Neural Networks 22 (2009) 326–337

Contents lists available at ScienceDirect

Neural Networks

journal homepage: www.elsevier.com/locate/neunet

2009 Special Issue

Evolving neural networks for strategic decision-making problems

Nate Kohl *, Risto Miikkulainen

Department of Computer Sciences, The University of Texas at Austin, 1 University Station C0500, Austin, TX, United States

ARTICLE INFO

Article history:
Received 22 December 2008
Received in revised form 27 February 2009
Accepted 13 March 2009

Keywords:
Neuroevolution
Fracture
NEAT
Cascade correlation
RBF networks

ABSTRACT

Evolution of neural networks, or neuroevolution, has been a successful approach to many low-level control problems such as pole balancing, vehicle control, and collision warning. However, certain types of problems – such as those involving strategic decision-making – have remained difficult for neuroevolution to solve. This paper evaluates the hypothesis that such problems are difficult because they are fractured: The correct action varies discontinuously as the agent moves from state to state. A method for measuring fracture using the concept of function variation is proposed and, based on this concept, two methods for dealing with fracture are examined: neurons with local receptive fields, and refinement based on a cascaded network architecture. Experiments in several benchmark domains are performed to evaluate how different levels of fracture affect the performance of neuroevolution methods, demonstrating that these two modifications improve performance significantly. These results form a promising starting point for expanding neuroevolution to strategic tasks.

1. Introduction

The process of evolving neural networks using genetic algorithms, or neuroevolution, is a promising new approach to solving reinforcement learning problems. While the traditional method of solving such problems involves the use of temporal difference methods to estimate a value function, neuroevolution instead relies on policy search to build a neural network that directly maps states to actions. This approach has proved to be useful in 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 (Gomez, Schmidhuber, & Miikkulainen, 2006; Kohl, Stanley, Miikkulainen, Samples, & Sherony, 2006; Reisinger, Bahceci, Karpov, & Miikkulainen, 2007; Stanley, Bryant, & Miikkulainen, 2005; Stanley & Miikkulainen, 2002, 2004a, 2004b). However, despite its efficacy on such low-level control problems, other types of problems such as concentric spirals, multiplexer, and high-level decision making in general have remained difficult for neuroevolution algorithms to solve. A better understanding of why neuroevolution works well on some problems – but not others – would be useful in designing the next generation of neuroevolution algorithms.

Most of the early work in neuroevolution was based on fixed-topology methods (Gomez & Miikkulainen, 1999; Moriarty & Miikkulainen, 1996; Saravanan & Fogel, 1995; Whitley, Dominic, Das, & Anderson, 1993; Wieland, 1991). 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 (Hornik, Stinchcombe, & White, 1989). 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. 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 have difficulty representing solutions beyond a certain level of complexity.

Neuroevolution algorithms that evolve both topology and weights (so-called constructive algorithms) were created to address this problem (Angeline, Saunders, & Pollack, 1993; Gruau, Whitley, & Pyeatt, 1996; Yao, 1999). While this approach met with some success, it struggled to effectively evolve both topology and weights simultaneously. One problem was competing conventions, wherein structures that evolve independently in different networks must be joined together meaningfully in a crossover operation. This difficulty was recently addressed with the introduction of historical markings, which provided a principled method of identifying homologous sections of two different networks (Stanley & Miikkulainen, 2002). This improvement allowed neuroevolution algorithms to compete with standard reinforcement learning algorithms on a variety of problems.

However, certain types of problems – such as high-level decision tasks – still remain difficult for neuroevolution algorithms to solve. This paper presents the *fractured problem hypothesis* as a possible explanation for this issue. By definition, fractured

* Corresponding author.
E-mail addresses: nate@cs.utexas.edu (N. Kohl), risto@cs.utexas.edu (R. Miikkulainen).

0893-6080/$ – see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.neunet.2009.03.001

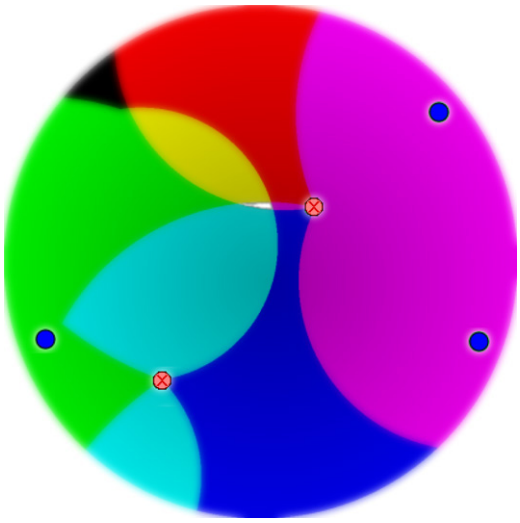

Fig. 1. The fractured decision space for one configuration of teammates and opponents in the keepaway soccer task. The color at each point represents the set of available receivers for a pass from that point. In order to perform well in this task, the player must be able to model a fractured decision space. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

problems have a highly discontinuous mapping between states and optimal actions. As an agent moves from state to state, the best action that the agent can take changes frequently and abruptly. In contrast, the optimal actions for a non-fractured problem change slowly and continuously.

Many challenging supervised learning tasks are fractured, such as multiplexer and concentric spirals. Importantly for reinforcement learning, high-level decision tasks where an agent must choose between several sub-behaviors are often fractured as well. For example, Fig. 1 shows the possible actions that a hand-coded keepaway soccer player considers when making a passing decision during a game. The three teammates that could receive the pass are indicated by darker circles; two opponents who might intercept the pass are indicated by lighter circles with crosses. The color at each point p represents the set of possible teammates that could successfully receive a pass if the player were at point p with the ball. As the player moves around the field with the ball, the set of possible teammates open for a pass changes frequently and discontinuously, giving this task a fractured quality. In addition, the nature of the fracture changes as both teammates and opponents move. The fractured problem hypothesis posits that neuroevolution performs poorly on such fractured problems because the evolved neural networks have difficulty representing such abrupt decision boundaries.

The first section of this paper introduces a quantitative definition of fracture built on the mathematical concept of function variation. Next, related work on fracture in machine learning is reviewed and two modified learning algorithms designed to solve fractured problems are proposed. These algorithms are empirically compared to a state-of-the-art constructive neuroevolution method called NEAT on five different fractured problems. The results confirm the fractured domain hypothesis, showing that standard neuroevolution techniques have difficulty performing well in fractured domains. The modified neuroevolution algorithms, however, perform much better, suggesting that neuroevolution can scale to high-level decision tasks as well.

2. Fractured problems

What makes problems like multiplexer, concentric spirals, and high-level decision tasks in general different from those on which other neuroevolution algorithms have done so well? This section proposes the hypothesis that these problems share a common property: They possess a "fractured" decision space, loosely defined as a *space where adjacent states require radically different actions* In this section, the concept of function variation is introduced as a way to more precisely quantify this idea. Section 5 will then describe several experiments to demonstrate that the difficulty neuroevolution has with fractured problems stems from an inability to generate networks with an appropriate amount of variation.

2.1. Measuring complexity

For many problems (such as the typical control or reinforcement learning benchmarks), the correct 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. For example, in Fig. 1, the left half of the state space in particular is quite fractured.

Clearly, the choice of state variables could change many aspects of a given problem, including the degree to which it is fractured. For example, solving the concentric spirals problem becomes much easier if the state space is represented in polar coordinates instead of Cartesian coordinates. For this work, a problem is considered a "black box" that already has associated states and actions. In other words, it is assumed that the definition of a problem includes a choice of inputs and outputs, and the goal of the agent is to learn given those constraints. Any definition of fracture then applies to the entire definition of the problem.

This definition of fracture, while intuitive, is not very precise. More formal definitions of complexity have been proposed for learning problems, including Minimum Description Length (Barron, Rissanen, & Yu, 1998; Chaitin, 1975), Kolmogorov complexity (Kolmogorov, 1965; Li & Vitanyi, 1993), and Vapnik–Chervonenkis (VC) dimension (Vapnik & Chervonenkis, 1971). Unfortunately, these metrics are often more suited to a theoretical analysis than they are to practical usage. For example, Kolmogorov complexity is a measure of complexity that depends on the computational resources required to specify an object – which sounds promising for measuring problem fracture – but it has been shown to be practically uncomputable (Maciejowskia, 1979).

An alternative way to measure fracture is to consider the degree to which solutions to a problem are fractured. VC dimension at first appears promising for this approach, since it describes the ability of a possible solution to "shatter" a set of randomly-labeled training examples into distinct groups. However, VC dimension is a general method for measuring the capabilities of a model, and does not apply to a specific problem. Furthermore, analyzing VC dimension of neural networks is difficult; while results exist for single-layer networks, it is much more difficult to analyze the networks with arbitrary (and possibly recurrent) connectivity that constructive neuroevolution algorithms generate (Mitchell, 1997).

A third possibility is described by Ho and Basu (2002), who surveyed a variety of complexity metrics for supervised classification problems and found a significant difference between random classification problems and those drawn from real-world datasets. In terms of measuring problem fracture, the most promising of these metrics is a gauge of the linearity of the decision boundary between two classes of data. However, these metrics are tied to a two-class supervised learning setting, which makes them less useful in a reinforcement learning setting, where the goal can involve learning a continuous mapping from states to actions.

