Chapter 10
Evolutionary Computation for Reinforcement Learning

Shimon Whiteson

Abstract. Algorithms for evolutionary computation, which simulate the process of natural selection to solve optimization problems, are an effective tool for discovering high-performing reinforcement-learning policies. Because they can automatically find good representations, handle continuous action spaces, and cope with partial observability, evolutionary reinforcement-learning approaches have a strong empirical track record, sometimes significantly outperforming temporal-difference methods. This chapter surveys research on the application of evolutionary computation to reinforcement learning, overviewing methods for evolving neural-network topologies and weights, hybrid methods that also use temporal-difference methods, coevolutionary methods for multi-agent settings, generative and developmental systems, and methods for on-line evolutionary reinforcement learning.

10.1 Introduction

Algorithms for *evolutionary computation*, sometimes known as *genetic algorithms* (Holland, 1975; Goldberg, 1989), are optimization methods that simulate the process of natural selection to find highly fit solutions to a given problem. Typically the problem assumes as input a *fitness function* $f : C \to \mathbb{R}$ that maps C, the set of all candidate solutions, to a real-valued measure of fitness. The goal of an optimization method is to find $c^* = \arg\max_c f(c)$, the fittest solution. In some cases, the fitness function may be stochastic, in which case $f(c)$ can be thought of as a random variable and $c^* = \arg\max_c E[f(c)]$.

Evolutionary methods search for c^* by repeatedly selecting and reproducing a population of candidate solutions. The initial population is typically chosen randomly, after which each member of the population is evaluated using f and the

Shimon Whiteson
Informatics Institute, University of Amsterdam
e-mail: `s.a.whiteson@uva.nl`

best performing ones are selected as the basis for a new population. This new population is formed via reproduction, in which the selected policies are mated (i.e., components of two different solutions are combined) and mutated (i.e., the parameter values of one solution are stochastically altered). This process repeats over many iterations, until a sufficiently fit solution has been found or the available computational resources have been exhausted.

There is an enormous number of variations on this approach, such as multi-objective methods (Deb, 2001; Coello et al, 2007) diversifying algorithms (Holland, 1975; Goldberg and Richardson, 1987; Mahfoud, 1995; Potter and De Jong, 1995; Darwen and Yao, 1996) and distribution-based methods (Larranaga and Lozano, 2002; Hansen et al, 2003; Rubinstein and Kroese, 2004). However, the basic approach is extremely general and can in principle be applied to all optimization problems for which f can be specified.

Included among these optimization problems are reinforcement-learning tasks (Moriarty et al, 1999). In this case, C corresponds to the set of possible policies, e.g., mappings from S to A, and $f(c)$ is the average cumulative reward obtained while using such a policy in a series of Monte Carlo trials in the task. In other words, in an evolutionary approach to reinforcement learning, the algorithm directly searches the space of policies for one that maximizes the expected cumulative reward.

Like many other policy-search methods, this approach reasons only about the value of entire policies, without constructing value estimates for particular state-action pairs, as temporal-difference methods do. The holistic nature of this approach is sometimes criticized. For example, Sutton and Barto write:

> Evolutionary methods do not use the fact that the policy they are searching for is a function from states to actions; they do not notice which states an individual passes through during its lifetime, or which actions it selects. In some cases this information can be misleading (e.g., when states are misperceived) but more often it should enable more efficient search (Sutton and Barto, 1998, p. 9).

These facts can put evolutionary methods at a theoretical disadvantage. For example, in some circumstances, dynamic programming methods are guaranteed to find an optimal policy in time polynomial in the number of states and actions (Littman et al, 1995). By contrast, evolutionary methods, in the worst case, must iterate over an exponential number of candidate policies before finding the best one. Empirical results have also shown that evolutionary methods sometimes require more episodes than temporal-difference methods to find a good policy, especially in highly stochastic tasks in which many Monte Carlo simulations are necessary to achieve a reliable estimate of the fitness of each candidate policy (Runarsson and Lucas, 2005; Lucas and Runarsson, 2006; Lucas and Togelius, 2007; Whiteson et al, 2010b).

