# An Integrated Neuroevolutionary Approach to Reactive Control and High-level Strategy

Nate Kohl and Risto Miikkulainen
Department of Computer Sciences
University of Texas at Austin
nate@cs.utexas.edu, risto@cs.utexas.edu

***Abstract*—One promising approach to general-purpose artificial intelligence is neuroevolution, which has worked well on a number of problems from resource optimization to robot control. However, state-of-the-art neuroevolution algorithms like NEAT have surprising difficulty on problems that are fractured, i.e. where the desired actions change abruptly and frequently. Previous work demonstrated that bias and constraint (e.g. RBF-NEAT and Cascade-NEAT algorithms) can improve learning significantly on such problems. However, experiments in this paper show that relatively unrestricted algorithms (e.g. NEAT) still yield the best performance on problems requiring reactive control. Ideally, a single algorithm would be able to perform well on both fractured and unfractured problems. This paper introduces such an algorithm called SNAP-NEAT that uses adaptive operator selection to integrate strengths of NEAT, RBF-NEAT, and Cascade-NEAT. SNAP-NEAT is evaluated empirically on a set of problems ranging from reactive control to high-level strategy. The results show that SNAP-NEAT can adapt intelligently to the type of problem that it faces, thus laying the groundwork for learning algorithms that can be applied to a wide variety of problems.**

***Index Terms*—Neuroevolution, NEAT, fracture, control, strategy.**

## I. Introduction

The field of artificial intelligence stands to have a significant impact in coming years through the application of current algorithms to problems in a variety of different disciplines. There are many examples of tasks that are suitable for intelligent automation, ranging from problems that are too dangerous for humans, such as cleaning and maintaining nuclear reactor cores, to problems that require advanced and repetitive calculation, such as stock market analysis.

Encouraging the broad application of AI techniques to these kinds of problems is vital, both to practitioners in the field and to society in general. For AI researchers, implementing algorithms on a variety of problems provides pragmatic feedback about the strengths and weaknesses of current approaches outside of laboratory settings. On a larger scale, society benefits every time AI algorithms can be used to save resources, human effort, and money.

However, a widespread adoption of AI techniques will require algorithms that can function robustly in the absence of experts. Many current approaches work well in controlled settings, but behave erratically when forced into new environments. Disaster can be avoided if an expert familiar with the algorithm is available to make the necessary adjustments to fit the new situation. However, it is impractical to assume that there will be enough expertise to support a widespread adoption of AI algorithms to all of the problems to which they might be applied. Such widespread adoption will require algorithms that can function effectively without an expert to tune them to the specific conditions at hand.

One promising approach to AI is the class of reinforcement learning methods known as neuroevolution, which evolve neural networks using genetic algorithms [14, 38, 45, 62, 63, 22, 64]. NeuroEvolution of Augmenting Topologies (or NEAT) is one of the most recent successful neuroevolution methods [49, 50, 47]. While the traditional approach to reinforcement learning involves the use of temporal difference methods to estimate a value function [55, 31, 33, 34, 41, 52, 54, 56], NEAT instead relies on policy search to build a neural network (topology and weights) that maps states to actions directly. This approach has proved to be useful on a wide variety of problems and is especially promising in challenging tasks where the state is only partially observable, such as pole balancing, vehicle control, collision warning, and character control in video games [16, 49, 29, 44, 48, 50, 51]. However, despite its efficacy on such reactive control problems, other types of problems such as concentric spirals classification, multiplexer, and high-level decision making in general have remained difficult for neuroevolution algorithms like NEAT to solve.

One explanation is the *fractured problem hypothesis*, which posits that high-level strategic problems are difficult to solve because the optimal actions change abruptly and repeatedly as agents move from state to state [26, 28, 27]. Previous investigations of NEAT's performance on fractured problems have confirmed this hypothesis, showing that biasing the network toward local decision regions by using radial basis function (RBF) nodes and constraining its topology to cascaded structures can improve performance significantly on these types of problems [28]. The resulting algorithms, RBF-NEAT and Cascade-NEAT, have been shown to perform well on problems that NEAT has difficulty in solving.

Having different algorithms that perform well on different classes of problems is a step in the right direction, but still requires expertise in pairing an appropriate algorithm with a given problem. This paper investigates how these three approaches to neuroevolution – NEAT, RBF-NEAT, and Cascade-NEAT – can be integrated together into a single algorithm that can be applied as is to a broad variety of problems.

The key idea is to allow evolution to select from unrestricted, RBF, and cascade mutations based on how effective they are in the domain.

The next section reviews prior work on fractured problems, NEAT, RBF-NEAT, and Cascade-NEAT, concluding that while RBF-NEAT and Cascade-NEAT perform well on problems that are fractured, the standard NEAT algorithm still works best on those that are not. Section III introduces an integrated algorithm, SNAP-NEAT, that combines the strengths of these three approaches. In Section IV, SNAP-NEAT is evaluated empirically on a variety of problems, fractured and non-fractured, ranging from reactive control to high-level strategy, and found to perform comparably to the best methods in each. Thus, SNAP-NEAT is a general approach that can be applied to a variety of problems from reactive control to high-level strategy.

## II. Neuroevolution and Locality

This section reviews the standard NEAT algorithm, the definition of fractured problems, and two modifications to NEAT – RBF-NEAT and Cascade-NEAT – designed to improve its performance on fractured problems. In addition, an empirical evaluation of these approaches on a benchmark pole-balancing problem is described that shows that while there are benefits to using RBF-NEAT and Cascade-NEAT in problems that are fractured, the standard NEAT algorithm still works best on those that are not.

### A. NEAT

Neuroevolution algorithms use some flavor of evolutionary search to generate neural network solutions to reinforcement learning problems. This section reviews one promising such algorithm, NEAT [49], which will serve as a focus of investigation for this paper.

Neuroevolution algorithms are frequently divided into two groups: those that optimize the weights of a fixed-topology network, and those that evolve both the network topology and the weights. Most of the early work in neuroevolution focused on fixed-topology algorithms [14, 38, 45, 62, 63]. This work was driven by the simplicity of dealing with a single network topology and theoretical results showing that a neural network with a single hidden layer of nodes could approximate any function, given enough nodes [21].

However, there are certain limits associated with fixed-topology algorithms. Chief among those is the issue of choosing an appropriate topology for learning a priori. Even assuming that the general class of network topology is known (i.e. number of hidden nodes, hidden layers, recurrent layers, and the associated connectivity between nodes) there is no clear procedure to choose the network size. Networks that are too large have extra weights, each of which adds an extra dimension of search. On the other hand, networks that are too small may be unable to represent solutions of a certain level of complexity, which can limit the algorithm unnecessarily.

Neuroevolution algorithms that evolve both topology and weights (so-called constructive neural network algorithms, or TWEANNs, i.e. topology and weight evolving artificial neural

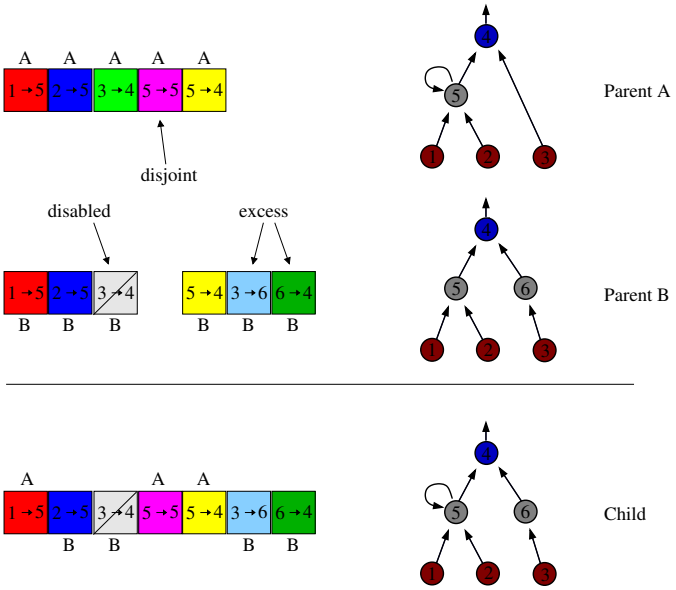



Fig. 1. An example of how NEAT evolves network topologies via innovation numbers, indicated by the color of each gene in this figure. By providing a principled mechanism to align genetic information between two genomes, NEAT is able to perform meaningful crossover between networks with different topologies.

network algorithms) were created to address this problem. One popular such algorithm is Neuroevolution of Augmenting Topologies (NEAT; Stanley and Miikkulainen, 2002).

NEAT is based on three key ideas. First, evolving network structure requires a flexible genetic encoding that allows two networks with arbitrary topology to be recombined. Each genome in NEAT includes a list of connection genes, each of which refers to two node genes being connected. Each connection gene specifies the in-node, the out-node, the weight of the connection, whether or not the connection gene is expressed (an enable bit), and an innovation number, which allows finding corresponding genes during crossover. Mutation can change both connection weights and network structures. Connection weights are mutated in a manner similar to any neuroevolution system. (In this paper, the probability $\varepsilon_W = 0.01$ was used for each gene.) Structural mutations, which allow complexity to increase, either add a new connection or a new node to the network (with probability $\varepsilon_N = \varepsilon_L = 0.05$). Through structural mutation, genomes of varying sizes are created, sometimes with completely different connections specified at the same positions. In order to perform meaningful crossover between two networks that may have differing topologies, NEAT uses the innovation numbers from each gene to "line up" genes with similar functionality (Figure 1).

Second, NEAT speciates the population so that individuals compete primarily within their own niches instead of with the population at large. This way, topological innovations are protected and have time to optimize their structure before they have to compete with other niches in the population. The reproduction mechanism for NEAT is explicit fitness sharing [12], where organisms in the same species must share the fitness of their niche, preventing any one species from taking over the population. In addition, an elitism mechanism

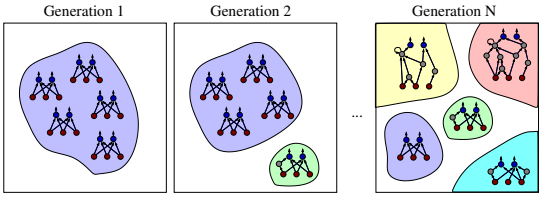



Fig. 2. An example of complexification in NEAT. An initial population of small networks gradually speciates into a more diverse population. This process allows NEAT to search efficiently in the high-dimensional space of network topologies.

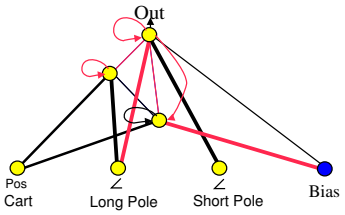



Fig. 3. A surprisingly small solution that was evolved by NEAT to solve the non-Markov double pole balancing problem [49]. Shared recurrent connections between the two poles allow the network to compute the velocity of the poles, allowing NEAT to generate a parsimonious solution to this problem.

preserves the best $\varpi = 5$ networks in the population.

Third, unlike other systems that evolve network topologies and weights [17, 64], NEAT uses *complexification*: It starts with simple networks and expands the search space only when beneficial, allowing it to find significantly more complex controllers than other neuroevolution algorithms can. More specifically, NEAT begins with a uniform population (of $\gamma = 50$) networks with no hidden nodes and randomly-initialized weights on the connections from inputs to outputs. New structure is introduced incrementally as structural mutations occur, and the only structures that survive are those that are found to be useful through fitness evaluations. In this manner, NEAT searches through a minimal number of weight dimensions and finds the appropriate level of complexity for the problem, making it an attractive method for evolving neural networks in complex tasks.