Therefore, in order to measure fracture, a more direct approach is developed in this paper: measuring how much the actions of optimal policies for the problem change from state to state.

In a fractured problem, good policies repeatedly yield different actions as the agent moves from state to state. Compared to the alternatives described above, this definition of problem fracture is easy to compute, because it turns out to be surprisingly simple to measure how much policies change over a known and bounded area.

Of course, this definition of problem fracture explicitly ties fracture to optimal policies. Intuitively, a problem may be considered difficult if the optimal policy has this fractured property. However, some fractured problems might have relatively unfractured policies that are quite close to optimal. Any learning algorithm could perform quite well on these problems, regardless of the amount of fracture in optimal policies. One simplifying assumption made in this paper, therefore, is that there is a relatively smooth continuum in both score and fracture between poor policies and optimal policies. Several experiments presented below suggest that this assumption is likely to be true in many realistic problems.

Estimating problem fracture depends on measuring how the actions of optimal policies change from state to state. The next section describes how this measurement can be made by treating policies as functions and measuring how much the functions change using the concept of total variation.

2.2. Measuring variation of a function

The *total variation* of a function (Leonov, 1998; Vitushkin, 1955) measures how much a function (or policy) changes over a certain interval. This section provides a technical description of multidimensional variation (adapted from Leonov (1998)) followed by several illustrations of how variation can be computed.

Consider an N-dimensional rectangular parallelepiped $B = B_a^b = B_{b_1\ldots b_N}^{a_1\ldots a_N} = \{x \in \Re^N : a_i \leq x_i \leq b_i, a_i < b_i, i = 1, \ldots, N\}$ and a function over this parallelepiped, $z(x_1, \ldots, x_N)$, whose variation is to be measured. From Bochner (1959), Kamke (1956), Leonov (1998) and Shilov and Gurevich (1967), the N-dimensional quasivolume σ_N for z over a sub-parallelepiped B_α^β of B is defined as

$$\sigma_N(B_\alpha^\beta) = \sum_{v_1=0}^{1} \cdots \sum_{v_N=0}^{1} (-1)^{v_1+\cdots+v_N} z[x_1, \ldots, x_N], \tag{1}$$

where x_c is

$$x_c = \beta_c + v_c(\alpha_c - \beta_c).$$

Now consider a partitioning of B into a set of sub-parallelepipeds $\Pi = \{B_j\}_{j=1}^n$ where none of the individual sub-parallelepipeds B_j intersect, and $B_1 + \cdots + B_n = B$. Let P be the set of all such partitions for all n. The N-dimensional variation (or Vitali variation) of the function z in the parallelepiped B is

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

Next, consider all of B's m-dimensional coordinate faces $B_{i_1,\ldots,i_m}$ for $1 \leq m \leq N-1$ that pass through the point $a \in B$ and are parallel to the axes $x_{i_1}, \ldots, x_{i_m}$ where $1 \leq i_1 < \cdots < i_m \leq N$. For convenience, mark all of the m-dimensional faces of the form $B_{i_1,\ldots,i_m}$ by a number r ($1 \leq r \leq \binom{N}{m} = N_m$). Each such face will be denoted by $B_r^{(m)}$.

Definition. The *total variation of the function z in the parallelepiped* B is the number

$$V(z, B) = \sum_{m=1}^{N-1} \left\{ \sum_{r=1}^{N_m} V_m(z, B_r^{(m)}) \right\} + V_N(z, B). \tag{3}$$

Several illustrative examples of this variation calculation follow, starting with the one-dimensional case. The variation of a 1-d function $z(x_1)$ over the range $a_1 \leq x_1 \leq b_1$ is simply the sum of the absolute value of the differences between adjacent values of z between a_1 and b_1. When $N = 1$, the variation of z over the interval B, $V(z, B)$, effectively becomes $V_1(z, B)$, which is computed by the summation in Eq. (2). For example, Fig. 2 shows a function z that has been divided into five sections inside the interval $[a_1, b_1]$. The differences between adjacent points (each computed by Eq. (1) and shown as dotted lines in Fig. 2) would be added together to determine the variation for z on the parallelepiped $B = B_{a_1}^{b_1}$, which is just the 1-d interval $[a_1, b_1]$.

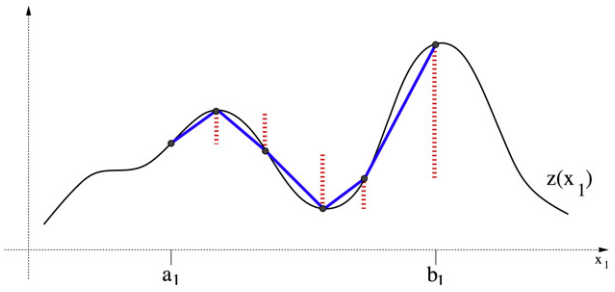

Fig. 2. An example of how the variation of a 1-d function is computed. The absolute value of the differences between adjacent points on the function (shown as dotted lines) over the interval $[a_1, b_1]$ are summed together to produce an estimate of the total variation.

In Fig. 2, the 1-d parallelepiped B (or the interval $[a_1, b_1]$) is divided into five sections by six points. Clearly, a different selection of points could produce a different estimate of variation. For example, if the variation calculation only used the first and last points in the interval, then the middle two "bumps" of the function would be skipped over, producing a lower variation. The choice of an appropriate set of points (referred to above as a partition Π of B) is dealt with in Eq. (2). To compute the $V_N(z, B)$, a partition Π of B should be chosen such that it maximizes $V_N(z, B)$. It is easy to see that as the discretization of the partition becomes increasingly fine, the variation will not decrease. In fact, when the discretization of the partition becomes infinitely small, the calculation of $V_N(z, B)$ in the 1-d case turns into

$$V_1(z, B_{a_1}^{b_1}) = \int_{a_1}^{b_1} |z(x)| \mathrm{d}x. \tag{4}$$

Infinitely-fine partitionings of B are fine for mathematicians, but practically speaking, computational resources will limit the degree to which it is possible to discretize B. For the work described here, the finest possible discretization $\hat{\Pi}$ of B is chosen given the limited computational resources available. This means that only one partition $\hat{\Pi}$ is considered, and the supremum in Eq. (2) is effectively ignored.

The computation of multidimensional variation is more involved than the 1-d case. To compute the variation for a 2-d function $z(x_1, x_2)$ on the parallelepiped $B = B_{a_1,a_2}^{b_1,b_2}$, three different terms are computed and summed together. Each of these terms is meant to measure the variation in a specific direction on B, and corresponds to a "face" of B (shown in Fig. 3):

- $B_1^{(1)}$: the first 1-d face of B, variation of z as x_1 changes;
- $B_2^{(1)}$: the second 1-d face of B, variation of z as x_2 changes; and
- B: the only 2-d face of B is B itself, variation of z as both x_1 and x_2 change.

To compute the variations for the two 1-d faces of B, $V_1(z, B_1^{(1)})$ and $V_1(z, B_2^{(1)})$, a calculation very similar to the one described above can be used: the variation is simply the sum of the absolute values of the differences between adjacent values of the function. Each difference between adjacent points α and β is

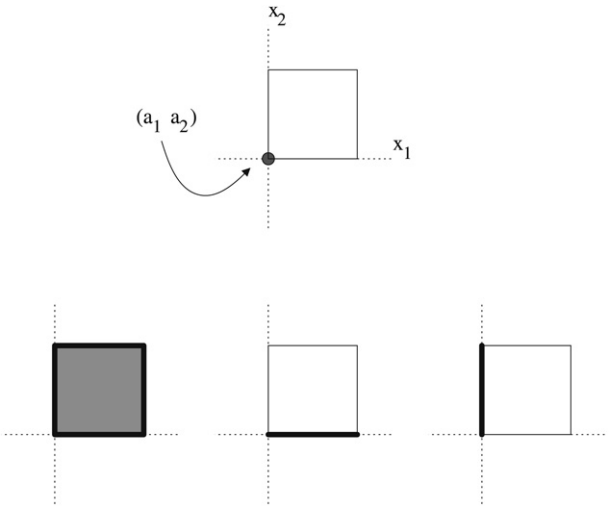

Fig. 3. The three faces (two 1-d faces and one 2-d face) of a 2-d parallelepiped that pass through the point (a_1, a_2). Measuring variation on each face is meant to capture how the function changes in different directions.

$\sigma_N(B_\alpha^\beta)$, represented by Eq. (1). The 2-d version of Eq. (1) involves measuring four points, instead of two. For example, when $N = 2$, the quasivolumes of the function z over the parallelepipeds $B_{\alpha_1,\alpha_2}^{\beta_1,\beta_2}$, $B_{\alpha_1}^{\beta_1}$, and $B_{\alpha_2}^{\beta_2}$ are

$$\sigma_2(B_{\alpha_1,\alpha_2}^{\beta_1,\beta_2}) = z(\beta_1, \beta_2) - z(\beta_1, \alpha_2) - z(\alpha_1, \beta_2) + z(\alpha_1, \alpha_2),$$
$$\sigma_1(B_{\alpha_1}^{\beta_1}) = z(\beta_1, a_2) - z(\alpha_1, a_2),$$
$$\sigma_1(B_{\alpha_2}^{\beta_2}) = z(a_1, \beta_2) - z(a_1, \alpha_2).$$

It should be noted that there are actually four 1-d faces of the 2-d parallelepiped B, but only two of the faces are used in this variation calculation, i.e. those that are on the "lower" edge of B (denoted by those faces that pass through the point $a \in B$).