However, despite these limitations, evolutionary computation remains a popular tool for solving reinforcement-learning problems and boasts a wide range of empirical successes, sometimes substantially outperforming temporal-difference methods (Whitley et al, 1993; Moriarty and Miikkulainen, 1996; Stanley and Miikkulainen, 2002; Gomez et al, 2008; Whiteson et al, 2010b). There are three main reasons why.

First, evolutionary methods can cope well with partial observability. While evolutionary methods do not exploit the relationship between subsequent states that an agent visits, this can be advantageous when the agent is unsure about its state. Since temporal-difference methods rely explicitly on the Markov property, their value estimates can diverge when it fails to hold, with potentially catastrophic consequences for the performance of the greedy policy. In contrast, evolutionary methods do not rely on the Markov property and will always select the best policies they can find for the given task. Severe partial observability may place a ceiling on the performance of such policies, but optimization within the given policy space proceeds normally (Moriarty et al, 1999). In addition, representations that use memory to reduce partial observability, such as recurrent neural networks, can be be optimized in a natural way with evolutionary methods (Gomez and Miikkulainen, 1999; Stanley and Miikkulainen, 2002; Gomez and Schmidhuber, 2005a,b).

Second, evolutionary methods can make it easier to find suitable representations for the agent's solution. Since policies need only specify an action for each state, instead of the value of each state-action pair, they can be simpler to represent. In addition, it is possible to simultaneously evolve a suitable policy representation (see Sections 10.3 and 10.4.2). Furthermore, since it is not necessary to perform learning updates on a given candidate solution, it is possible to use more elaborate representations, such as those employed by *generative and developmental systems* (GDS) (see Section 10.6).

Third, evolutionary methods provide a simple way to solve problems with large or continuous action spaces. Many temporal-difference methods are ill-suited to such tasks because they require iterating over the action space in each state in order to identify the maximizing action. In contrast, evolutionary methods need only evolve policies that directly map states to actions. Of course, actor-critic methods (Doya, 2000; Peters and Schaal, 2008) and other techniques (Gaskett et al, 1999; Millán et al, 2002; van Hasselt and Wiering, 2007) can also be used to make temporal-difference methods suitable for continuous action spaces. Nonetheless, evolutionary methods provide a simple, effective way to address such difficulties.

Of course, none of these arguments are unique to evolutionary methods, but apply in principle to other policy-search methods too. However, evolutionary methods have proven a particularly popular way to search policy space and, consequently, there is a rich collection of algorithms and results for the reinforcement-learning setting. Furthermore, as modern methods, such as distribution-based approaches, depart further from the original genetic algorithms, their resemblance to the process of natural selection has decreased. Thus, the distinction between evolutionary methods and other policy search approaches has become fuzzier and less important.

This chapter provides an introduction to and overview of evolutionary methods for reinforcement learning. The vastness of the field makes it infeasible to address all the important developments and results. In the interest of clarity and brevity, this chapter focuses heavily on *neuroevolution* (Yao, 1999), in which evolutionary methods are used to evolve *neural networks* (Haykin, 1994), e.g., to represent policies. While evolutionary reinforcement learning is by no means limited to neural-network representations, neuroevolutionary approaches are by far the most common. Furthermore, since neural networks are a popular and well-studied representation in general, they are a suitable object of focus for this chapter.

The rest of this chapter is organized as follows. By way of introduction, Section 10.2 describes a simple neuroevolutionary algorithm for reinforcement learning. Section 10.3 considers *topology- and weight-evolving artificial neural networks* (TWEANNs), including the popular NEAT method, that automatically discover their own internal representations. Section 10.4 considers hybrid approaches, such as *evolutionary function approximation* and *learning classifier systems*, that integrate evolution with temporal-difference methods. Section 10.5 discusses *coevolution*, in which multiple competing and/or cooperating policies are evolved simultaneously. Section 10.6 describes *generative and developmental systems* (GDSs) such as HyperNEAT, which rely on *indirect encodings*: more complex representations in which the agent's policy must be constructed or grown from the evolved parameter values. Section 10.7 discusses on-line methods that strive to maximize reward during evolution instead of merely discovering a good policy quickly.