### B. Performance of NEAT

The three key ideas of NEAT allow it to search quickly and efficiently through the space of possible network topologies to find the right neural network for the task at hand. This approach is highly effective: NEAT has outperformed other neuroevolution methods on complex control tasks like double pole balancing [49] and robotic strategy-learning [50]. For instance in pole balancing, NEAT is able to discover surprisingly small and elegant solutions that utilize network structures, and in particular recurrence, to achieve smooth control (Figure 3).

However, NEAT is limited to small, incremental changes in the network structure. While such mutations are useful when building relatively small networks, tasks that require complicated or repeated internal structure are difficult for NEAT. Furthermore, any small mutations that are made to network structure can potentially have a global impact on network output. If solving a problem requires local adjustments to network output, NEAT's performance may suffer [28, 27]. Indeed, it has turned out to be surprisingly difficult to get NEAT to perform well in problems such as concentric spirals, multiplexer, and high-level strategy problems in general.

The next section reviews the fractured problem hypothesis, which posits that such problems are difficult to solve because the correct action changes frequently and abruptly as the agent encounters different states.

### C. Fractured Problems

For many problems (such as the typical control problems or the standard reinforcement learning benchmarks), the correct

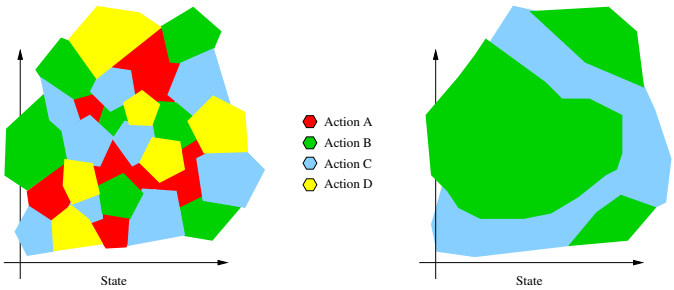



Fig. 4. A simple example of a 2-d state-action space that is (a) fractured and (b) unfractured. In (a), the correct actions vary frequently and discontinuously as an agent moves through the state space. If a learning algorithm cannot represent these abrupt changes, its performance will be limited.

action for one state is similar to the correct action for neighboring states, varying smoothly and infrequently. In contrast, for a fractured problem, the correct action changes repeatedly and discontinuously as the agent moves from state to state. Figure 4 shows simple examples of a fractured and unfractured two-dimensional state space.

This definition of fracture, while intuitive, is not precise enough to be used to measure learning performance. More formal definitions of difficulty have been proposed for learning problems, including Minimum Description Length [2, 5], Kolmogorov complexity [30, 35], and Vapnik-Chervonenkis (VC) dimension [59]. Unfortunately, these metrics are often more suited to a theoretical analysis than they are to practical usage. For example, Kolmogorov complexity depends on the computational resources required to specify an object, which sounds promising for measuring problem fracture, but it has been shown to be uncomputable in practice [36].

Fortunately, previous work has shown that it is possible to define fracture rigorously using the mathematical concept of total variation [26]. By treating a solution to a problem as a function, it is possible to measure the amount of variation of that function, yielding an estimate of how fractured the solution space is for that problem. In particular, consider the optimal solution for a problem to be a function $z$ over a region of the state space $B$. The amount of fracture in this solution can be defined as $V(z, B)$:

$$V(z,B) = \sum_{m=1}^{N-1} \left\{ \sum_{r=1}^{N_m} V_m(z, B_r^{(m)}) \right\} + V_N(z,B), \quad (1)$$

where the function $V_N$ is defined as

$$V_N(z, B) = \sup_{\Pi} \left\{ \sum_{j=1}^{n} |\sigma_N(B_j)| : \Pi = \{B_j\}_{j=1}^{n} \in P \right\}, \tag{2}$$

and $\sigma_N$ is defined as

$$\sigma_N(B_\alpha^\beta) = \sum_{v_1=0}^{1} \cdots \sum_{v_N=0}^{1} (-1)^{v_1+\ldots+v_N} \Lambda, \tag{3}$$

where

$$\Lambda = z[\beta_1 + v_1(\alpha_1 - \beta_1), \ldots, \beta_N + v_N(\alpha_N - \beta_N)]. \tag{4}$$

More informally, measuring the fracture of a function over a given area involves summing a number of individual variation calculations, one for each combination of dimensions of the state space. For example, for a function over a three dimensional space, variation would be measured along each dimension separately, along each pair of two dimensions, and inside all three dimensions. The sum of all of these individual measurements reflects how much the function changes in different directions. This approach to defining fracture is relatively simple, well-founded mathematically, and can be used on any functional form (e.g. neural networks). For details concerning this variation computation as well as several examples of how it can be applied, see [26, 28].

By the formulation above, fractured problems can be characterized as problems where optimal solutions have high variation. One strategy for discovering such discontinuous solutions is to focus on algorithms that are able to make local, non-disruptive adjustments to policies. The next two sections review RBF-NEAT and Cascade-NEAT, two recent neuroevolution algorithms that were designed to solve such fractured problems by taking advantage of the locality introduced by biasing and constraining the growth of network topologies.

### D. RBF-NEAT

Radial basis function networks [18, 37, 40, 42] are well-known in the supervised machine learning literature for their ability to construct complex decision regions. This ability is based on nodes with local activation functions such as the Gaussian. The first locality algorithm, called RBF-NEAT, extends NEAT by introducing a new topological mutation that adds such a radial basis function node to the network. This mutation is an addition to the normal mutation operators used by NEAT, giving it the ability to generate networks that have both sigmoid-based nodes and basis-function nodes.

Like NEAT, RBF-NEAT starts with a minimal topology, in this case consisting of a single layer of weights connecting inputs to outputs, and no hidden nodes. In addition to the normal "add link" and "add node" mutations, RBF-NEAT also employs an "add RBF node" mutation with probability $\varepsilon_{\text{RBF}} = 0.05$ (Figure 5). Each RBF node is activated by an axis-parallel Gaussian with variable center and size, and is connected to all input and output nodes by randomly-weighted

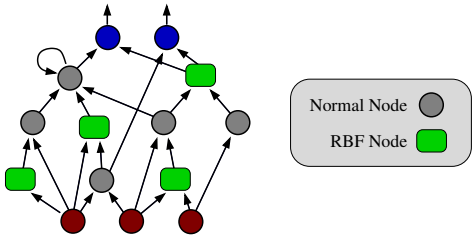



Fig. 5. An example of network topology evolved by the RBF-NEAT algorithm. Radial basis function nodes, initially connected to inputs and outputs, are provided as an additional mutation to the algorithm. Because the RBF nodes have local activation functions, the resulting network will be able to make decisions based on small differences in the input, i.e. on problems where the decision boundary is fractured.

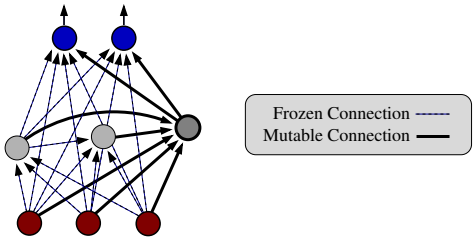



Fig. 6. An example of a network constructed by Cascade-NEAT. Only connections associated with the most recently added hidden node are evolved. Compared to NEAT and RBF-NEAT, Cascade-NEAT constructs networks with a regular topology that results in local processing.

connections. All free parameters of the network, including RBF node parameters (center and width) and connection weights, are determined by the same genetic algorithm used in NEAT [49]. Since RBF-NEAT builds on the standard NEAT algorithm, the only additional parameter that it introduces is $\varepsilon_{\text{RBF}}$, the probability of adding an RBF node. The value used for this additional parameter was determined empirically and held constant for all experiments described in this paper.

RBF-NEAT is designed to evaluate whether local processing nodes can be useful in policy-search reinforcement learning problems. The addition of a RBF node mutation provides a bias towards local-processing structures, but the normal NEAT mutation operators still allow the algorithm to explore the space of arbitrary network topologies. RBF-NEAT is effective in particular in low-dimensional problems because Gaussian functions are a simple way to isolate pieces of the input space and the number of parameters required to define each dimension of such Gaussian functions remains manageable.

### E. Cascade-NEAT

An alternative way to introduce locality is to constrain the topology search to a specific set of structures that make local refinements to the decision regions. The cascade-correlation algorithm [9] is a powerful such approach that has proved to be useful in many supervised learning problems. The cascade architecture (shown in Figure 6) is a regular form of network where each hidden node is connected to inputs, outputs, and all previously-existing hidden nodes. The second extended algorithm, Cascade-NEAT, constrains the search process to topologies that have such a cascaded structure.

Like NEAT, Cascade-NEAT starts from a minimal network consisting of a single layer of connections from inputs to outputs. Instead of the normal NEAT mutations, however, Cascade-NEAT uses an "add cascade node" mutation (with probability $\varepsilon_{\text{Cascade}} = 0.05$) that adds a standard hidden node to the network. This hidden node receives connections from all inputs and existing hidden nodes in the network, and is connected to all outputs. All of these connections are initialized with random weights. In addition, whenever a hidden node is added, all pre-existing network structure is frozen in place. Thus, at any given time, the only mutable parameters of the network are the connections that involve the most recently-added hidden node. This freezing process focuses the search for network weights on a small subset of the overall network structure, greatly reducing the size of the search space. Like RBF-NEAT, Cascade-NEAT builds off of the baseline NEAT algorithm, and therefore shares all parameters that NEAT has. The only additional parameter introduced by Cascade-NEAT is $\varepsilon_{\text{Cascade}}$, the probability of adding a cascade node. Like all NEAT parameters, the value for this additional parameter was determined empirically and held constant for all of the experiments in this paper.

The constraint that Cascade-NEAT adds to the search for network topologies is considerable, given the wide variety of network structures that the normal NEAT algorithm examines. The idea is that this restriction results in gradual abstraction and refinement, which allows the discovery of solutions with local processing structure useful in fractured problems.

### *F. Performance of RBF-NEAT and Cascade-NEAT*

In prior work, RBF-NEAT and Cascade-NEAT were compared to the standard NEAT algorithm on a benchmark suite of problems including variation generation, function approximation, concentric spirals, multiplexer, and keepaway soccer [28, 27]. Each of these problems were chosen because they cover a variety of different types of domains, yet are simple enough such that optimal solutions are known a priori. It was therefore possible to directly measure the fracture for these problems by computing the solutions as functions and measuring the total variation of those functions.

The results of this comparison showed that as the level of fracture in a problem increases, RBF-NEAT and Cascade-NEAT perform progressively better than NEAT [26]. Biasing and constraining the construction of networks allows these approaches to better model the local decision regions that make the problem fractured. In effect, RBF-NEAT and Cascade-NEAT extend the NEAT approach to fractured problems.

But what about other types of problems? The benefits of biasing network construction that RBF-NEAT and Cascade-NEAT use could conceivably allow them to dominate the standard NEAT algorithm on all domains. However, experiments showed that NEAT still performs better on certain types of control problems such as pole-balancing [50]. As Figure 3 shows, it is possible to do well on double pole-balancing with a very small recurrent network. Since the NEAT algorithm starts with a population of minimal networks, it is well-prepared to solve problems that have small solutions.