The next section continues this discussion of function variation with a description of how total variation can be measured in the context of neuroevolution.

2.3. Measuring variation of a neural network

A neural network produced by a neuroevolution algorithm can be thought of as a function that maps states to actions. Because the variation calculation does not care what form the function takes – it only requires input and output pairs from the function – it is straightforward to calculate the variation of a neural network.

The first step is to select a parallelepiped P of the input space over which variation will be measured. In a reinforcement learning setting, it is frequently the case that the inputs have already been truncated or scaled to a certain range, effectively defining P. For example, an agent controlling a racecar might receive an angular input describing the location of the nearest opponent. This input can be scaled into the range $[-\pi, \pi]$, defining P for that dimension. Different dimensions of P can have different bounds.

The next step is to quantize P into some partition Π. As described above, the ideal partition $\Pi_{optimal}$ contains infinitely small slices of P. However, a finite amount of computation limits how finely P can be discretized. Practically speaking, P is quantized into $\hat{\Pi}$, which is the finest uniform discretization of P that is possible given the computational resources that are available.

The definition of P and $\hat{\Pi}$ determine a finite set of points from the input space. The output of the neural network is then measured and stored for each of these input points. After measuring these values, a series of summations over each possible combination of dimensions of the input space (described by Eqs. (1)–(3)) determines the variation of the network. For networks with multiple outputs, this work assumes that the total variation of the network is the average of the multiple independent variation calculations for each output. This assumption has proven effective for the experiments below; however an interesting avenue for future work involves closer examination of this assumption.

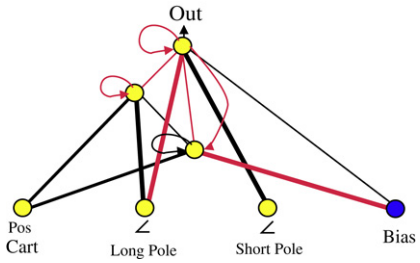

Fig. 4. A surprisingly small solution that was evolved by the NEAT neuroevolution to solve the non-Markov double pole balancing problem. NEAT was able to take advantage of recurrent connections to generate a parsimonious solution to this problem.

It should be noted that this definition of variation assumes that the network represents a function. In some applications of neuroevolution, evolved networks are not functions in the strictest sense; they map states to actions, but the experimenter does not reset node activation levels between successive states. Evolved networks used in this manner are less similar to state-action functions and more akin to dynamical systems with internal state over time that happen to pick actions. With such networks, it is not meaningful to simply query a network for its chosen action given a single state. Instead, the network must be evaluated over a set of states, starting from a specific initial state, while maintaining activation levels of individual nodes in the network across state transitions. This approach can be used to great advantage in non-Markov problems. For example, in the non-Markov pole-balancing task, an evolved recurrent network was found to use such recurrent connections to compute the derivative of the pole angle (Stanley & Miikkulainen, 2002). This information about the direction of pole movement proved useful in solving the task quickly with a small network (Fig. 4).

It is difficult to measure the variation of a network that is not as a function. Instead of simply querying the network for its output at a given state, the entire succession of states that lead up to the state in question must be queried in order—and it is still not clear that such an approach would yield appropriate values for a variation computation. Because of this restriction, this paper focuses on learning state-action mappings for Markov problems. Fortunately, there are many interesting problems that are Markov or that can be made Markov with additional state variables.

Even in Markovian tasks, the recurrent topologies that constructive neuroevolution algorithms produce may be useful. The activation process for these networks starts from a uniform unactivated state where only the input nodes have activation values. Values from the input nodes are propagated through the network until all output nodes have received some input value. The network is then activated κ times, where each activation allows values from the input nodes to propagate one level deeper into the network. The input nodes maintain their output over all κ activations. This repeated activation scheme allows recurrent connections and values delayed during propagation to affect the computation of an action for a state.

Using this procedure, it is possible to evaluate the variation of any neural network produced by neuroevolution in Markovian tasks. This calculation provides a quantitative description of the amount of fracture that a learning algorithm is capable of modeling for a given problem. By measuring the variation of good policies, this metric can also be used to estimate the difficulty of a given problem.

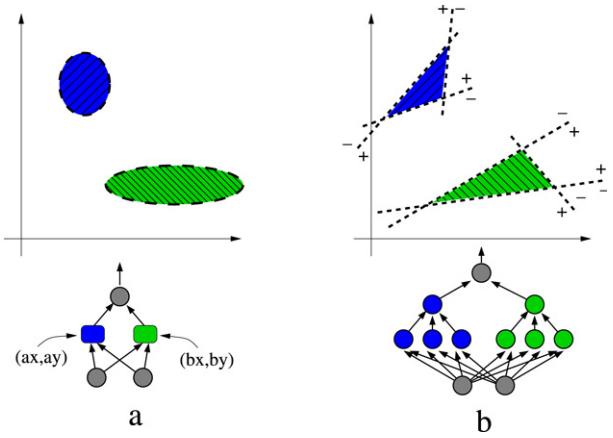

Fig. 5. A comparison of how (a) an RBF network and (b) a sigmoid-node network (b) might isolate two specific areas of a 2-d input space. The local functionality of the RBF network can identify comparable spaces using far fewer parameters.

The intuitive concept that fracture makes a problem difficult is familiar to the machine learning community. The next section describes previous approaches to solving fractured problems. These insights are then utilized in Section 4 to develop two new neuroevolution methods for such problems.

3. Related work

In order to perform well in a fractured problem, a learning algorithm must be able to generate representations that capture local features of the problem. For example, after the algorithm experiences a new state in the environment, it needs to associate a specific action for that state. If the problem is fractured, it may not be useful to generalize from the actions of nearby states. Furthermore, any large-scale changes the algorithm may attempt to make could disrupt the individual actions tailored for other states. Therefore, the algorithm must be able to make local changes to isolate that particular state from its neighbors and associate the correct action with it. This concept of localized change – as opposed to large-scale, global change – has also appeared before in many parts of the machine learning community, and will serve as the starting point for the proposed methods as well.

3.1. Supervised learning

One promising method for learning local features is radial basis function (RBF) networks (Gutmann, 2001; Moody & Darken, 1989; Park & Sandberg, 1991; Platt, 1991). RBF networks originated in the supervised learning community, and are usually described as neural networks with a single hidden layer of basis-function nodes. Each of these nodes computes a function (usually a Gaussian) of the inputs, and the output of the network is a linear combination of all of the basis nodes. RBF networks are usually trained in two stages: The locations and sizes of the basis functions are determined, and then the parameters that combine the basis functions are computed. Fig. 5 shows a simple example of how an RBF network can isolate local areas of the input space with fewer mutable parameters than a sigmoid-node neural network. For an overview of RBF networks in the supervised learning literature, see Ghosh and Nag (2001).

The local processing in RBF networks has proved to be useful in many problems, frequently outperforming other function approximation techniques (Lawrence, Tsoi, & Back, 1996; Wedge, Ingram, McLean, Mingham, & Bandar, 2005). Such local approaches have been particularly useful on supervised learning problems that might be considered fractured, like the concentric spirals classification task (Chaiyaratana & Zalzala, 1998). This success suggests that an RBF approach could be useful for fractured reinforcement learning problems as well. Of course, supervised RBF algorithms take advantage of labeled training data when deciding how to build and tune RBF nodes, and such data is not available in reinforcement learning. Furthermore, most of the network architectures proposed in supervised RBF algorithms are fixed before learning or are constrained to be highly regular (e.g. a single hidden layer of RBF nodes). This constraint could limit the ability of the learning algorithms to find an appropriate representation for the problem at hand. Moreover, it may be possible to evolve the RBF networks and thereby construct complex networks for fractured problems.

Another interesting idea for generating locality in the supervised learning community is deep learning (Hinton & Salakhutdinov, 2006). The idea is that neural networks with a large number of nodes between input and output are able to form progressively high-level and abstract representations of input features and could generate fractured decision boundaries as well (Bengio, 2007; LeCun & Bengio, 2007). However, it is difficult to train deep networks with standard techniques like backpropagation because the error signal diminishes quickly over the many connections. In order to solve this problem, deep learning networks are pre-trained using unsupervised methods to cluster input patterns into distinct groups. This pre-training sets the weights of the network close to good values, which then allows backpropagation to run successfully.

The arguments for deep learning are complementary to those for constructive neuroevolution; both approaches result in complicated network structures that can hold sophisticated representations of input data, as opposed to single-layer architectures. The two approaches diverge in that the networks are not constructed in deep learning and that the second stage of deep learning relies on supervised feedback. However, it would be possible to incorporate deep learning's initialization of network weights into a neuroevolution algorithm, and thereby bias the search towards local solutions.

