping in robot navigation," in *Proc. of International Conference on Robotics and Automation*, 2001. [Online]. Available: http://citeseer.nj.nec.com/peterson01towards.html

[14] C. Igel, "Neuroevolution for reinforcement learning using evolution strategies," in *Proceeedings of the 2003 Congress on Evolutionary Computation (CEC 2003)*, vol. 4. Piscataway, NJ: IEEE, 2003, pp. 2588–2595. [Online]. Available: http://www.neuroinformatik.ruhr-uni-bochum.de/ini/PEOPLE/igel/NfRLUES.pdf

[15] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002. [Online]. Available: http://nn.cs.utexas.edu/keyword?stanley:ec02