Thus, it would be desirable to combine the strengths of RBF-NEAT, Cascade-NEAT, and standard NEAT into a single algorithm that can perform well on both low-level control tasks and fractured problems. The next section describes how this goal can be achieved.

## III. An Integrated Approach

The combined approach proposed in this section takes advantage of the fact that NEAT, RBF-NEAT, and Cascade-NEAT are almost completely identical except in their topological mutation strategy. The standard NEAT algorithm uses two topological mutation operators: add-link (between two unconnected nodes) and add-node (split a link into two links with a node between them). RBF-NEAT adds a third mutation operator, add-RBF-node, which adds a special Gaussian basis-function node and connects it to inputs and outputs. In contrast, Cascade-NEAT uses only a single structural mutation operator, add-cascade-node, which adds a normal node that receives input from input and hidden nodes and which sends output to the output nodes. In addition, this operator freezes the previously-existing network structure to prevent the effective search space for connection weights from increasing too quickly. The goal of this approach is to combine these mutations intelligently into an algorithm that utilizes each mutation when it is the most effective.

### *A. Adaptive operator selection*

The problem of choosing the correct mutation operators for a domain is known as adaptive operator selection [11, 1, 23, 57, 6]. The traditional and by far the simplest approach is to choose uniformly randomly between all operators. However, if certain operators are more useful than others, the selection of poor operators can limit learning performance. The goal of adaptive operator selection research is to make a more informed decision about which operators to choose.

Early research in adaptive operator selection collected statistics such as how frequently a chosen operator resulted in an improvement in score over its parents or resulted in a new best score for the entire population [7, 23]. In order to give credit to the operators that led to such individuals, the estimated value of operators was propagated backwards from an individual to its parents. Some methods attempted to avoid the whole credit assignment problem by periodically re-calculating the value of all operators using only information from the current population [58]. After the value of the various operators was estimated using one of these methods, the probability of choosing an operator was calculated in a process known as Probability Matching [11, 1]. Such algorithms simply assign operator probabilities proportionally to expected value, while also enforcing certain minimum probabilities for each operator.

One of the more popular modern adaptive operator selection algorithms is Thieren's Adaptive Pursuit algorithm [57]. At every timestep, this algorithm attempts to identify the optimal probability for choosing operator $o_i$, with the goal of maximizing expected cumulative reward of the algorithm. It keeps estimates of the value $Q_{o_i}$ for each operator, and then uses those estimates to weight the probability $P_{o_i}$ of selecting each

operator. Adaptive Pursuit is designed to respond quickly to changes in estimated operator value and emphasize selection of the highest-valued operator without completely ignoring other possible operators.

For example, given two operators $o_1$ and $o_2$, rewards $R_{o_1} = 10$ and $R_{o_2} = 9$, and a minimum probability $P_{\min} = 0.1$, then Probability Matching will assign probabilities of $P_{o_1} = 0.52$ and $P_{o_2} = 0.48$ to the operators. It would be arguably preferable to have an algorithm that assigns probabilities of $P_{o_1} = 0.9$ and $P_{o_2} = 0.1$. Adaptive Pursuit achieves this goal by increasing the selection probability of the operator with the highest value, $o_* = \text{argmax}_i[Q_{o_i}]$:

$$P_{o_*}(t+1) = P_{o_*}(t) + \beta[P_{\max} - P_{o_*}(t)] \tag{5}$$

while decreasing the probabilities of all other operators:

$$\forall o_i \neq o_* : P_{o_i}(t+1) = P_{o_i}(t) + \beta[P_{min} - P_{o_i}(t)] \tag{6}$$

In these equations, $\beta$ is a free parameter that controls the rate at which these probabilities are updated. Another free parameter, $\alpha$, serves a similar role in governing how fast the reward $Q_{o_i}$ is updated. The value of $P_{max}$ is constrained to be $1 - (K-1)P_{min}$, where $K$ is the number of operators. These calculations effectively select the estimated optimal operator with probability $P_{max}$, while choosing uniformly among the other operators the rest of the time. This strategy allows Adaptive Pursuit to place much higher value on the single best operator than strategies like Probability Matching. Empirically, this decision has shown to improve performance, making Adaptive Pursuit an appealing choice for a NEAT-combination algorithm [57, 6]. However, it is not necessarily straightforward to integrate Adaptive Pursuit with NEAT, as will be discussed next.

### B. Continuous updates

Previous approaches to adaptive operator selection, including Adaptive Pursuit, estimate operator value immediately after application. For example, after an operator is chosen and used to create or update a member of the population, the resulting change in score is noted and applied to that operator. This approach, while certainly straightforward, is not necessarily appropriate for algorithms based on NEAT. One of the tenets of NEAT is that new structural mutations may require some time to be optimized before they become competitive with existing structures. The purpose of speciation in NEAT is to provide temporary shelter for new structures that arise in the population, giving them a fair chance to compete with structures that have had more time to be optimized. This concept of delayed evaluation has proven useful in NEAT [50], but conflicts slightly with the approach taken by Adaptive Pursuit. Estimating the value of an operator immediately after application could result in an inaccurate estimate of the value of the operator.

An alternative method of estimating operator value is to keep track of which operator most recently affected each member of the population. As a given member of the population improves, the estimate of the value of the operator that most recently contributed to it will also be updated. In essence, every time an individual is updated (regardless of whether or not it was just modified by a structural mutation operator) an updated reward signal will be generated for the operator that most recently contributed to that individual. This process keeps operator values up-to-date with the current population, and also utilizes a much larger percentage of the information that the learning algorithm has available to it. Performing such continuous updates to operator values also fits nicely with the NEAT philosophy, where speciation is used to give new network topologies in the population a chance to compete.

### C. Initial estimation

The standard Adaptive Pursuit algorithm uses a winner-take-all strategy to increase the likelihood of choosing the best operator at every timestep. This greedy approach is offset by a minimum probability $P_{min}$ for each operator, which is designed to make it possible for the algorithm to change its operator selection strategy in the middle of learning. However, the winner-take-all strategy can still be sensitive to initial conditions.

If two operators have expected values that are close to each other, small differences in early evaluations can cause the Adaptive Pursuit algorithm to greedily choose the wrong operator. Such a mistake early in learning is not necessarily lethal — thanks to the minimum probabilities associated with each operator — but if the learning rate that governs how quickly probabilities can change is low, it can take a while to recover from initial errors in probability estimation.

In order to better estimate the initial values $P_{o_i}$ of all operators $o_i$, another modification to the Adaptive Pursuit algorithm was developed in this paper. The main idea is that the first $N$ evaluations will serve as an evaluation period for all operators, wherein each operator will be evaluated an equal number of times. During this period, the probability $P_{o_i}$ for each operator $o_i$ will remain fixed and uniform. After the $N$ evaluations have been completed, the information gained from those evaluations will be used to compute estimated values $Q_{o_i}$ for each operator $o_i$. The algorithm then uses these initial value estimates to compute initial probability estimates $P_{o_i}$ and resumes normal operation for the remaining evaluations.

Of course, since this initial evaluation period is used only to compute good estimates for operator values and does not attempt to take advantage of operators that appear to be performing well, such an approach could prove detrimental to learning. However, if it is important to start with good initial values for each operator, taking time for this initial evaluation could prove worthwhile. An empirical evaluation of how useful both continuous updates and an initial estimation period are is presented below.

### D. SNAP-NEAT

SNAP-NEAT is a new version of the NEAT algorithm that uses the adaptive operator selection mechanisms from Adaptive Pursuit to integrate the mutation operators from NEAT, Cascade-NEAT, and RBF-NEAT. The two NEAT operators, add-node and add-link, are grouped together into a single

operator for the purposes of estimating operator value and probability. When this operator is selected for actual use, a coin flip determines whether add-node or add-link is actually run. This grouping forces the NEAT operators to change values in tandem.

SNAP-NEAT incorporates the two modifications discussed above, continuous updates and initial estimation. During the initial estimation period, SNAP-NEAT cycles repeatedly between the three topological mutation types, noting the scores associated with each operator. When the initial estimation period ends, the value for each operator $Q_{o_i}$ is initialized to one standard deviation above the mean of the values accumulated for $o_i$. In a manner similar to interval estimation, this method of initialization incorporates uncertainty about the true value for $o_i$ [24]. For the remaining evaluations, a structural mutation operator $o_i$ is selected according to its probability $P_{o_i}$, and both expected values $Q_{o_i}$ and probabilities $P_{o_i}$ are updated after each evaluation.

Since SNAP-NEAT is built on NEAT, RBF-NEAT, and Cascade-NEAT, it utilizes the parameters for those algorithms. In addition, SNAP-NEAT adds parameters defining the initialization period (set to $N = 10000$ evaluations, i.e. one-fifth of total $T = 50000$ evaluations performed during learning) learning rates $\alpha$ and $\beta$ (both $= 0.05$), and a minimum probability for each operator in adaptive pursuit ($\varphi = 0.10$). Values for these additional parameters were determined empirically and held constant for all experiments in this paper. Psuedocode describing how the SNAP-NEAT algorithm uses these parameters can be found in Algorithm 1. Note that implementations of NEAT, RBF-NEAT, and Cascade-NEAT can be derived from this explanation of SNAP-NEAT by setting $P$ to a constant value that always selects a specific structural mutation.

With reasonable values for these parameters, SNAP-NEAT is able to find an appropriate learning algorithm for the problem at hand. This allows experimenters to avoid the unpleasant situation of deciding whether poor performance during learning stems from a bad choice of parameters or a bad choice of learning algorithm. Thus, SNAP-NEAT uses a modified version of Adaptive Pursuit to make intelligent decisions about whether to favor NEAT, RBF-NEAT, or Cascade-NEAT for a given problem. The next section evaluates SNAP-NEAT on a suite of benchmark problems to determine its efficacy.

**Algorithm 1** Pseudocode for the SNAP-NEAT algorithm.

```
pop = initializePopulation(γ)
Q = {0}
P = {1/3}  // three mutations: NEAT, RBF, Cascade
numEvals = 0
numGens = T/γ
for i = 1 to numGens do
  pop' = {}
  for n in pop do
    s = evaluate(n)
    updateValue(Q, lastOperator(n), s)
    updateProbability(Q, P, α, β, φ)
    if isElite(s, ϖ) then
      add(pop', n)
    else if rand < P_c then
      operator = chooseWeighted(P)
      n' = mutate(n, operator)
      add(pop', n')
    else
      n' = mutateWeights(n)
      add(pop', n')
    end if
  end for
  pop = pop'
end for
```

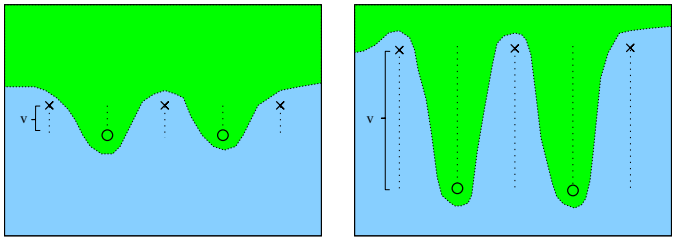
## IV. Empirical Evaluation

If SNAP-NEAT works properly, then it should be able to recognize which NEAT mutation strategy is required for a given problem. Selecting an appropriate strategy should improve SNAP-NEAT's performance relative to algorithms with a fixed strategy that is not suited to the given problem. In addition, examining how well the final probabilities for each operator match the best known algorithm can help determine how successful SNAP-NEAT is at selecting appropriate operators. This section evaluates how well SNAP-NEAT can learn to identify the correct operators for a variety of problems that require different strategies to solve.