3.2. Reinforcement learning

In contrast to the approaches described above, reinforcement learning algorithms are designed to solve problems where labeled training data is unavailable. The idea of local processing has also proved to be effective for value-function reinforcement learning algorithms. Such methods frequently benefit from approximating value functions using highly local function approximators like tables, CMACs, or RBF networks (Kretchmar & Anderson, 1997; Li & Duckett, 2005; Li, Martinez-Maron, Lilienthal, & Duckett, 2006; Peterson & Sun, 1998; Stone, Kuhlmann, Taylor, & Liu, 2006; Sutton, 1996; Taylor, Whiteson, & Stone, 2006). For example, Sutton used a CMAC (a function approximator consisting of multiple overlapping receptive fields, known for its ability to generalize locally) successfully on a set of problems that had previously proved difficult to solve using global function approximators (Sutton, 1996). Asada et al. improved the learning performance of a value-function algorithm by grouping local patches of the state space together that shared the same action (Asada, Noda, & Hosoda, 1995). More recently, Stone et al. found that in the benchmark keepaway soccer problem, an RBF-based value-function approximator significantly outperformed a normal neural network value-function approximator (Stone et al., 2006). Such results suggest that local behavioral adjustments could be useful for policy-search reinforcement learning algorithms – like neuroevolution – as well.

3.3. Evolutionary computation

Evolutionary approaches to learning using the cascade correlation architecture have proven to be highly effective on certain

benchmark problems like concentric spirals (Potter & Jong, 2000; Tulai & Oppacher, 2002). Although the only concept that these approaches borrow from cascade correlation is the network architecture (i.e. the process of training hidden nodes to correlate with pre-existing error is not used), this topology restriction alone results in good performance on the concentric spirals problem. It is possible that this is a good approach to fractured problems in general.

Learning classifier systems (LCS) are another family of algorithms that use local processing to solve reinforcement learning problems. LCS approximate functions with a population of classifiers, each of which is responsible for a small part of the input space. A competitive contributory mechanism encourages classifiers to cover as much space as possible, removing redundant classifiers and increasing generalization. A number of LCS algorithms have been developed that vary both in how the classifiers cover the input space and in how they approximate local functions (Bull & O'Hara, 2002; Butz, 2005; Butz & Herbort, 2008; Howard, Bull, & Lanzi, 2008; Lanzi, Loiacono, Wilson, & Goldberg, 2005, 2006; Wilson, 2002, 2008). Of particular interest are approaches like those used in Neural XCSF, which use a fixed-topology or variable-size single-layer neural network to define both conditions and actions of a simple LCS. Although early work examining the role of constructive neural networks in LCS has been promising (Howard et al., 2008), the full potential of a combination of LCS and constructive neuroevolution has not yet been explored.

Third, several hybrid algorithms have been proposed that use various flavors of genetic algorithms to reduce the amount of required human expertise in supervised learning, usually by automatically determining the number, size, and location of basis functions (Angeline, 1997; Billings & Zheng, 1995; Chaiyaratana & Zalzala, 1998; Gonzalez et al., 2003; Guillen et al., 2007, 2006; Guo, Huang, & Zhao, 2003; Maillard & Gueriot, 1997; Sarimveis, Alexandridis, Mazarakis, & Bafas, 2004; Whitehead & Choate, 1996). These approaches still rely on supervised training data, at least in part, and typically are also constrained to produce single-layer network architectures.

For instance, the Global–Local ANN (GL-ANN) architecture proposed by Wedge et al. first trains a single-layer sigmoid-node network, then constructively adds RBF nodes, and finally adjusts all parameters of the network (Wedge, Ingram, McLean, Mingham, & Bandar, 2006). Similarly, the Structural Modular Neural Networks approach uses a genetic algorithm to evolve single-layer networks with both sigmoid and RBF nodes (Jiang, Zhao, & Ren, 2003). These approaches are intriguing in that they combine global approximation with sigmoid nodes with the local adjustments of RBF nodes. The resulting network architectures are still quite regular when compared to the unbiased architectures that constructive neuroevolution algorithms can discover.

A fourth area of of related work is genetic programming (GP), where Rosca developed methods to allow GP to decompose problems into useful hierarchies and abstractions (Rosca, 1997). To the extent that the fracture for a given problem is organized in a hierarchical manner, the adaptive representations could be used to bias search towards small, repeated motifs. Of course, the notion of reusable modules of code is easier to define for genetic programs than it is for neural networks. In order to take advantage of this work in GP, it is necessary to understand how modular neural networks can be developed, but a cascaded structure is a possible start.

The concept of locality appears in many fields other than machine learning. One particularly interesting area is the study of human cognition and cognitive modeling. Although not the primary focus of this work, it is fascinating and potentially useful to consider the role of locality in human cognition. In particular, several papers in this special issue show that the ability to make local changes to cognitive processes is biologically plausible, whether viewed as attractors in the prefrontal cortex (Levine, 2009), division of general knowledge into discrete chunks (Kozma & Freeman, 2009), or the representation of language in the brain using different states (Perlovsky, 2009).

The next section describes how the ideas above might be incorporated into current neuroevolution algorithms. The approach is to create modified versions of a state-of-the-art neuroevolution algorithm, as will be described next.

4. Utilizing locality in neuroevolution

One of the most promising neuroevolution algorithms to date is the Neuroevolution of Augmenting Topologies (NEAT) algorithm (Stanley & Miikkulainen, 2002, 2004a). This section reviews the NEAT algorithm and describes two modifications to NEAT designed to improve its performance in fractured problems by biasing or constraining the search for network topologies. In the following section, these modifications will be compared empirically to NEAT and a linear baseline algorithm on several fractured problems.

4.1. The NEAT neuroevolution method

NEAT is designed to solve difficult reinforcement learning problems by automatically evolving network topology to fit the complexity of the problem. NEAT combines the usual search for the appropriate network weights with complexification of the network structure. It starts with simple networks and expands the search space only when beneficial, allowing it to find significantly more complex controllers than fixed-topology evolution. These properties make NEAT an attractive method for evolving neural networks in complex tasks.

NEAT is based on three key ideas. First, evolving network structure requires a flexible genetic encoding. 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 NE system. Structural mutations, which allow complexity to increase, either add a new connection or a new node to the network. Through mutation, genomes of varying sizes are created, sometimes with completely different connections specified at the same positions.

Each unique gene in the population is assigned a unique innovation number, and the numbers are inherited during crossover. Innovation numbers allow NEAT to do crossover without the need for expensive topological analysis. Genomes of different organizations and sizes stay compatible throughout evolution, and the problem of matching different topologies (Radcliffe, 1993) is essentially avoided.

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 (Goldberg & Richardson, 1987), where organisms in the same species must share the fitness of their niche, preventing any one species from taking over the population.

Third, unlike other systems that evolve network topologies and weights (Gruau et al., 1996; Yao, 1999), NEAT begins with a uniform population of simple networks with no hidden nodes. New structure is introduced incrementally as structural mutations

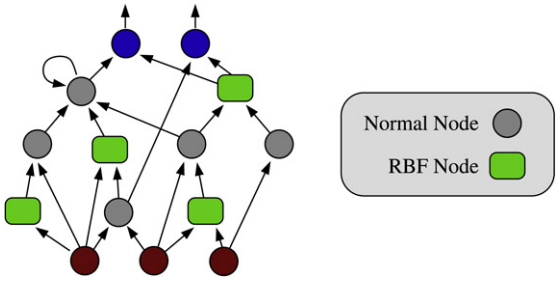

Fig. 6. An example of the network topology evolved by the RBF-NEAT algorithm. Radial basis function nodes, initially connected to inputs and outputs, are provided as an additional mutation to the algorithm. These nodes allow evolution to utilize local structures where they may be appropriate, e.g. in fractured problems.

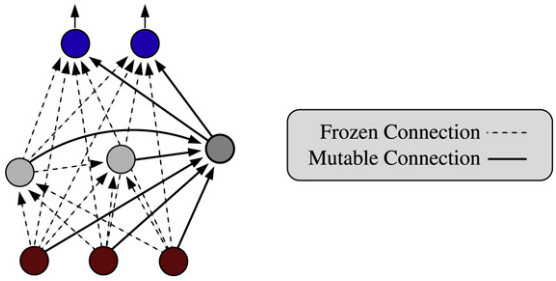

Fig. 7. An example of a network constructed by Cascade-NEAT. Only connections associated with the most recently added hidden node are evolved. Compared to NEAT and RBF-NEAT, Cascade-NEAT constructs networks with a regular topology, where successively more refined decision boundaries are produced at each cascaded level.

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.

These three ideas allow NEAT to find surprisingly small solutions to a variety of reinforcement learning problems. However, NEAT's ability to solve fractured problems is limited. This section continues with descriptions of two modified versions of NEAT – inspired by the related literature – that are designed to perform better on fractured problems. Both of these approaches are essentially extensions to the standard NEAT algorithm that are designed to improve performance on fractured problems by either biasing or constraining the types of network structure that NEAT explores towards more local representations.

4.2. The RBF-NEAT algorithm

The first algorithm, called RBF-NEAT, extends NEAT by introducing a new topological mutation that adds a radial basis function node to the network. Like NEAT, the algorithm 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 usual "add link" and "add node" mutations in NEAT, with probability $\epsilon = 0.05$ an "add RBF node" mutation occurs (Fig. 6). Each RBF node is activated by an axis-parallel Gaussian with variable center and size. All free parameters of the network, including RBF node parameters and link weights, are determined by a genetic algorithm similar to the one in NEAT (Stanley & Miikkulainen, 2002).