10.2 Neuroevolution

Neural networks (Haykin, 1994) are an extremely general-purpose way of representing complex functions. Because of their concision, they are a popular representation for reinforcement-learning policies, not only for evolutionary computation, but also for other policy-search algorithms and for temporal-difference methods. This section introduces the basics of neuroevolutionary approaches to reinforcement learning, in which evolutionary methods are used to optimize neural-network policies.

Figure 10.1 illustrates the basic steps of a neuroevolutionary algorithm (Yao, 1999). In each generation, each network in the population is evaluated in the task. Next, the best performing are selected, e.g., via rank-based selection, roulette wheel selection, or tournament selection (Goldberg and Deb, 1991). The selected networks are bred via crossover and mutation and reproduced (Sywerda, 1989; De Jong and Spears, 1991)) to form a new population and the process repeats.

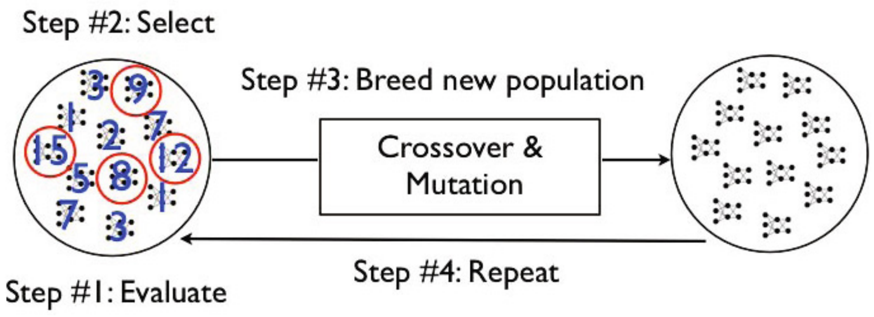

Fig. 10.1 The basic steps of neuroevolution

In a reinforcement-learning setting, each input node in a network typically corresponds to a state feature, such that the value of the inputs together describe the agent's state. There are many ways to represent actions. If the actions can also be described with features, as is common with continuous action spaces, then each output node can correspond to an action feature. In this case, the value of the outputs together describe the action to be selected for the given state. When the number of actions is small and discrete, a separate network can be used for each action, as is common in value-function approximation. In this case, each network has only one output node. The policy's action for a given state then corresponds to the network that produces the largest output for that state.

In this chapter, we focus on a variation of this approach called *action-selection* networks. As before, we assume the set of actions is small and discrete. However, only one network is used to represent the policy. This network has one output node for each action. The policy's action for a given state then corresponds to the node that produces the largest output for that state.

Algorithm 21 contains a high-level description of a simple neuroevolutionary method that evolves action-selection networks for an episodic reinforcement-learning problem. It begins by creating a population of random networks (line 4). In each generation, it repeatedly iterates over the current population (lines 6–7). During each step of a given episode, the agent takes whatever action corresponds to the output with the highest activation (lines 10–12). Note that s' and a' are the current state and action while s and a are the previous state and action. Neuroevolution maintains a running total of the reward accrued by the network during its evaluation (line 13). Each generation ends after e episodes, at which point each network's average fitness is $N.fitness/N.episodes$. In stochastic domains, e typically must be much larger than $|P|$ to ensure accurate fitness estimates for each network. Neuroevolution creates a new population by repeatedly calling the BREED-NET function (line 18), which generates a new network from highly fit parents.

Note that, while the action selection described in lines 10–12 resembles greedy action selection from a value function, the network should not be interpreted as a value function.[1] Evolution does not search for the networks that best approximate the optimal value function. Instead, it searches for networks representing high performing policies. To perform well, a network need only generate more output for the optimal action than for other actions. Unlike with value functions, the scale of the outputs can be arbitrary, as well as the relative outputs of the non-selected actions.