All of the data shown in this section represent averages of 100 independent runs. The error bars that appear on graphs denote standard error of the mean. All conclusions described in this paper as being significant were confirmed with a Student's *t*-test with a probability of at least $p > 0.95$ [39]. All of the parameters used by the learning algorithms were determined empirically to work well on a variety of problems, and were not fine-tuned for any particular algorithm or problem. All of the parameters that were shared between NEAT, RBF-NEAT, Cascade-NEAT, and SNAP-NEAT were the same for all experiments.

### A. N-Point classification

The first problem is a simple N-Point classification task. The goal is to classify each of a set of $N = 10$ alternating points properly into one of two groups. This problem is interesting because it can be either fractured or unfractured, depending on the distance between the two categories of points. Thirteen different versions of this problem were created to examine how different amounts of variation (i.e. fracture, as defined in Section II-F) in optimal policies impact learning performance. In each version, the two classes of alternating points were separated by amounts varying from $\upsilon = 0.01$ to 1.0. When $\upsilon$ was low, the optimal policy for distinguishing the two groups required very little variation. As $\upsilon$ increased, so did the amount of variation required for an optimal policy. For example, Figure 7 shows two of the 13 problems when $N = 5$, along with examples of optimal policies for each problem. When the two classes of points are relatively close together, the decision boundary between the two classes can be relatively smooth. As the two classes are moved farther apart, the boundary becomes increasingly non-linear, which increases the minimal variation




(a) (b)

Fig. 7. Examples of solutions to the N-Points problem when $N = 5$ and the separation between the two classes of points is (a) $\upsilon = 0.1$ and (b) $\upsilon = 0.8$. As $\upsilon$ increases, the two classes of points move further away from each other, and the minimal variation required to describe the boundary between the two classes increases, making the problem more fractured.

required to describe the boundary. Thus the degree of fracture grows larger as $\upsilon$ increases.

Each network was evaluated on a series of inputs, each having a value in $[0, 1]$ that represented one of the $N$ 1-d points. Network activation was reset between successive inputs. The correct output for the network depended on both the class to which the current point belonged and the separation parameter $\upsilon$. When $\upsilon$ was small, there was a large range of values that the network could produce that would yield a correct classification. As $\upsilon$ increased, that window shrank, such that when $\upsilon = 1.0$, the only correct values that a network could produce were 0 if the point belonged to the first class and 1 if it belonged to the second class. After being presented with all $N$ input points, the fitness for a network was defined to be $10 - \chi$, where $\chi$ was the number of misclassified points.

The minimal amount of variation required to solve an instance of the N-Point classification problem is relatively straightforward to compute. In order to separate the $N$ alternating points into two groups, a function must alternate between producing values at least as large as $\upsilon/2$ and at least as small as $-\upsilon/2$. Since there are $N - 1$ gaps between adjacent points that the function must alternate over, the minimum amount of variation required to properly classify all $N$ points is $(N-1)\upsilon$.

Because of its relatively low-dimensional input space, RBF-NEAT generated the best performance in the 10-Point classification problem. Figure 8 compares the performance of SNAP-NEAT to the other algorithms on this problem (averaged over 100 independent runs for each algorithm) and Figure 9 shows the learned probabilities for SNAP-NEAT. SNAP-NEAT has learned to heavily favor the add-RBF-node mutation, confirming its ability to find the appropriate operator for this problem.

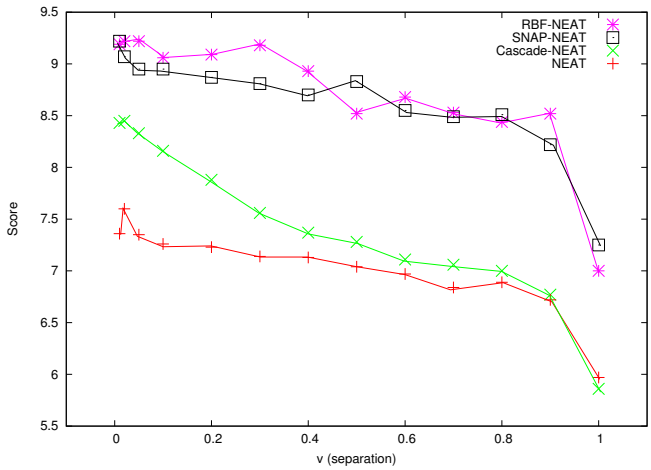



Fig. 8. The performance of SNAP-NEAT on the 10-point classification problem. Each point represents the average of 100 independent runs. SNAP-NEAT is able to take advantage of the add-RBF-node mutation on this problem, giving it a score comparable to that of RBF-NEAT ($p > 0.95$).

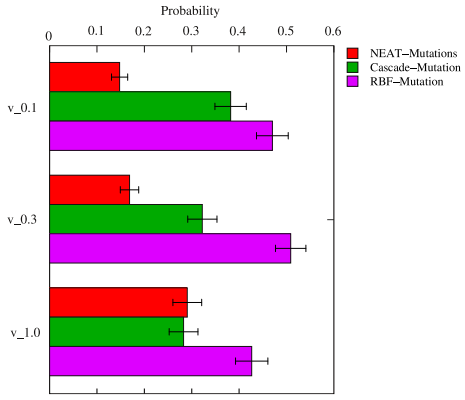



Fig. 9. The probabilities learned by SNAP-NEAT for each operator for the 10-point classification problem when $\upsilon$ is 0.1, 0.3, and 1.0 . As variation increases, RBF-NEAT offers the best performance for this problem, and SNAP-NEAT learns to favor the add-RBF-node mutation.

### B. Multiplexer

The multiplexer problem is a challenging benchmark from the evolutionary computation community. An agent must learn to split the input into address and data fields, then decode the address and use it to select a specific piece of data. For example, the agent might receive as input six bits of information, where the first two bits denote an address and the remaining four bits represent the data field. The two address bits indicate which one of the four data bits should be selected as output.

This section describes experiments with four versions of the multiplexer problem, which are shown in Figure 10. These four problems differ in the size of the input, ranging from three inputs (one address bit and two data bits) to nine inputs (three address bits and six data bits). As before, these different configurations of the problem are fractured to different degrees: the three-input problem is relatively unfractured, whereas the nine-input problem is relatively fractured. Note that in order to make the problem tractable, not all inputs involving the third address bit are used for the two largest versions of the problem.

Each version of the multiplexer problem effectively defines a binary function from the input bits to a single output bit. During learning, every possible combination of binary inputs (given the constraints on address and data bits) was presented to each network in turn. As before, network state was cleared between consecutive inputs. The fitness for each network was

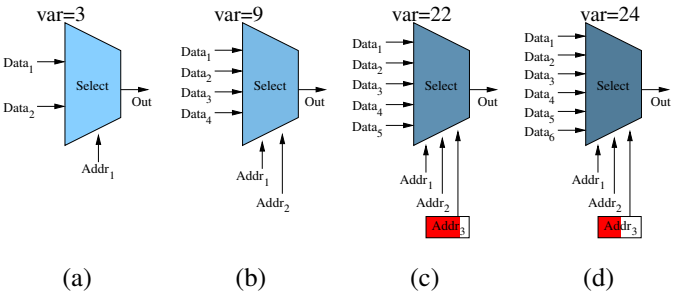



Fig. 10. Four versions of the multiplexer problem, where the goal is to use address bits to select a particular data bit. For (c) and (d), not all of the values for the third address bit were used. The amount of variation required to solve the multiplexer problem increases as the number of total inputs (address bits plus data bits) increases.

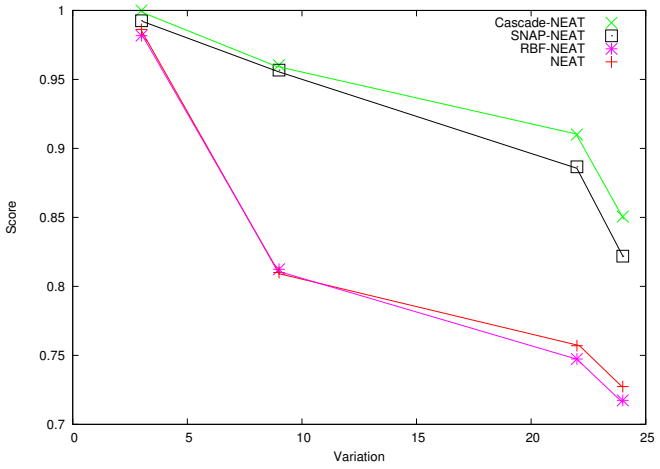



Fig. 11. An evaluation of SNAP-NEAT on four versions of the multiplexer problem. Each point represents the average of 100 independent runs. SNAP-NEAT's performance is near that of Cascade-NEAT, and both approaches are significantly better than RBF-NEAT and NEAT on the more fractured problems ($p > 0.95$).

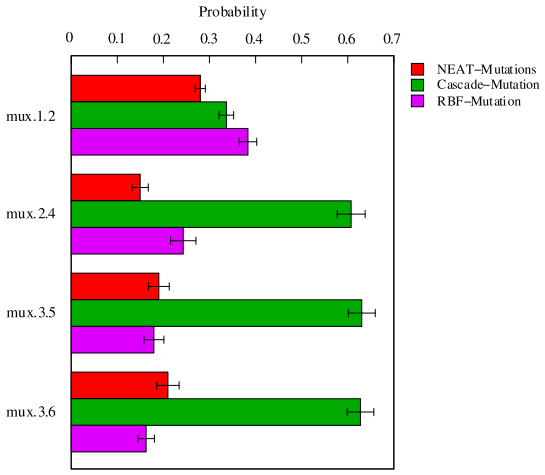



Fig. 12. The probabilities learned by SNAP-NEAT for each operator for the four versions of the multiplexer problem. When the problem is relatively unfractured (the upper problems), the learned probabilities are similar. However, on the more difficult versions of this problem (near the bottom), SNAP-NEAT learns to rely heavily on the add-cascade-node mutation.

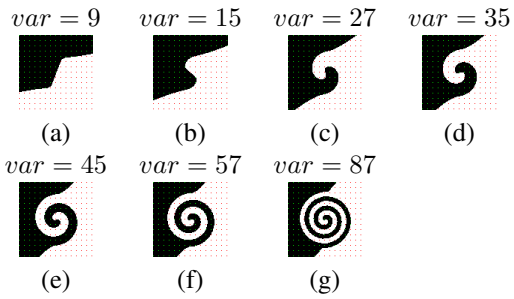



Fig. 13. Seven versions of the concentric spirals problem that vary in the degree to which the two spirals are intertwined. The colored dots indicate the discretization used to generate data from each spiral. As the spirals become increasingly twisted, the variation of the optimal policy increases.

the inverted mean squared error over all inputs.

Results for 100 independent run of four versions of the multiplexer problem are shown in Figure 11, followed by learned probabilities in Figure 12. In contrast to N-Point classification, the utility of the Cascade-NEAT approach is exceedingly clear for this problem, and SNAP-NEAT correspondingly learns to heavily emphasize the add-cascade-node mutation. As before, this result demonstrates SNAP-NEAT's ability to favor one operator with near-exclusivity when the problem demands it.