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.

4.3. The Cascade-NEAT algorithm

The search for network topologies can also be biased towards fractured solutions by constraining the search to cascaded structures. The cascade architecture (shown in Fig. 7) is a regular form of network architecture where each hidden node is connected to inputs, outputs, and all hidden nodes to its left.

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 of "add node" and "add connection", Cascade-NEAT uses an "add cascade node" mutation: With probability $\epsilon = 0.05$, a hidden node is added to the network. This hidden node has inputs from all inputs and existing hidden nodes in the network, and is connected to all outputs. 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.

Cascade-NEAT adds a considerable constraint to the search for appropriate network topologies, given the wide variety of network structures that the normal NEAT algorithm examines. The next section examines the effect of this constraint – and the effect of the bias in RBF-NEAT – on a series of fractured problems.

5. Empirical analysis

In order to test the hypothesis that biasing and constraining topology search to local solutions is beneficial in fractured problems, RBF-NEAT and Cascade-NEAT were compared with the standard NEAT algorithm on several different benchmark problems. Also included was a baseline algorithm consisting of NEAT without any structural mutation operators, i.e. a method that evolves a single layer of weights with no hidden nodes. This linear combination of input features is the same initial network topology that NEAT starts with, and is included to provide a sense of scale to the following graphs.

5.1. Generating maximal variation

The first experiment was designed to evaluate how much variation these different learning algorithms can produce in an unrestricted setting. A problem was created where the only goal was to produce a "solution" that contained as much variation as possible.

Three versions of this problem were created, each with a different number (one, two or three) of inputs. The input space for each problem was uniformly divided into roughly 200 points. An evaluation consisted of evaluating a network on a each of these points and noting the value that was produced from the single output. The score for a network was the total variation of the discretized function that the network represented, calculated in a manner described in Section 2.3.

The results from this experiment are shown in Fig. 8. Cascade-NEAT is able to produce significantly higher variation than other algorithms. Interestingly RBF-NEAT produces relatively high variation for a single input but less as the number of inputs increases, suggesting that RBF-NEAT is mainly effective in low-dimensional settings.

5.2. Function approximation

The general function approximation problem requires the learning algorithm to evolve neural networks to approximate fixed 1-d functions. Each network is evaluated on a series of numbers representing the input to the function. The network state is cleared

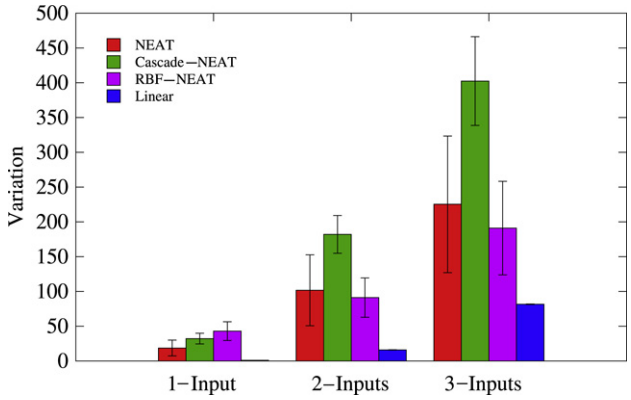

Fig. 8. Performance of four learning algorithms on a problem where the goal is to produce a solution with as much variation as possible. When the dimensionality is small, RBF-NEAT does well, but in general, Cascade-NEAT is able to produce the highest amount of variation.

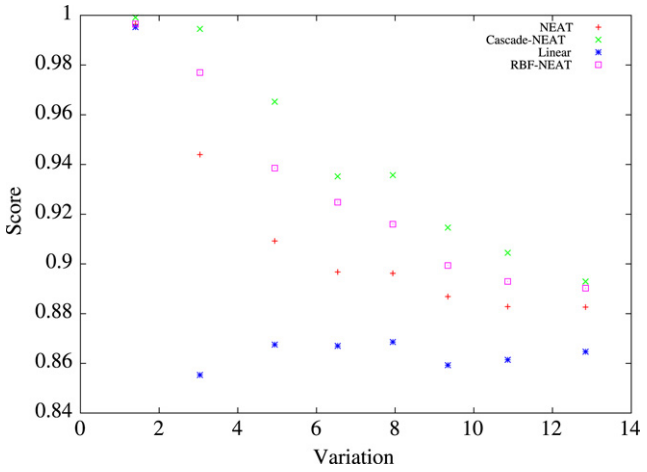

Fig. 10. Results for the sine wave function approximation problem. Performance drops as the amount of variation required to solve the problem increases, but RBF-NEAT and Cascade-NEAT outperform the standard NEAT algorithm significantly ($p > 0.95$).

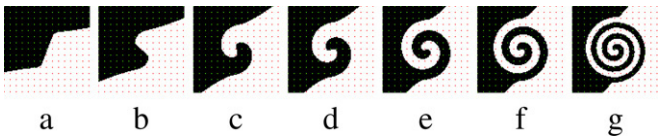

Fig. 11. Seven versions of the concentric spirals problem that vary in the degree to which the two spirals are intertwined. The colored dots indicate the discretization used to generate data from each spiral. As the spirals become increasingly intertwined, the variation of the optimal policy increases. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

before each new input is presented, then the input is fed into the network for $\kappa = 10$ activations. The squared error between the output of the network and the target function is recorded for a series of $\tau = 100$ input points. After the network has been evaluated on all τ input points for a function, the mean squared error is inverted and used as a fitness signal. Function approximation is a good test problem because it is easy to visualize, and because it is straightforward to calculate the variation of the optimal solution.

The functions to be approximated follow the form $\sin(\alpha x)$. The six different versions of this sine function (shown in Fig. 9) have increasing variation, corresponding to larger values of α.

Fig. 10 shows the performance of NEAT, Cascade-NEAT, RBF-NEAT, and the linear baseline algorithm (each averaged over 100 runs) on each of these six function approximation problems. The horizontal position of each pair of points indicates the variation of the optimal solution for that problem.

As variation increases, the score for NEAT drops, confirming the hypothesis that variation measures how difficult the problem is for NEAT. Although all algorithms perform similarly in the problem with the least amount of variation, a marked difference appears as variation increases. The Cascade-NEAT and RBF-NEAT algorithms generate scores that are nearly twice as good as the normal NEAT algorithm (using the linear version of NEAT as a baseline), supporting the hypothesis that incorporating a locality bias into network construction makes learning high-variation problems easier.

5.3. Concentric spirals

Concentric spirals is a classic supervised learning benchmark task often used to evaluate the Cascade Correlation architecture. Originally proposed by Wieland (Potter & Jong, 2000), the problem consists of correctly identifying points from two intertwined spirals. Solving this problem involves repeatedly tagging nearby regions of the input space with different labels, which makes the decision task fractured.

In order to examine the effect of fracture on NEAT, seven semi-supervised versions of increasing difficulty of the concentric spirals problem of were created (Fig. 11). As the spirals become increasingly intertwined, the variation of the optimal policy increases. Note that this version of the problem differs from the supervised version, where a learner receives feedback about individual points. This modified version of concentric spirals – like all the domains examined in this paper – is cast as a reinforcement learning problem, which means the learning agent receives dramatically less information about its performance. In this problem, the only feedback an agent receives is the number of points properly classified. This makes the task of correctly identifying points on the two spirals much more difficult.

Fig. 12 shows the score for the four learning algorithms (NEAT, Cascade-NEAT, RBF-NEAT, and the linear baseline algorithm) averaged over 25 runs. Again, scores decrease as variation increases, showing that the variation of each problem correlates closely with problem difficulty. However, Cascade-NEAT and RBF-NEAT are able to offer significant increases in performance over that of the standard NEAT algorithm.

Fig. 13 shows the output of the best evolved solutions from each learning algorithm for two of these problems. NEAT is able to find an approximate solution for the simpler problem, but is

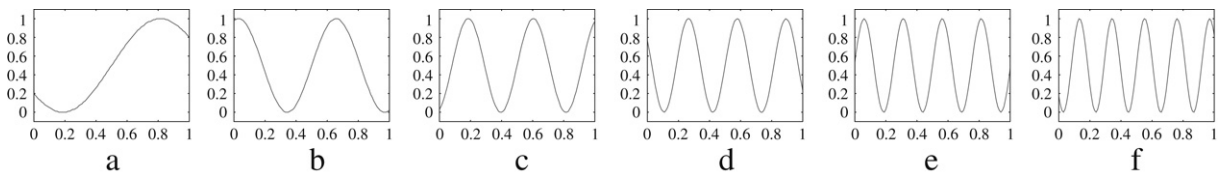

Fig. 9. Six versions of a sine wave function approximation problem. Sine waves with higher frequency have higher variation, making them harder to approximate.