The neural network employed by Algorithm 21 could be a simple *feed-forward* network or a more complex *recurrent* network. Recurrent neural networks can contain cycles (e.g., the output emerging from an output node can be fed into an input node). Consequently, such networks can contain internal state. In a reinforcement-learning context, this internal state can record aspects of the agent's observation history, which can help it cope with partial observability (Wieland, 1991; Gomez and Miikkulainen, 1999; Moriarty et al, 1999; Stanley and Miikkulainen, 2002; Igel, 2003; Gomez and Schmidhuber, 2005a,b).

[1] This does not apply to the hybrid methods discussed in Section 10.4.

```
1: // S: set of all states, A: set of all actions, p: population size
2: // g: number of generations, e: episodes per generation
3:
4: $P \leftarrow$ INIT-POPULATION($S$,$A$,$p$)    // create new population P with random networks
5: for $i \leftarrow 1$ to $g$ do
6:     for $j \leftarrow 1$ to $e$ do
7:         $N,s,s' \leftarrow P[j \% p]$, null, INIT-STATE($S$)    // select next network
8:         repeat
9:             $Q \leftarrow$ EVAL-NET($N$,$s'$)    // evaluate selected network on current state
10:            $a' \leftarrow \text{argmax}_i Q[i]$    // select action with highest activation
11:            $s,a \leftarrow s',a'$
12:            $r,s' \leftarrow$ TAKE-ACTION($a'$)    // take action and transition to new state
13:            $N.fitness \leftarrow N.fitness + r$    // update total reward accrued by N
14:        until TERMINAL-STATE?($s$)
15:        $N.episodes \leftarrow N.episodes + 1$    // update total number of episodes for N
16:    $P' \leftarrow$ new array of size $p$    // new array will store next generation
17:    for $j \leftarrow 1$ to $p$ do
18:        $P'[j] \leftarrow$ BREED-NET($P$)    // make a new network based on fit parents in P
19:    $P \leftarrow P'$
```

Algorithm 21. NEUROEVOLUTION(S,A,p,g,e)

While Algorithm 21 uses a traditional genetic algorithm to optimize neural-network weights, many variations are also possible. For example, *estimation of distribution algorithms* (EDAs) (Larranaga and Lozano, 2002; Hansen et al, 2003; Rubinstein and Kroese, 2004) can be used instead. EDAs, also called *probabilistic model-building genetic algorithms* (PMBGAs), do not explicitly maintain a population of candidate solutions. Instead, they maintain a distribution over solutions. In each generation, candidate solutions are sampled from this distribution and evaluated. A subset is then selected and used to update the distribution using *density estimation* techniques, unsupervised learning techniques for approximating the distribution from which a set of samples was drawn.

One of the most popular and effective EDAs is the *covariance matrix adaptation evolution strategy* (CMA-ES) (Hansen et al, 2003), a variable-metric EDA in which the distribution is a multivariate Gaussian whose covariance matrix adapts over time. When used to optimize neural networks, the resulting method, called CMA-NeuroES, has proven effective on a wide range of reinforcement-learning tasks (Igel, 2003; Heidrich-Meisner and Igel, 2008, 2009a,b,c).

10.3 TWEANNs

In its simplest form, Algorithm 21 evolves only neural networks with fixed representations. In such a setup, all the networks in a particular evolutionary run have the same *topology*, i.e., both the number of hidden nodes and the set of edges connecting the nodes are fixed. The networks differ only with respect to the weights of these edges, which are optimized by evolution. The use of fixed representations is by no means unique to neuroevolution. In fact, though methods exist for automatically discovering good representations for value-functions (Mahadevan and Maggioni, 2007; Parr et al, 2007) temporal-difference methods typically also use fixed representations for function approximation.

Nonetheless, reliance on fixed representations is a significant limitation. The primary reason is that it requires the user of the algorithm to correctly specify a good representation in advance. Clearly, choosing too simple a representation will doom evolution to poor performance, since describing high quality solutions becomes impossible. However, choosing too complex a representation can be just as harmful. While such a representation can still describe good solutions, finding them may become infeasible. Since each weight in the network corresponds to a dimension of the search space, a representation with too many edges can lead to an intractable search problem.