SNAP-NEAT also performs well when compared to other learning approaches in the multiplexer domain. In particular, with Gene Expression Programming on the 2-4 multiplexer problem, nearly twice as many evaluations were required (100,000 evaluations versus 50,000 for SNAP-NEAT) to achieve results comparable to those of SNAP-NEAT [10].

### C. Concentric spirals

The concentric spirals problem is a classic supervised learning benchmark task popularized by the cascade-correlation literature. Originally proposed by Wieland [43], the problem consists of identifying 2-d points as belonging to one of two intertwined spirals. Each network to be evaluated is presented with a selection of 2-d input points in the range $[0, 1]$, and the output of the network represents a binary signal (black $< 0.5$, white $\geq 0.5$) describing which spiral the network has assigned to each point. Fitness is defined as the number of properly classified points. As before, network state was cleared between consecutive inputs. Solving the concentric spirals problem involves tagging nearby regions of the input space repeatedly with different labels, which matches the description of a fractured problem intuitively.

In order to examine the effect of changing amounts of fracture on the learning algorithms, seven different versions of the problem were created, varying the degree to which the two spirals are intertwined. These versions are shown in Figure 13. As the spirals become increasingly intertwined, the variation of the optimal policy (which classifies each point as being on one spiral or the other) increases, indicating an increased level of fracture.

The results of 100 runs of each algorithm are shown in Figure 14. As with the multiplexer problem, Cascade-NEAT performs consistently better than RBF-NEAT on these problems. SNAP-NEAT performs similarly to Cascade-NEAT, demonstrating that it has chosen the right operators for these problems. However, in several of the low-fracture cases it

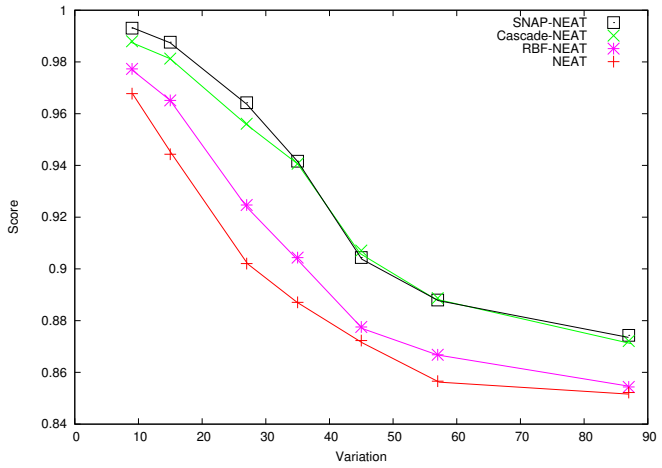



Fig. 14. Performance of SNAP-NEAT on several versions of the concentric spirals problem. Each point represents the average of 100 independent runs. SNAP-NEAT is able to perform comparably to Cascade-NEAT, and in low-variation cases, actually slightly better ($p > 0.95$).

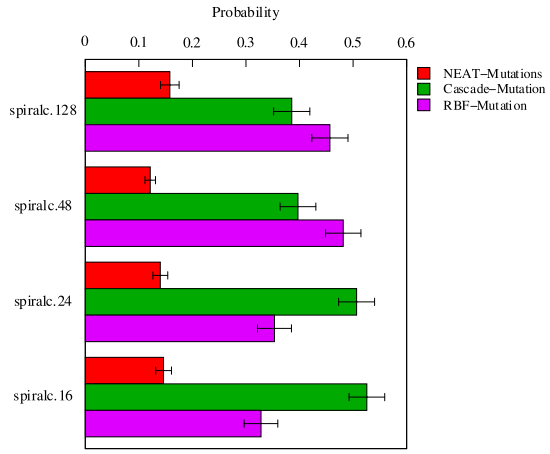



Fig. 15. Operator probabilities learned by SNAP-NEAT for four versions of the concentric spirals problem, arranged from least fractured (top) to most fractured (bottom). SNAP-NEAT combines RBF and cascade mutations in these problems, resulting in better performance than any single approach.

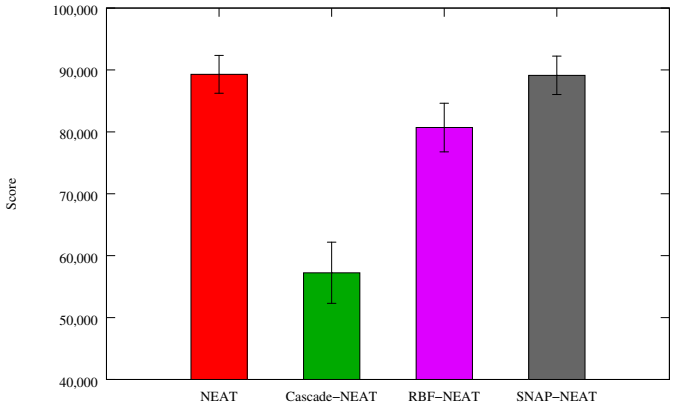



Fig. 16. A comparison of NEAT, Cascade-NEAT, RBF-NEAT, and SNAP-NEAT on the Markov double pole-balancing problem. Each bar represents the average of 100 independent runs of each algorithm. SNAP-NEAT learns that the standard NEAT mutation operators are most useful on this problem, giving it a performance comparable to that of NEAT ($p > 0.95$).

manages to outperform Cascade-NEAT, which is surprising. All of these results are significant with $p > 0.95$.

The reason is clear from the operator probabilities shown in Figure 15. In all cases, SNAP-NEAT includes a significant number of cascade mutations, as is to be expected. However, in low-fracture cases, it actually favors RBF mutations slightly. This combination allows it to perform better than any single approach on these problems, demonstrating the added value of the SNAP-NEAT approach.

Among previous evolutionary computation approaches to the concentric spirals problem, Potter and DeJong's genetic cascade-correlation algorithm is the best known [43]. Although their experiments are not directly comparable to the data presented above, genetic cascade-correlation takes nearly three times as many evaluations as SNAP-NEAT (139,500 evaluations for genetic cascade-correlation versus 50,000 for SNAP-NEAT) to solve a similar version of the concentric spirals problem. These results suggest that SNAP-NEAT is competitive in this benchmark domain as well.

### *D. Pole balancing*

The double pole-balancing problem is a classic reinforcement learning benchmark that has been used to gauge the performance of many learning algorithms [47, 49]. In this problem, the goal of the learning algorithm is to find a controller that can balance two poles of different lengths that are attached to a cart on a one-dimensional track. The controller receives input describing the position of the cart on the track and the angles of the two poles relative to the cart. In the Markov version of the problem, it also receives rates of change for these three variables; in the non-Markov version, it needs to estimate these rates for itself by integrating information from previous states. The actions available to the controller provide an impulse to the cart that accelerates it in either direction on the track. Fitness is proportional to the amount of time that the poles remain in the air, with the constraint that the cart must remain on a fixed section of the track. As is typical of control problems, the decisions in double pole balancing are relatively continuous, i.e. it is a non-fractured problem. As a result, it should be very well suited for the NEAT approach, and less so for RBF-NEAT and Cascade-NEAT.

Figures 16 and 17 compare SNAP-NEAT to NEAT, RBF-NEAT, and Cascade-NEAT on Markov and non-Markov versions of this problem. In the Markov case, RBF-NEAT and especially Cascade-NEAT perform poorly, as expected. However, SNAP-NEAT's performance is indistinguishable from that of NEAT ($p > 0.95$). In the non-Markov case, the differences are even larger between NEAT and the local methods. SNAP-NEAT also takes a small performance hit, apparently because it spends time considering the add-cascade-node mutation, which is relatively useless in this problem. However, it still performs well, achieving a level of performance near that of NEAT.

NEAT and SNAP-NEAT also perform well when compared to other pole-balancing algorithms. Although direct comparisons between the results in this paper (measuring fitness

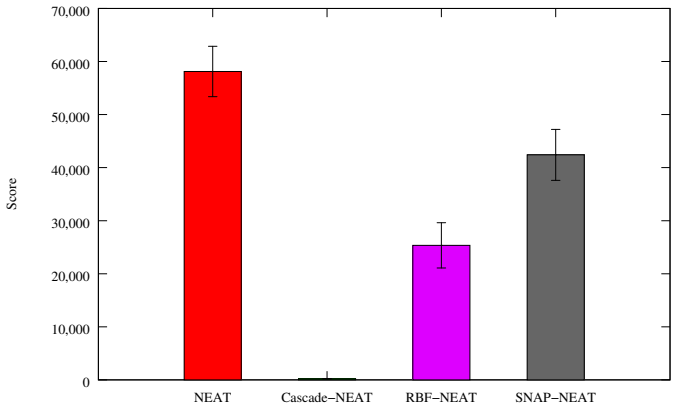



Fig. 17. A comparison of several learning algorithms on the non-Markov double pole-balancing problem, averaged over 100 independent runs. SNAP-NEAT is able to achieve a performance level near that of NEAT, although its use of the add-RBF-node and add-cascade-node mutations limits its performance somewhat compared to NEAT.

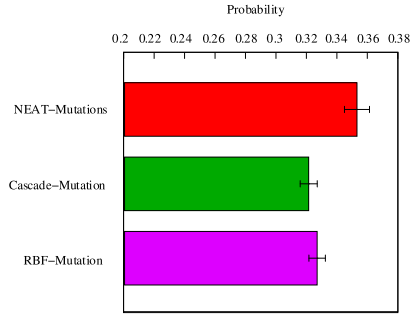



Fig. 18. The learned operator probabilities for SNAP-NEAT in Markov double pole-balancing. SNAP-NEAT discovers that the NEAT mutations are the most useful for this problem, allowing it to perform at a level comparable to NEAT.

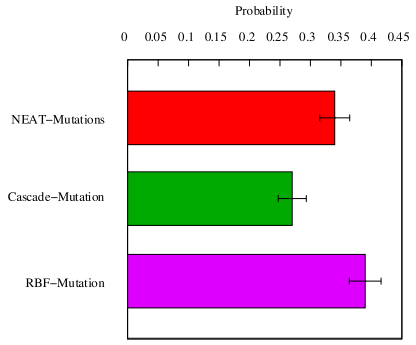



Fig. 19. The learned operator probabilities for SNAP-NEAT in non-Markov double pole-balancing. SNAP-NEAT's ability to de-emphasize the add-cascade-node mutation allows it to find solutions almost as good as those found by NEAT. However, an over-reliance on the add-RBF-node operator, although not entirely unreasonable, results in lower performance than in the Markov version of the problem.

achieved within a given number of evaluations) and prior work (measuring number of evaluations required for a solution) are not possible, a prior comprehensive comparison by [8] can be used to put the results in perspective. In that comparison, NEAT was found to require two orders of magnitude fewer calculations than cellular encoding [17] and evolutionary programming [45], and an order of magnitude fewer than conventional neuroevolution [63] and Q-Learning [60]. On the other hand, CMA-ES [19] and CoSyNE [8] performed slightly better than NEAT. Since SNAP-NEAT performs comparably to NEAT, it compares similarly to these other methods. However, CMA-ES and CoSyNE are partly based on techniques that could be incorporated into SNAP-NEAT as well; such combinations constitute an interesting direction for future work.