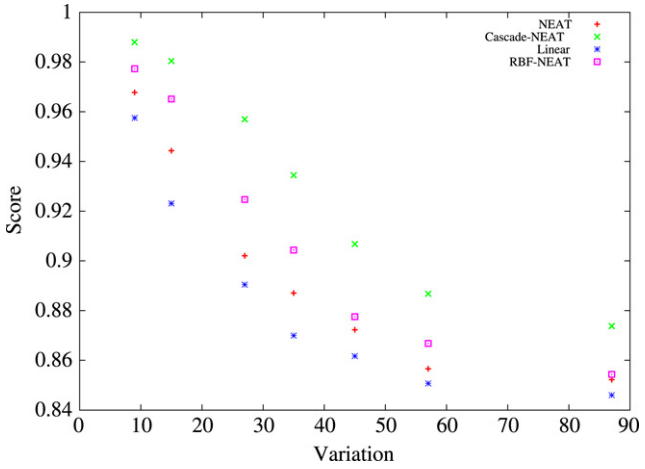

Fig. 12. Average score for the four learning algorithms on seven versions of the concentric spirals problem. As the variation of the problem increases, performance falls, but Cascade-NEAT and RBF-NEAT are able to significantly outperform the standard NEAT algorithm ($p > 0.95$).

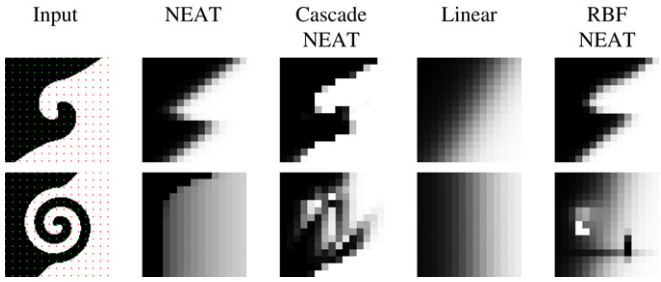

Fig. 13. Output of the best solutions found by each learning algorithm for two versions of the challenging semi-supervised concentric spirals problem. Cascade-NEAT and RBF-NEAT do a much better job than NEAT at generating the subtle variations required by the more complicated version of the problem.

unable to discover a network that can represent the variation required to do well on the more complex problem. The solutions that Cascade-NEAT and RBF-NEAT generate, while not perfect, are able to encompass more variation than those discovered by NEAT.

5.4. Multiplexer

The multiplexer is a challenging benchmark problem from the evolutionary computation community. Performing well on this problem requires an agent to 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 set as output. The binary representation and the division of the input into two separate logical groups suggests intuitively that the multiplexer problem is fractured.

Four experiments were performed with increasingly difficult versions of the problem, which are shown in Fig. 14. These four problems differ in the size of the input, ranging from three (one address bit and two data bits) to nine (three address bits and six data bits). Note that not all values for the third address bit are used for the two largest versions of the problem. As the number of inputs increases, the variation of the optimal solution also increases. This increase allows the impact of variation on performance to be measured.

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 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 the inverted mean squared error over all inputs.

Fig. 15 shows the performance of NEAT, Cascade-NEAT, RBF-NEAT, and the linear baseline algorithm on these multiplexer problems. As in previous sections, each group of four vertical points represents one of the problems. While NEAT is able to perform well on the simplest multiplexer, its performance falls off quickly as the required variation increases. Interestingly, RBF-NEAT does not offer significant increases in performance over regular NEAT for any other versions. However, Cascade-NEAT is able to outperform all other algorithms significantly.

5.5. Keepaway soccer

The final empirical comparison expands the benchmark comparisons above to a high-level decision-making task. The four learning algorithms described above were evaluated on a version of the 4-versus-2 keepaway soccer problem (Stone et al., 2006; Whiteson, Kohl, Miikkulainen, & Stone, 2005). Keepaway soccer is a challenging high-level decision task with continuous input. The goal is for the four keepers to prevent the two takers from controlling the ball in a bounded area. One feature that makes this particular version of keepaway difficult is that the takers can move five times faster than the keepers, which forces the keepers to develop a robust passing strategy instead of merely running with the ball. Fig. 16 shows a typical initial state of a keepaway game.

The takers behave according to a fixed, hand-coded algorithm that focuses on covering passing lanes and converging on the ball. The four keepers are controlled by a mix of hand-coded and evolved behaviors. When a game starts, the keeper nearest the ball is made "responsible" for the ball. If this responsible keeper is not close enough to the ball, it executes a pre-existing intercept behavior in an effort to get control of the ball. The keepers not responsible for the ball execute a pre-existing get-open behavior, designed to put the keepers in a good position to receive a pass.

However, when the responsible keeper has control of the ball (defined by being within ϕ meters of the ball) it must choose between executing a pre-existing hold behavior or attempting a pass to one of its three teammates. The goal of learning is to make the appropriate decision given the state of the game at this point.

To make this decision, the network controlling the responsible keeper receives ten continuous inputs. The first input describes the keeper's distance from the center of the field. The network also receives three inputs for each teammate: the distance to that teammate, the angle between that teammate and the nearest taker, and the distance to that nearest taker. All angles and distances are normalized to the range [0, 1]. The network has one output for each possible action (hold, or pass to one of the three teammates). The output with the highest activation is interpreted as the keeper's action.

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

Each network was evaluated from $\tau = 30$ different randomly-chosen initial configurations of takers and keepers. In each configuration, the ball is initially placed near one of the keepers. Each of the players executes the appropriate hand-coded behavior, and the current network is used to select an action when the keeper responsible for the ball needs to choose between holding and passing. The game is allowed to proceed until a timeout is reached, the ball goes out of bounds, or a taker achieves control of the ball (by getting within ϕ meters of it). The score for a single

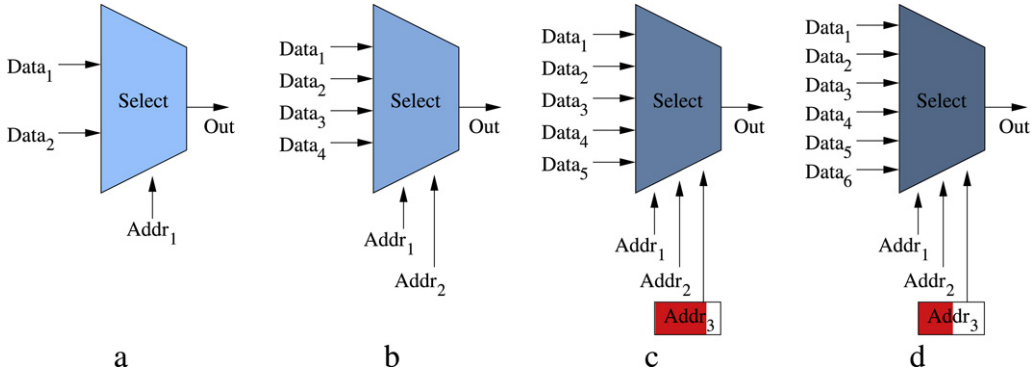

Fig. 14. Four versions of the multiplexer problem, where the goal is to use address bits to select a particular data bit. For (c) and (d), not all of the values for the third address bit were used. The amount of variation required to solve the multiplexer problem increases as the number of total inputs (address bits plus data bits) increases, making the problem harder.

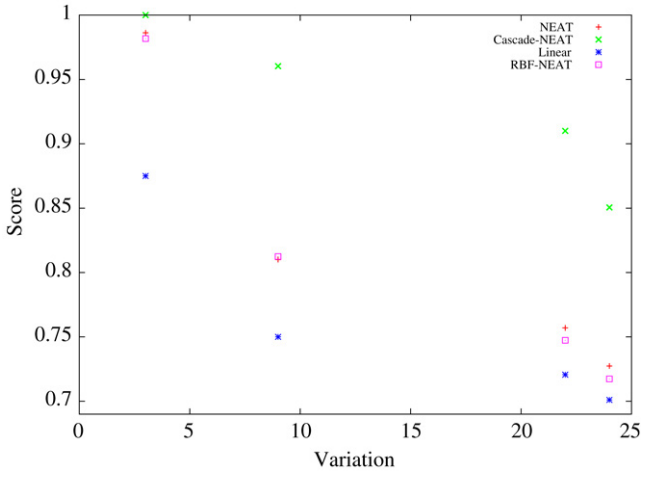

Fig. 15. Performance of the four learning algorithms on four versions of the multiplexer problem. Cascade-NEAT is able to dramatically improve performance over the other algorithms ($p > 0.95$).

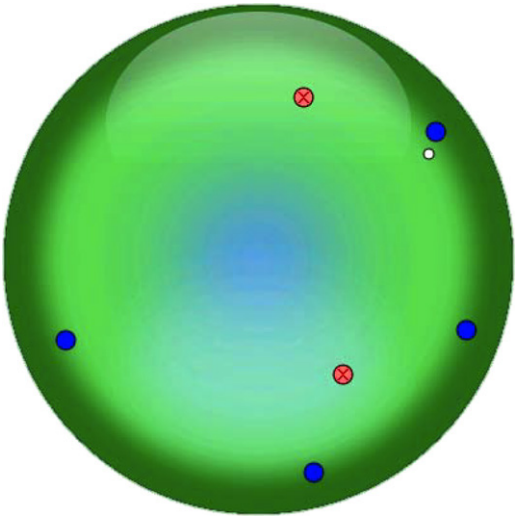

Fig. 16. A starting configuration of players for the 4-versus-2 keepaway soccer problem. The four keepers (the darker players) attempt to keep the ball (shown in white) away from the two takers (the lighter players with crosses). Keepaway is a challenging high-level strategy problem with continuous inputs and a fractured decision space.

game is the number of timesteps that the game takes. The overall score for the network is the sum of the scores for all τ games.