In most tasks, the user is not able to correctly guess the right representation. Even in cases where the user possesses great domain expertise, deducing the right representation from this expertise is often not possible. Typically, finding a good representation becomes a process of trial and error. However, repeatedly running evolution until a suitable representation is found greatly increases computational costs. Furthermore, in on-line tasks (see Section 10.7) it also increases the real-world costs of trying out policies in the target environment.

For these reasons, many researchers have investigated ways to automate the discovery of good representations (Dasgupta and McGregor, 1992; Radcliffe, 1993; Gruau, 1994; Stanley and Miikkulainen, 2002). Evolutionary methods are well suited to this challenge because they take a direct policy-search approach to reinforcement learning. In particular, since neuroevolution already directly searches the space of network weights, it can also simultaneously search the space of network topologies. Methods that do so are sometimes called *topology- and weight-evolving artificial neural networks* (TWEANNs).

Perhaps the earliest and simplest TWEANN is the *structured genetic algorithm* (sGA) (Dasgupta and McGregor, 1992), which uses a two-part representation to describe each network. The first part represents the connectivity of the network in the form of a binary matrix. Rows and columns correspond to nodes in the network and the value of each cell indicates whether an edge exists connecting the given pair of nodes. The second part represents the weights of each edge in the network. In principle, by evolving these binary matrices along with connection weights, sGA can automatically discover suitable network topologies. However, sGA suffers from several limitations. In the following section, we discuss these limitations in order to highlight the main challenges faced by all TWEANNs.

10.3.1 Challenges

There are three main challenges to developing a successful TWEANN. The first is the *competing conventions* problem. In most tasks, there are multiple different policies that have similar fitness. For example, many tasks contain symmetries that give rise to several equivalent solutions. This can lead to difficulties for evolution because of its reliance on crossover operators to breed new networks. When two networks that represent different policies are combined, the result is likely to be destructive, producing a policy that cannot successfully carry out the strategy used by either parent.

While competing conventions can arise in any evolutionary method that uses crossover, the problem is particularly severe for TWEANNs. Two parents may not only implement different policies but also have different representations. Therefore, to be effective, TWEANNs need a mechanism for combining networks with different topologies in a way that minimizes the chance of catastrophic crossover. Clearly, sGA does not meet this challenge, since the binary matrices it evolves are crossed over without regard to incompatibility in representations. In fact, the difficulties posed by the competing conventions problem were a major obstacle for early TWEANNs, to the point that some researchers simply avoided the problem by developing methods that do not perform crossover at all (Radcliffe, 1993).

The second challenge is the need to protect topological innovations long enough to optimize the associated weights. Typically, when new topological structures are introduced (e.g., the addition of a new hidden node or edge), it has a negative effect on fitness even if that structure will eventually be necessary for a good policy. The reason is that the weights associated with the new structure have not yet been optimized.

For example, consider an edge in a network evolved via sGA that is not activated, i.e., its cell in the binary matrix is set to zero. The corresponding weight for that edge will not experience any selective pressure, since it is not manifested in the network. If evolution suddenly activates that edge, the effect on fitness is likely to be detrimental, since its weight is not optimized. Therefore, if topological innovations are not explicitly protected, they will typically be eliminated from the population, causing the search for better topologies to stagnate.

Fortunately, protecting innovation is a well-studied problem in evolutionary computation. *Speciation* and *niching* methods (Holland, 1975; Goldberg and Richardson, 1987; Mahfoud, 1995; Potter and De Jong, 1995; Darwen and Yao, 1996) ensure diversity in the population, typically by segregating disparate individuals and/or penalizing individuals that are too similar to others. However, using such methods requires a distance metric to quantify the differences between individuals. Devising such a metric is difficult for TWEANNs, since it is not clear how to compare networks with different topologies.