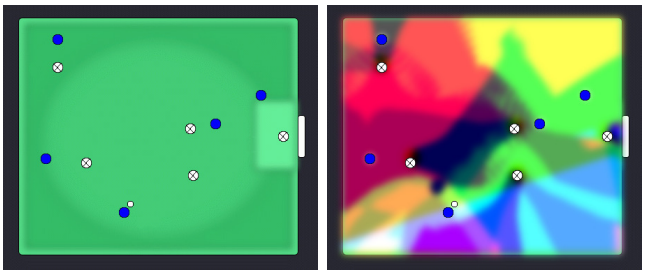
Examining the probabilities for each operator that are learned by SNAP-NEAT is another way to gauge the effectiveness of the integrated approach. Since the standard NEAT algorithm generates the best performance on this problem, a successful operator selection algorithm should learn to favor the NEAT mutation operators.

Figures 18 and 19 show the final average probabilities at the end of learning for successful runs of the SNAP-NEAT algorithm. In the Markov case, SNAP-NEAT learns to emphasize the NEAT mutations. This behavior is reasonable, given the high performance of the standard NEAT algorithm on this problem. Interestingly, in the non-Markov case, SNAP-NEAT does not learn to rely heavily on the NEAT mutations, instead striking a balance between NEAT and RBF-NEAT mutations. This result makes some sense, given that the input space is relatively low-dimensional (as a matter of fact, RBF-NEAT itself performs moderately well on this problem). Perhaps the most important concept that it learned was to avoid using the add-cascade-node operator, which provides little utility on this problem (as witnessed by the low performance of Cascade-NEAT). The additional overhead of optimizing the many connections introduced by this operator outweigh the benefits of being able to isolate local regions of the input space. However, because SNAP-NEAT does not learn to de-emphasize the RBF operator to the same extent as it does the cascade operator, its performance remains slightly below that of NEAT. Determining how to further improve the accuracy of SNAP-NEAT's operator evaluations is an important direction for future work that will be discussed further in Section V-B.

### *E. Half-field Soccer*

The results presented above provide evidence that SNAP-NEAT can discover which operators work best for problems that are fractured, like point classification, multiplexer, and concentric spirals, as well as for reactive control problems like pole-balancing. These test problems were chosen primarily because they are easy to understand and analyze. An interesting question is to what extent do the same conclusions apply to "real" fractured high-level decision problems that may include elements of both. The next section addresses this question by evaluating how well NEAT, RBF-NEAT, Cascade-NEAT, and SNAP-NEAT perform in the challenging reinforcement learning problem of half-field soccer [25]. This problem is interesting because it is a control problem with significant fracture.

The version of half-field soccer used in this paper features five offenders, five defenders, and a ball. A game starts with a random configuration of players on a rectangular field, as shown in Figure 20. One of the defenders is designated as the "goalie", and is tasked with defending a goal on the right side



(a) (b)

Fig. 20. (a) An example configuration of the half-field soccer domain, where five offenders (darker players) attempt to score goals on five defenders (lighter players with crosses). (b) An illustration of which actions would be successful for an offender with the ball at various points on the field, given a configuration shown in (a) for the other players. Each color represents one of the $2^6$ subsets of actions (holding the ball, shooting on goal, or passing to one of four teammates) that, if executed, would not immediately result in the end of an episode. Deciding which actions to use is a difficult high-level control problem that requires modeling a fractured decision space.

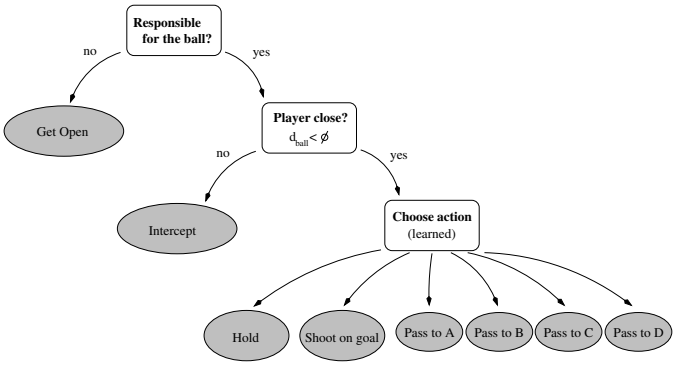



Fig. 21. The decision tree used to control the offensive players in the half-field soccer problem. Most of the behaviors are simple and are therefore hand-coded. However, the decision most crucial for the game (i.e. which one of the several possible actions to perform with the ball) needs to be learned.

of the field. The other defenders follow a hand-coded behavior designed to cover the field, prevent goals, and intercept the ball from the offending team.

The offenders are controlled by a hierarchy of hand-coded and learned behaviors (Figure 21). Their objective is to gain control of the ball, keep it away from the defenders, and score a goal. When a game starts, the offensive player nearest to the ball is designated as responsible for the ball. If this player is not close enough to the ball, it executes a pre-existing intercept behavior in an effort to get control of the ball. The other offensive players not responsible for the ball execute a pre-existing get-open behavior, designed to put them in good positions to both receive passes and to score goals.

However, when the responsible offender has control of the ball (defined by being within $\phi$ meters of the ball) it must choose between pre-existing behaviors of holding the ball, kicking the ball at the goal, or attempting a pass to one of its four teammates. The goal of learning is to make the appropriate decision given the state of the game at this point. This decision is both difficult and crucial for the game, making it a good test for learning algorithms [25, 53, 61, 52].

To make this decision, the network controlling the responsible offender receives 14 continuous inputs (Figure 22). The

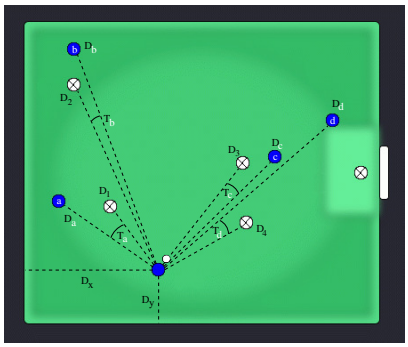


Fig. 22. A graphical depiction of the 14 state variables that an offensive player observes when making decisions in the half-field soccer problem. The inputs represent position on the field as well as distances and angles between teammates and opponents, normalized into the range $[0, 1]$.

first two inputs describe the player's position on the field. The network also receives three inputs for each of its four teammates: the distance to that teammate, the angle between that teammate and the nearest defender, and the distance to that nearest defender. All angles and distances are normalized to the range $[0, 1]$. The network has one output for each possible action (hold, shoot, or pass to one of the four teammates). The output with the highest activation is interpreted as the offender's action.

If the offender chooses to pass, the teammate receiving the pass is designated the new responsible offender. After initiating the pass, the original offender begins executing the get-open behavior.

Each network was evaluated in $\tau = 50$ different randomly chosen initial configurations of defenders and offenders. In each configuration, the ball is initially placed near one of the offensive players. Each of the players executes the appropriate hand-coded behavior. When the player responsible for the ball needs to choose between holding, shooting, and passing, the current network is used to select an action. The game is allowed to proceed until a goal is scored, a timeout is reached (after 1000 timesteps), the ball goes out of bounds, or a defender achieves control of the ball (by getting within $\phi$ meters of it). The score for a single game is the number of timesteps that the game takes, or, if a goal is scored, a fixed reward of 10,000. The overall score for the network is the sum of the scores for all $\tau$ games.

In order to evaluate the performance of the NEAT-related algorithms, several other learning methods that have shown promise on domains like half-field soccer were also evaluated. The first one is the standard reinforcement learning approach known as SARSA, which was the best learning approach in the original half-field soccer study [25]. This type of classic reinforcement learning approach has been shown to work well on challenging problems like keepaway and half-field soccer [53]. The version used for this comparison employs the same system of shared updates and parameter settings described by Kalyanakrishnan et al. [25], which was found to offer better performance than the baseline SARSA approach. Similarly, a CMAC function approximator was used to model the value function during learning, because it also was shown

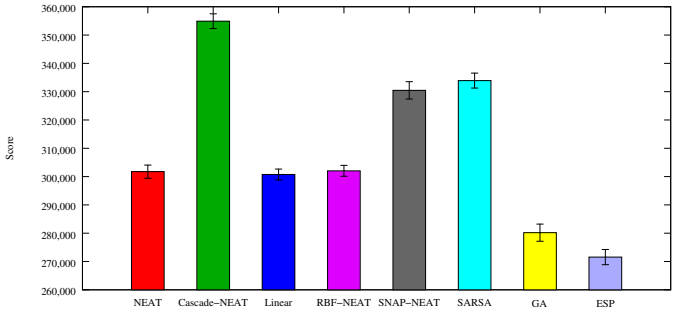



Fig. 23. A comparison of several learning algorithms on the half-field soccer problem. Cascade-NEAT is well suited for this problem and results in the best performance on this problem to date ($p > 0.95$). SNAP-NEAT is close behind, statistically similar to SARSA+CMAC ($p > 0.95$). The results provide evidence that combining NEAT with the ability to model local decision regions is a powerful approach for learning high-level control.

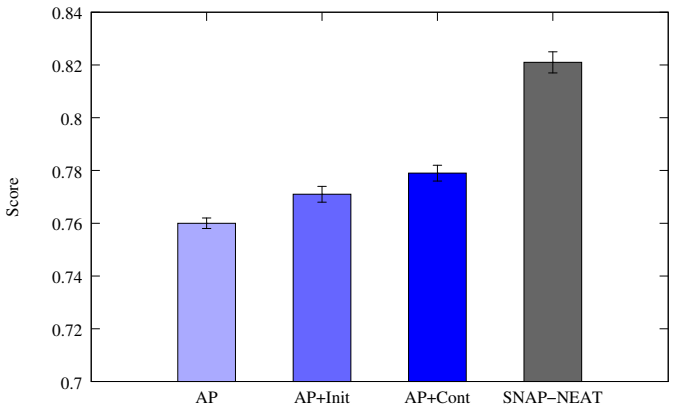



Fig. 24. A comparison of the different versions of Adaptive Pursuit and SNAP-NEAT on the most difficult version of the multiplexer problem from Section IV-B. Initialization periods and continuous updates both increase performance over the baseline Adaptive Pursuit. However, SNAP-NEAT performs much better, demonstrating that these extensions leverage eachother.

to generate the best results in previous work.

In addition to the NEAT variants, the ESP neuroevolution algorithm [15, 13] was evaluated on this problem. ESP has been shown to be effective in the past at generating solutions for non-linear control tasks such as rocket stabilization [16], often outperforming other reinforcement learning approaches. Since ESP relies on a fixed network topology to be chosen a priori by the experimenter, several different recurrent and non-recurrent network topologies were examined. The best approach ended up using a network with five hidden nodes with fully recurrent connections and the default parameter settings described in [15].

A third comparison was performed with a hand-coded policy optimized by a vanilla genetic algorithm. This policy is based on linear combinations of evolved parameters to make decisions about the chances of success for shooting, holding, and passing. The genetic algorithm used to evolve these parameters was the same algorithm that was used to optimize the weights of networks in the NEAT variants.

Performance of each algorithm was evaluated over 50 different start states. Figure 23 shows the scores for NEAT, Cascade-NEAT, RBF-NEAT, SARSA, ESP, the hand-coded/GA, and a linear baseline version of NEAT (which optimized a fixed topology consisting of a single layer of weights with no complexification operators), averaged over 100 runs.

NEAT is able to do reasonably well on this problem, outperforming ESP and the hand-coded/GA approaches. SNAP-NEAT performs statistically as well as the SARSA+CMAC approach. However, Cascade-NEAT generates the highest level of performance by a clear margin. These results suggest that combining NEAT with an ability to model local decision regions is a promising approach for learning high-level control. They also show that such control can be learned by a general method that works both with high and low fracture.