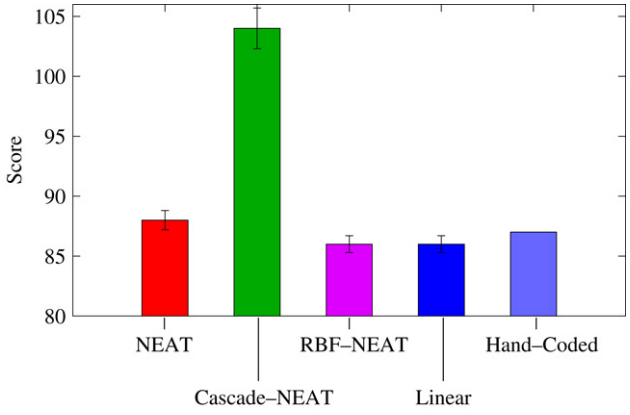

Fig. 17. Comparison of the four learning algorithms and a hand-coded solution in the keepaway soccer problem. While NEAT is able to slightly improve on the hand-coded behavior, Cascade-NEAT offers the best performance by a wide margin. Animations of the best learned policies can be seen at nn.cs.utexas.edu/?fracture.

Fig. 17 shows a comparison of the four learning algorithms (NEAT, Cascade-NEAT, RBF-NEAT, and the linear baseline algorithm) as well as a hand-coded solution for the keepaway soccer problem. NEAT was able to offer moderate improvement over the hand-coded policy, but Cascade-NEAT offers the highest performance by a wide margin. Animations of the best learned and hand-coded policies can be seen at nn.cs.utexas.edu/?fracture.

One method of varying the amount of fracture in the keepaway domain is to change τ, the number of initial states on which each network is evaluated. Reducing the number of starting states should reduce variation, making the problem easier to solve. Intuitively, this has the effect of reducing the amount of area over which the network must generalize, resulting in a simpler function that the network must approximate. As the number of required states decreases, it should become easier to solve the problem with a relatively simple mapping from states to actions. Fig. 18 shows the effect of reducing the number of starting states on the four learning algorithms.

In general, versions of the keepaway problem with fewer starting states are easier to solve. However, as the number of starting states increases, the superior performance of Cascade-NEAT becomes more pronounced. This result supports the hypothesis that problems become increasingly fractured as the scope of learning increases, and that Cascade-NEAT is much more adept at solving these fractured problems.

6. Discussion and future work

The experiments in this paper confirm the hypothesis that NEAT has difficulty in solving fractured domains. When the amount

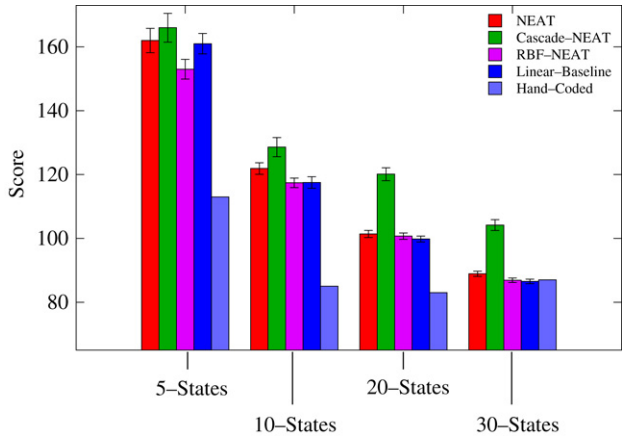

Fig. 18. Measuring the effect of the number of starting states on learning performance. As the number of starting states increases, the relative performance gain provided by Cascade-NEAT increases. This result suggests that the utility of Cascade-NEAT increases with problem fracture, making it a good candidate for learning in high-level decision-making tasks.

of variation required to solve a problem is small, NEAT does well. But as the required variation increases, NEAT's performance falls off quickly. However, biasing and constraining network construction towards local structure is found to dramatically improve performance on highly-fractured problems. Both RBF-NEAT and Cascade-NEAT offer improved performance on all problems.

Interestingly, 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. 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.

A related avenue for future work involves the possible combination of RBF-NEAT and Cascade-NEAT. These two algorithms show promise in different scenarios, and a combination of the two could result in a better overall algorithm.

In addition to the cascade architecture and basis functions, there are other useful ideas from the machine learning community that could be applied to neuroevolution. Chief among these possibilities is the potential for an initial unsupervised training period to initialize a large network, similar to the initial step of training that happens in deep learning. Using unsupervised learning to provide a good starting point for the search process could have a dramatic effect on learning performance.

Finally, it would be useful to evaluate the lessons learned here on other high-level reinforcement learning problems. One potential candidate is a multi-agent vehicle control task, such as that examined in Stanley, Kohl, Sherony, and Miikkulainen (2005a). Previous work has shown that algorithms like NEAT are effective at generating low-level control behaviors, like efficiently steering a car through S-curves on a track. Successfully evolving higher-level behavior to reason about opponents or race strategy has proven difficult, but may be possible with algorithms like Cascade-NEAT and RBF-NEAT.

7. Conclusion

Despite its success in the past, neuroevolution in general, and NEAT in particular, has surprising difficulty solving certain types of high-level decision-making problems. This paper presents the hypothesis that this difficulty arises because these problems are fractured: The correct action varies discontinuously as the agent moves from state to state. A method for measuring fracture using the concept of function variation is proposed, and several examples of high-level reinforcement learning problems that possess such a fractured quality are presented. While NEAT is shown to perform rather poorly on these fractured problems, two modifications to NEAT, called RBF-NEAT and Cascade-NEAT, improve performance significantly by biasing or constraining the search for network topologies towards local solutions. Thus, these methods lay the groundwork for the next generation of neuroevolution algorithms that can discover high-level strategic behavior.

References

Angeline, P. J. (1997). Evolving basis functions with dynamic receptive fields. In *IEEE international conference on systems, man, and cybernetics*: *Vol. 5* (pp. 4109–4114).
Angeline, P. J., Saunders, G. M., & Pollack, J. B. (1993). An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, *5*, 54–65.
Asada, M., Noda, S., & Hosoda, K. (1995). Non-physical intervention in robot learning based on LFE method. In *Proceedings of machine learning conference workshop on learning from examples vs. programming by demonstration* (pp. 25–31).
Barron, A., Rissanen, J., & Yu, B. (1998). The minimum description length principle in coding and modeling. *IEEE Transactions on Information Theory*, *44*(6), 2743–2760.
Bengio, Y. (2007). Learning deep architectures for ai. *Tech. rep. 1312*. Dept. IRO, Universite de Montreal.
Billings, S. A., & Zheng, G. L. (1995). Radial basis function network configuration using genetic algorithms. *Neural Networks*, *8*, 877–890.
Bochner, S. (1959). Lectures on Fourier integrals with an author's supplement on monotonic functions. In *Stieltjes integrals and harmonic analysis*. Princeton University Press.
Bull, L., & O'Hara, T. (2002). Accuracy-based neuro and neuro-fuzzy classifier systems. In *Proceedings of the genetic and evolutionary computation conference* (pp. 905–911).
Butz, M. V. (2005). Kernel-based, ellipsoidal conditions in the real-valued xcs classifier system. In *Proceedings of the 2005 conference on genetic and evolutionary computation* (pp. 1835–1842).
Butz, M. V., & Herbort, O. (2008). Context-dependent predictions and cognitive arm control with xcsf. In *Proceedings of the 2008 conference on genetic and evolutionary computation*.
Chaitin, G. (1975). A theory of program size formally identical to information theory. *Journal of the ACM*, *22*, 329–340.
Chaiyaratana, N., & Zalzala, A. M. S. (1998). Evolving hybrid rbf-mlp networks using combinedgenetic/unsupervised/supervised learning. In *UKACC international conference on control*: *Vol. 1* (pp. 330–335).
Ghosh, J., & Nag, A. (2001). An overview of radial basis function networks. In *Studies in fuzziness and soft computing: Radial basis function networks 2: New advances in design* (pp. 1–36).
Goldberg, D. E., & Richardson, J. (1987). Genetic algorithms with sharing for multimodal function optimization. In *Proceedings of the second international conference on genetic algorithms* (pp. 148–154).
Gomez, F., & Miikkulainen, R. (1999). Solving non-Markovian control tasks with neuroevolution. In *Proceedings of the 16th international joint conference on artificial intelligence*.
Gomez, F., Schmidhuber, J., & Miikkulainen, R. (2006). Efficient non-linear control through neuroevolution. In *Proceedings of the European conference on machine learning*.
Gonzalez, J., Rojas, I., Ortega, J., Pomares, H., Fernandez, F., & Diaz, A. (2003). Multiobjective evolutionary optimization of the size, shape, and position parameters of radial basis function networks for function approximation. *IEEE Transactions on Neural Networks*, *14*, 1478–1495.
Gruau, F., Whitley, D., & Pyeatt, L. (1996). A comparison between cellular encoding and direct encoding for genetic neural networks. In J. R. Koza, D. E. Goldberg, D. B. Fogel, & R. L. Riolo (Eds.), *Genetic programming 1996: Proceedings of the first annual conference* (pp. 81–89). MIT Press.
Guillen, A., Pomares, H., Gonzalez, J., Rojas, I., Herrera, L. J., & Prieto, A. (2007). Parallel multi-objective memetic RBFNNS design and feature selection for function approximation problems 4507/2007. pp. 341–350.
Guillen, A., Rojas, I., Gonzalez, J., Pomares, H., Herrera, L. J., & Paechter, B. (2006). Improving the performance of multi-objective genetic algorithm for function approximation through parallel islands specialisation 4304/2006. pp. 1127–1132.
Guo, L., Huang, D.-S., & Zhao, W. (2003). Combining genetic optimisation with hybrid learning algorithm for radial basis function neural networks. *Electronics Letters*, *39*, 1600–1601.
Gutmann, H. (2001). A radial basis function method for global optimization. *Journal of Global Optimization*, *19*, 201–227.
Hinton, G. E., & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science*, *313*(5786), 504–507.

Ho, T., & Basu, M. (2002). Complexity measures of supervised classification problems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *24*(3), 289–300.
Hornik, K. M., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 359–366.
Howard, D., Bull, L., & Lanzi, P.-L. (2008). Self-adaptive constructivism in neural XCS and XCSF. In *Proceedings of the 2008 genetic and evolutionary computation conference*.
Jiang, N., Zhao, Z., & Ren, L. (2003). Design of structural modular neural networks with genetic algorithms. *Advances in Software Engineering*, *34*, 17–24.
Kamke, E. (1956). Das lebesgue-stieltjes integral.
Kohl, N., Stanley, K., Miikkulainen, R., Samples, M., & Sherony, R. (2006). Evolving a real-world vehicle warning system. In *Proceedings of the genetic and evolutionary computation conference 2006* (pp. 1681–1688).
Kolmogorov, A. (1965). Three approaches to the quantitative definition of information. *Problems of Information Transmission*, *1*, 4–7.
Kozma, R., & Freeman, W. (2009). The kiv model of intentional dynamics and decision making. *Neural Networks*,.
Kretchmar, R., & Anderson, C. (1997). Comparison of CMACS and radial basis functions for local function approximators in reinforcement learning. In *Proceedings of the International Conference on neural networks*.
Lanzi, P. L., Loiacono, D., Wilson, S. W., & Goldberg, D. E. (2005). XCS with computed prediction for the learning of boolean functions. In *Proceedings of the IEEE congress on evolutionary computation conference*.
Lanzi, P. L., Loiacono, D., Wilson, S. W., & Goldberg, D. E. (2006). Classifier prediction based on tile coding. In *Proceedings of the genetic and evolutionary computation conference* (pp. 1497–1504).
Lawrence, S., Tsoi, A., & Back, A. (1996). Function approximation with neural networks and local methods: Bias, variance and smoothness. In *Australian conference on neural networks* (pp. 16–21).
LeCun, Y., & Bengio, Y. (2007). Scaling learning algorithms towards ai. In *Large-scale kernel machines*.
Leonov, A. S. (1998). On the total variation for functions of several variables and a multidimensional analog of Helly's selection principle. *Mathematical Notes*, *63*(1), 61–71.
Levine, D. (2009). Brain pathways for cognitive-emotional decision making in the human animal. *Neural Networks*,.
Li, J., & Duckett, T. (2005). 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*.
Li, J., Martinez-Maron, T., Lilienthal, A., & Duckett, T. (2006). Q-ran: A constructive reinforcement learning approach for robot behavior learning. In *Proceedings of IEEE/RSJ international conference on intelligent robot and system*.
Li, M., & Vitanyi, P. (1993). *An introduction to Kolmogorov complexity and its applications*. Springer-Verlag.
Maciejowskia, J. M. (1979). Model discrimination using an algorithmic information criterion. *Automatica*, *15*, 579–593.
Maillard, E., & Gueriot, D. (1997). RBF neural network, basis functions and genetic algorithm. In *International Conference on neural networks*: *Vol. 4*.
Mitchell, T. (1997). *Machine learning*. McGraw Hill.
Moody, J., & Darken, C. J. (1989). Fast learning in networks of locally tuned processing units. *Neural Computation*, *1*, 281–294.
Moriarty, D. E., & Miikkulainen, R. (1996). Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, *22*, 11–32.
Park, J., & Sandberg, I. W. (1991). Universal approximation using radial-basis-function networks. *Neural Computation*, *3*, 246–257.
Perlovsky, L. (2009). Language and cognition. *Neural Networks*,.
Peterson, T., & Sun, R. (1998). An rbf network alternative for a hybrid architecture. In *IEEE International Joint Conference on neural networks*: *Vol. 1* (pp. 768–773).
Platt, J. (1991). A resource-allocating network for function interpolation. *Neural Computation*, *3*(2), 213–225.
Potter, M. A., & Jong, K. A. D. (2000). Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation*, *8*(1), 1–29.
Radcliffe, N. J. (1993). Genetic set recombination and its application to neural network topology optimization. *Neural Computing and Applications*, *1*(1), 67–90.
Reisinger, J., Bahceci, E., Karpov, I., & Miikkulainen, R. (2007). Coevolving strategies for general game playing. In *Proceedings of the IEEE symposium on computational intelligence and games*.
Rosca, J. P. (1997). Hierarchical learning with procedural abstraction mechanisms. *Ph.D. thesis*. Rochester, NY 14627, USA. citeseer.ist.psu.edu/rosca97hierarchical.html.
Saravanan, N., & Fogel, D. B. (1995). Evolving neural control systems. *IEEE Expert*, 23–27.
Sarimveis, H., Alexandridis, A., Mazarakis, S., & Bafas, G. (2004). A new algorithm for developing dynamic radial basis function neural network models based on genetic algorithms. *Computers and Chemical Engineering*, *28*, 209–217.
Shilov, G. E., & Gurevich, B. L. (1967). Integral, measure, derivative.
Stanley, K., Kohl, N., Sherony, R., & Miikkulainen, R. (2005). Neuroevolution of an automobile crash warning system. In *Proceedings of the genetic and evolutionary computation conference 2005* (pp. 1977–1984).
Stanley, K. O., Bryant, B. D., & Miikkulainen, R. (2005). Real-time neuroevolution in the NERO video game. *IEEE Transactions on Evolutionary Computation*, *9*(6), 653–668.
Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, *10*(2).
Stanley, K. O., & Miikkulainen, R. (2004a). Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence Research*, *21*, 63–100.
Stanley, K.O., & Miikkulainen, R. (2004b). Evolving a roving eye for go. In *Proceedings of the genetic and evolutionary computation conference*.
Stone, P., Kuhlmann, G., Taylor, M. E., & Liu, Y. (2006). Keepaway soccer: From machine learning testbed to benchmark. In I. Noda, A. Jacoff, A. Bredenfeld, & Y. Takahashi (Eds.), *RoboCup-2005: Robot soccer world cup IX*: *Vol. 4020* (pp. 93–105). Berlin: Springer Verlag.
Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in neural information processing systems 8* (pp. 1038–1044).
Taylor, M., Whiteson, S., & Stone, P. (2006). Comparing evolutionary and temporal difference methods for reinforcement learning. In *Proceedings of the genetic and evolutionary computation conference* (pp. 1321–28).
Tulai, A. F., & Oppacher, F. (2002). Combining competitive and cooperative coevolution for training cascade neural networks. In *Proceedings of the genetic and evolutionary computation conference* (pp. 618–625).
Vapnik, V., & Chervonenkis, A. (1971). On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, *16*, 264–280.
Vitushkin, A. G. (1955). On multidimensional variations.
Wedge, D., Ingram, D., McLean, D., Mingham, C., & Bandar, Z. (2005). Neural network architectures and wave overtopping. In *Proc. Inst. Civil Engineering 2005: Maritime engineering*: *Vol. 158* (pp. 123–133). MA3.
Wedge, D., Ingram, D., McLean, D., Mingham, C., & Bandar, Z. (2006). On global–local artificial neural networks for function approximation. *IEEE Transactions on Neural Networks*, *17*(4), 942–952.
Whitehead, B., & Choate, T. (1996). Cooperative-competitive genetic evolution of radial basis functioncenters and widths for time series prediction. *IEEE Transactions on Neural Networks*, *7*, 869–880.
Whiteson, S., Kohl, N., Miikkulainen, R., & Stone, P. (2005). Evolving keepaway soccer players through task decomposition. *Machine Learning*, *59*, 5–30.
Whitley, D., Dominic, S., Das, R., & Anderson, C. W. (1993). Genetic reinforcement learning for neurocontrol problems. *Machine Learning*, *13*, 259–284.
Wieland, A. (1991). Evolving neural network controllers for unstable systems. In *Proceedings of the International Joint Conference on neural networks* (pp. 667–673).
Wilson, S. W. (2002). Classifiers that approximate functions. *Natural Computing*, *1*, 211–234.
Wilson, S. W. (2008). Classifier conditions using gene expression programming. *Tech. rep. 2008001*. Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign.
Yao, X. (1999). Evolving artificial neural networks. *Proceedings of the IEEE*, *87*(9), 1423–1447.