It should be noted that because an optimal solution is not known for either the pole-balancing or half-field soccer problems, the precise level of fracture for these problems is also unknown. However, the empirical results show that these problems are fundamentally different: NEAT performs best on pole-balancing, whereas Cascade-NEAT performs best on half-field soccer. One explanation that fits this data is that the high-level soccer task is fractured, whereas the reactive pole-balancing problem is not. But regardless of whether that explanation is correct, the main result that this paper presents is that SNAP-NEAT is able to intelligently combine the strengths of both NEAT and Cascade-NEAT to perform well on both problems.

### *F. Evaluation of Operator Selection*

Recall that SNAP-NEAT augments the baseline Adaptive Pursuit algorithm with two modifications: continuous evaluations and a period of initial estimation. These changes make intuitive sense, and make it easier to integrate Adaptive Pursuit with NEAT. However, it is worthwhile to examine how useful these modifications to the baseline Adaptive Pursuit algorithm actually are.

The multiplexer tasks described in Section IV-B represent a spectrum of fractured problems suitable for algorithms like RBF-NEAT and Cascade-NEAT. Figure 24 compares the performance of SNAP-NEAT to three versions of Adaptive Pursuit. The first version (labeled "AP") is baseline Adaptive Pursuit with no modifications. The second two versions (labeled "AP+Init" and "AP+Cont") represent Adaptive Pursuit augmented with either the initialization period or continuous evaluation modification described above. The main result is that SNAP-NEAT outperforms Adaptive Pursuit by a large margin. That is, the performance of Adaptive Pursuit can be significantly increased by including continuous updates and an initialization period. Individually, the two modifications to Adaptive Pursuit offer modest improvements in performance.

The learned probabilities for this multiplexer problem are shown in Figure 25. The baseline Adaptive Pursuit algorithm has difficulty in favoring the add-cascade-node operator, which is most useful for this problem. When modified to include continuous evaluations, Adaptive Pursuit makes much better use of the cascade mutation. However, SNAP-NEAT favors the add-cascade-node operator even more heavily, and as a result does very well on this problem.

Figure 26 revisits the non-Markov version of the double pole-balancing problem described in Section 3, where

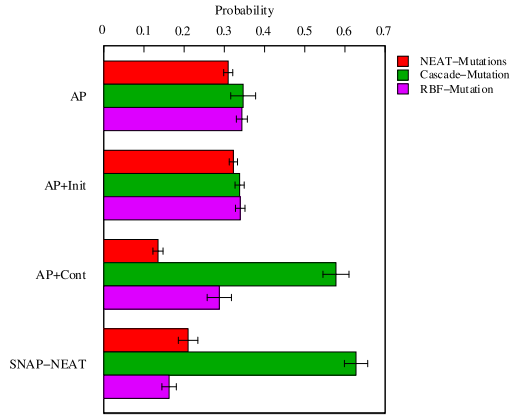



Fig. 25. The learned operator distributions for the Adaptive Pursuit variants and SNAP-NEAT for the most challenging multiplexer problem. When modified to include continuous evaluations, Adaptive Pursuit learns to rely on the add-cascade-node operator. However, SNAP-NEAT also achieves this effect while maintaining a better mix of the other two operators, giving it a higher performance. This result shows that when used together, continuous evaluations and an initial evaluation period are important in estimating operator values accurately.

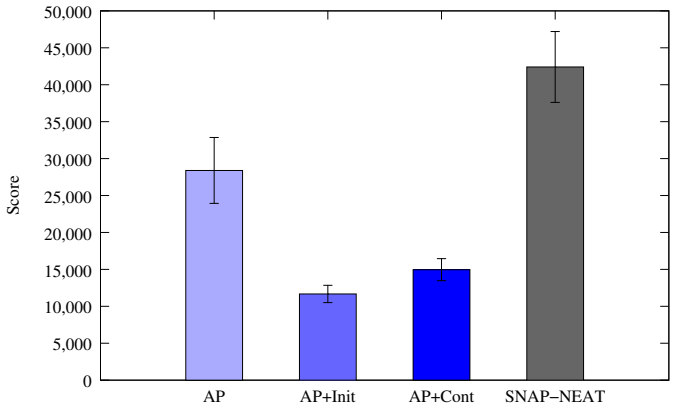



Fig. 26. A comparison of three different versions of Adaptive Pursuit and SNAP-NEAT on the non-Markov double pole-balancing problem. Interestingly, both continuous updates and the initialization period decrease the performance of Adaptive Pursuit when used alone. However, their combination in SNAP-NEAT does very well, suggesting that they work synergistically.

NEAT mutations were found most useful. The results show that SNAP-NEAT out-performs the standard Adaptive Pursuit algorithm and that interestingly enough, either of these to modifications by themselves actually decreases performance, suggesting that a synergy exists between them. A better understanding of this interaction is an important area for future work, and is discussed in Section V-A.

Figure 27 compares the operator probabilities for SNAP-NEAT and the Adaptive Pursuit algorithms. The differences in the learned operator probabilities are small, but significant. In particular, the Adaptive Pursuit algorithms do not learn to avoid the add-cascade-node mutation. Since the Adaptive Pursuit methods perform poorly compared to SNAP-NEAT, it is reasonable to conclude that their distribution of operator probabilities is not appropriate for the pole-balancing problem. On the other hand, SNAP-NEAT learns to suppress the add-cascade-node mutation, instead favoring the NEAT and RBF-NEAT mutations. Thus, using both continuous updates and a

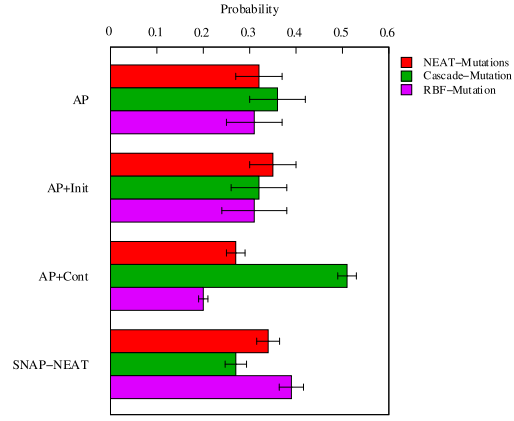



Fig. 27. A comparison of the learned probabilities for three versions of Adaptive Pursuit and SNAP-NEAT on the double pole-balancing problem. SNAP-NEAT is able to de-emphasize the add-cascade-node mutation as needed for this problem.

period of initial estimation allows SNAP-NEAT to discover this distribution of operator probabilities and outperform the Adaptive Pursuit algorithms significantly.

The results in these two example domains demonstrate that SNAP-NEAT utilizes both continuous evaluations and an initialization period to learn to favor the appropriate operators for a given problem, whether it be fractured or unfractured.

## V. Discussion and Future Work

The results described above show that SNAP-NEAT is a good way to combine the strengths of NEAT, RBF-NEAT, and Cascade-NEAT into a single algorithm that is effective on both reactive control and high-level strategy problems. This section discusses these results and describes several avenues for future work.

### A. Extending Adaptive Pursuit

The generic Adaptive Pursuit algorithm described by Theirens [57] makes several assumptions about the nature of the learning process that are not consistent with the ideas behind NEAT. In order to incorporate Adaptive Pursuit into NEAT effectively, continuous updates and initial estimation were added. On problems like the multiplexer, each of these extensions individually provide a moderate increase in performance over the baseline Adaptive Pursuit algorithm (Figure 24). Experiments in other domains yielded similar results. On pole-balancing, however, each extension alone actually decreases performance; each works well only when combined with the other.

These results show that there is a synergy between continuous updates and initial estimation. On most problems, the net effect is larger than the sum of their individual contributions. To some extent, this result is intuitive: When the operator values are initialized accurately, the best operators tend to be chosen. If the correct operators offer more consistent feedback than the worst operators, continuous updates will pull operator values in the right direction. In contrast, without accurate initialization, a poor choice of operators could result in noisy

evaluations. Using continuous updates in this case could make the algorithm sensitive to noise early in the learning process, and run the risk of pulling operator values away from the correct distribution. On the other hand, using only initial estimation and avoiding continuous updates could set the algorithm on the right path initially, but make it too slow at responding to changing operator values.

However, it is still surprising that each extension individually could decrease performance, as they do in pole-balancing. It is possible that the dynamics of this problem — which are different from those of fractured problems, since they favor the NEAT algorithm — make these individual contributions more dependent on each other. It is also possible that without a period of initial estimation, the continuous updates might use noisy data to change operator values too quickly, which could cause learning to diverge. It is less clear why initial estimation would only work when coupled with continuous updates. Further investigation of the interaction between these two modifications is an interesting direction for future work.

### B. Evaluating SNAP-NEAT

Section III introduced SNAP-NEAT as an example of adaptive operator selection. This class of algorithms explores how to employ a set of operators best during the learning process, usually making few or no assumptions about the nature of those operators. The operator selection mechanisms are relatively independent of the actual operators available, perhaps with the exception that performance could suffer from poor sampling if the number of operators becomes too large.

One interesting avenue for future work is to examine the role of operators other than add-node, add-link, add-cascade-node, and add-RBF-node. There are many different types of network topologies that have been explored in the neural network literature: networks with varying numbers of hidden layers, with or without recurrency, receptive fields that model those found in biology, etc. Also of interest are various fixed-topology approaches, like the multiple neuron population approach used by ESP [16]. A large array of operators inspired by these various topologies and organizational principles could provide an excellent base for an algorithm like SNAP-NEAT, allowing it to be applied to a broad spectrum of problems.

It is also interesting that on several different problems (e.g. concentric spirals) SNAP-NEAT is able to actually outperform its constituent algorithms. This result suggests that the best strategy for some problems involves the application of multiple operators. Since SNAP-NEAT (like Adaptive Pursuit on which it is based) is designed to heavily exploit the best performing operator, it may be possible to improve SNAP-NEAT's performance by allowing it to explore combinations of multiple operators more easily. Determining what kinds of problems might benefit from such a mix is also an interesting and challenging avenue for future work.

As demonstrated on the non-Markov pole balancing problem (Figure 19), SNAP-NEAT does not always discover the best operators for a given problem. Improving the accuracy of SNAP-NEAT's operator value estimation is thus one way to improve it in the future. One possibility is to run multiple independent learning instances, each of which has a fixed association with one or more operators. By avoiding the application of different operators in succession, the individual merit of an operator or set of operators may be more clear. Incorporating ideas such as this into SNAP-NEAT's operator estimation is an interesting direction for future work.

### C. Extending Network Construction

The results in this paper suggest that RBF-NEAT works best in low-dimensional settings. This result is understandable — as the number of inputs increases, the curse of dimensionality makes it increasingly difficult to set all of the parameters correctly for each basis function. This limitation suggests that a better method of incorporating basis functions into a constructive algorithm would be to situate those basis nodes on top of the evolved network structure. The lower levels of such a network can be thought of as transforming the input into a high-level representation, similar to the kernel transformation used by support vector machines [4]. The high-level representation is likely to be of smaller dimensionality than the original representation and basis nodes operating at this level may be effective at selecting useful features. Determining how to evolve such a multi-stage network effectively is an interesting direction for future work.

In addition to the cascade architecture and basis functions, there are other useful ideas from supervised machine learning that could be applied to neuroevolution. One such idea is to use an initial unsupervised training period to initialize a large network, similar to the initial step of training that happens in deep learning [20, 3, 32]. Using unsupervised learning to provide a good starting point for the search process could have a dramatic effect on learning performance. Conversely, the adaptive pursuit algorithm on which SNAP-NEAT is based is a general-purpose approach for choosing intelligently between multiple mutation operators, and is applicable to many different types of evolutionary algorithms. The generality of this approach suggests that it could be used to improve the performance of a wide variety of evolutionary algorithms.

### D. Extending Evaluation and Applications

The data presented in this paper were drawn from a variety of different problems, ranging from simple but easy-to-analyze domains to challenging high-level strategy problems. The goal in examining such a broad spectrum was to obtain solid empirical evidence in support of the hypothesis that a single algorithm can work well across a variety of problems without explicit knowledge.

However, the current analysis only scratches the surface. There are countless challenging and interesting problems that learning algorithms currently can not solve, and the exploration of any of these problems could yield valuable insight into the strengths and weaknesses of algorithms like SNAP-NEAT. There are ways in which a problem might be considered difficult other than fracture; empirical evaluation like the kind presented in this paper is one of the most direct ways to identify these axes of difficulty and to determine which problems feature them.

In particular, it would be useful to evaluate the lessons learned in this paper on other high-level reinforcement learning problems. One potential candidate is a multi-agent vehicle control task, such as that examined in [46]. Previous work showed that algorithms like NEAT are effective at generating low-level control behaviors, like efficiently steering a car through S-curves on a track. Evolving higher-level behavior to reason about opponents or race strategy has proven difficult, but may be possible with algorithms like Cascade-NEAT, RBF-NEAT, and SNAP-NEAT.

## VI. Conclusion

While previous neuroevolution algorithms such as NEAT, RBF-NEAT, and Cascade-NEAT have been shown to work well on specific classes of problems, their performance can suffer when they are applied to certain types of new domains. The results in this paper show how one approach to neuroevolution, SNAP-NEAT, can be successfully used to solve a variety of different types of problems without a priori domain knowledge. SNAP-NEAT is a hybrid approach that uses a modified version of Adaptive Pursuit to combine the strengths of NEAT, RBF-NEAT, and Cascade-NEAT. This approach is evaluated empirically on a set of problems, ranging from reactive control to high-level strategy. The results show that SNAP-NEAT is able to intelligently select the best operators for the problem at hand, allowing it to change how it behaves depending on the type of problem that it faces. This kind of general approach is crucial in encouraging the broad application of AI techniques to real-world problems, where domain expertise is not always available.

## References


[1] H. J. C. Barbosa and A. M. Sa. On adaptive operator probabilities in real coded genetic algorithms. In *In Proc. XX International Conference of the Chilean Computer Science Society*, 2000.

[2] A. Barron, J. Rissanen, and B. Yu. The minimum description length principle in coding and modeling. *IEEE Trans. Information Theory*, 44(6):2743–2760, 1998.

[3] Yoshua Bengio. Learning deep architectures for ai. Technical Report 1312, Dept. IRO, Universite de Montreal, 2007.

[4] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In *COLT '92: Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152, New York, NY, USA, 1992. ACM.

[5] G.J. Chaitin. A theory of program size formally identical to information theory. *Journal of the ACM*, 22:329–340, 1975.

[6] Luis DaCosta, Alvaro Fialho, Marc Schoenauer, and Michle Sebag. Adaptive operator selection with dynamic multi-armed bandits. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 913–920, 2008.

[7] L. Davis. Adapting operator probabilities in genetic algorithms. In *Proc. 3rd International Conference on Genetic Algorithms*, pages 61–69, 1989.

[8] J Schmidhuber F. Gomez and R. Miikkulainen. Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research*, 9:937–965, 2008.

[9] S. E. Fahlman and C. Lebiere. The cascade-correlation learning architecture. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 524–532, Denver 1989, 1990. Morgan Kaufmann, San Mateo.

[10] C. Ferreira. *Gene Expression Programming: Mathematical Modeling by an Artificial Intelligence*. 2002.

[11] David Goldberg. Probability matching, the magnitude of reinforcement, and classifier system bidding. 5(4):407–426, 1990.

[12] David E. Goldberg and J. Richardson. Genetic algorithms with sharing for multimodal function optimization. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 148–154, 1987.

[13] F. Gomez and R. Miikkulainen. Incremental evolution of complex general behavior, 1997.

[14] F. Gomez and R. Miikkulainen. Solving non-markovian control tasks with neuroevolution. In *In Proceedings of the 16th International Joint Conference on Artificial Intelligence*, 1999.

[15] Faustino Gomez. *Robust Non-Linear Control Through Neuroevolution*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 2003.

[16] Faustino Gomez, Juergen Schmidhuber, and Risto Miikkulainen. Efficient non-linear control through neuroevolution. In *Proceedings of the European Conference on Machine Learning (ECML-06, Berlin)*, 2006.

[17] Frederic Gruau, Darrell Whitley, and Larry Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 81–89. MIT Press, 1996.

[18] H.M. Gutmann. A radial basis function method for global optimization. *Journal of Global Optimization*, 19:201–227, 2001.

[19] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolutionary strategies. *Evolutionary Computation*, 9:159–195, 2001.

[20] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[21] K. M. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, pages 359–366, 1989.

[22] C. Igel. Neuroevolution for reinforcement learning using evolution strategies. In *Congress on Evolutionary Computation 2003 (CEC 2003)*, 2003.

[23] B. A. Julstrom. What have you done for me lately? adapting operator probabilities in a steady-state genetic algorithm. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 81–87, 1995.

[24] Leslie P. Kaelbling. *Learning in Embedded Systems*. MIT Press, 1993.

[25] Shivaram Kalyanakrishnan, Yaxin Liu, and Peter Stone. Half field offense in RoboCup soccer: A multiagent reinforcement learning case study. In Gerhard Lakemeyer, Elizabeth Sklar, Domenico Sorenti, and Tomoichi Takahashi, editors, *RoboCup-2006: Robot Soccer World Cup X*, pages 72–85, Berlin, 2007. Springer Verlag.

[26] Nate Kohl. *Learning in Fractured Problems with Constructive Neural Network Algorithms*. PhD thesis, Department of Computer Sciences, University of Texas at Austin, 2009.

[27] Nate Kohl and Risto Miikkulainen. Evolving neural networks for fractured domains. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1405–1412. July 2008.

[28] Nate Kohl and Risto Miikkulainen. Evolving neural networks for strategic decision-making problems. *Neural Networks*, 22:326–337, 2009. Special issue on Goal-Directed Neural Systems.

[29] Nate Kohl, Kenneth Stanley, Risto Miikkulainen, Michael Samples, and Rini Sherony. Evolving a real-world vehicle warning system. In *Proceedings of the Genetic and Evolutionary Computation Conference 2006*, pages 1681–1688, July 2006.

[30] A.N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems of Information Transmission*, 1:4–7, 1965.

[31] R.M. Kretchmar and C. Anderson. Comparison of cmacs and radial basis functions for local function approximators in reinforcement learning. In *Proceedings of the International Conference on Neural Networks*, 1997.

[32] Y. LeCun and Y Bengio. Scaling learning algorithms towards ai. *Large-Scale Kernel Machines*, 2007.

[33] J. Li and T. Duckett. Q-learning with a growing rbf network for behavior learning in mobile robotics. In *Proceedings of the Sixth IASTED International Conference on Robotics and Applications*, 2005.

[34] Jun Li, T. Martinez-Maron, A. Lilienthal, and T. Duckett. Q-ran: A constructive reinforcement learning approach for robot behavior learning. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robot and System*, 2006.

[35] M. Li and P. Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, 1993.

[36] J. M. Maciejowskia. Model discrimination using an algorithmic information criterion. *Automatica*, 15:579–593, 1979.

[37] J. Moody and C. J. Darken. Fast learning in networks of locally tuned processing units. *Neural Computation*, 1: 281–294, 1989.

[38] D. E. Moriarty and R. Miikkulainen. Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22:11–32, 1996.

[39] Michael O'Mahony. *Sensory Evaluation of Food: Statistical Methods and Procedures*. 1986.

[40] J. Park and I. W. Sandberg. Universal approximation using radial-basis-function networks. *Neural Computation*, 3:246–257, 1991.

[41] T. Peterson and R. Sun. An rbf network alternative for a hybrid architecture. In *IEEE International Joint Conference on Neural Networks*, volume 1, pages 768–773, 1998.

[42] John Platt. A resource-allocating network for function interpolation. *Neural Computation*, 3(2):213–225, 1991.

[43] Mitchell A. Potter and Kenneth A. De Jong. Cooperative coeveolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation*, 8(1):1–29, 2000.

[44] Joseph Reisinger, Erkin Bahceci, Igor Karpov, and Risto Miikkulainen. Coevolving strategies for general game playing. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2007.

[45] N. Saravanan and D. B. Fogel. Evolving neural control systems. *IEEE Expert*, pages 23–27, 1995.

[46] Kenneth Stanley, Nate Kohl, Rini Sherony, and Risto Miikkulainen. Neuroevolution of an automobile crash warning system. In *Proceedings of the Genetic and Evolutionary Computation Conference 2005*, pages 1977–1984, 2005.

[47] Kenneth O. Stanley. *Efficient Evolution of Neural Networks Through Complexification*. PhD thesis, Department of Computer Sciences, University of Texas at Austin, 2003.

[48] Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen. Real-time neuroevolution in the NERO video game. *IEEE Transactions on Evolutionary Computation*, 9(6):653–668, 2005.

[49] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2), 2002.

[50] Kenneth O. Stanley and Risto Miikkulainen. Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence Research*, 21:63–100, 2004.

[51] Kenneth O. Stanley and Risto Miikkulainen. Evolving a roving eye for go. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 2004.

[52] Peter Stone, Gregory Kuhlmann, Matthew E. Taylor, and Yaxin Liu. Keepaway soccer: From machine learning testbed to benchmark. In Itsuki Noda, Adam Jacoff, Ansgar Bredenfeld, and Yasutake Takahashi, editors, *RoboCup-2005: Robot Soccer World Cup IX*, volume 4020, pages 93–105. Springer Verlag, Berlin, 2006.

[53] Peter Stone, Richard S. Sutton, and Gregory Kuhlmann. Reinforcement learning for RoboCup-soccer keepaway. *Adaptive Behavior*, 2005.

[54] Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, pages 1038–1044, 1996.

[55] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning I: Introduction*. 1998.

[56] Matthew Taylor, Shimon Whiteson, and Peter Stone. Comparing evolutionary and temporal difference methods for reinforcement learning. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1321–28, July 2006.

[57] Dirk Thierens. An adaptive pursuit strategy for allocating

operator probabilities. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1539–1546, 2005.

[58] A. Tuson and P. Ross. Adapting operator settings in genetic algorithms. *Evolutionary Computation*, 6(2):161–184, 1998.

[59] V. Vapnik and A. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16:264–280, 1971.

[60] C.J.C.H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

[61] Shimon Whiteson, Nate Kohl, Risto Miikkulainen, and Peter Stone. Evolving keepaway soccer players through task decomposition. *Machine Learning*, 59:5–30, May 2005.

[62] D. Whitley, S. Dominic, R. Das, and C. W. Anderson. Genetic reinforcement learning for neurocontrol problems. *Machine Learning*, 13:259–284, 1993.

[63] A. Wieland. Evolving neural network controllers for unstable systems. In *In Proceedings of the International Joint Conference on Neural Networks*, pages 667–673, 1991.

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