The third challenge is how to evolve minimal solutions. As mentioned above, a central motivation for TWEANNs is the desire to avoid optimizing overly complex topologies. However, if evolution is initialized with a population of randomly chosen topologies, as in many TWEANNs, some of these topologies may already be too complex. Thus, at least part of the evolutionary search will be conducted in an unnecessarily high dimensional space. It is possible to explicitly reward smaller solutions by adding size penalties in the fitness function (Zhang and Muhlenbein, 1993). However, there is no principled way to determine the size of the penalties without prior knowledge about the topological complexity required for the task.

10.3.2 NEAT

Perhaps the most popular TWEANN is *neuroevolution of augmenting topologies* (NEAT) (Stanley and Miikkulainen, 2002). In this section, we briefly describe NEAT and illustrate how it addresses the major challenges mentioned above.

NEAT is often used to evolve action selectors, as described in Section 10.2. In fact, NEAT follows the framework described in Algorithm 21 and differs from traditional neuroevolution only in how INIT-POPULATION and BREED-NET are implemented.

To represent networks of varying topologies, NEAT employs a flexible genetic encoding. Each network is described by a list of *edge genes*, each of which describes an edge between two *node genes*. Each edge gene specifies the in-node, the out-node, and the weight of the edge. During mutation, new structure can be introduced to a network via special mutation operators that add new node or edge genes to the network (see Figure 10.2).

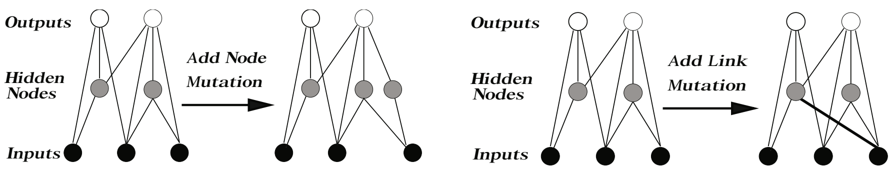

Fig. 10.2 Structural mutation operators in NEAT. At left. a new node is added by splitting an existing edge in two. At right, a new link (edge) is added between two existing nodes.

To avoid catastrophic crossover, NEAT relies on *innovation numbers*, which track the historical origin of each gene. Whenever a new gene appears via mutation, it receives a unique innovation number. Thus, the innovation numbers can be viewed as a chronology of all the genes produced during evolution.

During crossover, innovation numbers are used to determine which genes in the two parents correspond to each other. Genes that do not match are either *disjoint* or *excess*, depending on whether they occur within or outside the range of the other parent's innovation numbers. When crossing over, pairs of genes with the same innovation number (one from each parent) are lined up. Genes that do not match are inherited from the fitter parent. This approach makes it possible for NEAT to minimize the chance of catastrophic crossover without conducting an expensive topological analysis. Since genomes with different topologies nonetheless remain compatible throughout evolution, NEAT essentially avoids the competing conventions problem.

Innovation numbers also make possible a simple way to protect topological innovation. In particular, NEAT uses innovation numbers to speciate the population based on topological similarity. The distance δ between two network encodings is a simple linear combination of the number of excess (E) and disjoint (D) genes, as well as the average weight differences of matching genes ($\overline{W}$):

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W}$$

The coefficients c_1, c_2, and c_3 adjust the importance of the three factors, and the factor N, the number of genes in the larger genome, normalizes for genome size. Networks whose distance is greater than δ_t, a compatibility threshold, are placed into different species. *Explicit fitness sharing* (Goldberg, 1989), in which networks in the same species must share the fitness of their niche, is employed to protect innovative species.

To encourage the evolution of minimal solutions, NEAT begins with a uniform population of simple networks with no hidden nodes and inputs connected directly to outputs. New structure is introduced incrementally via the mutation operators that add new hidden nodes and edges to the network. Since only the structural mutations that yield performance advantages tend to survive evolution's selective pressure, minimal solutions are favored.

NEAT has amassed numerous empirical successes on difficult reinforcement-learning tasks like non-Markovian double pole balancing (Stanley and Miikkulainen, 2002), game playing (Stanley and Miikkulainen, 2004b), and robot control (Stanley and Miikkulainen, 2004a; Taylor et al, 2006; Whiteson et al, 2010b). However, Kohl and Miikkulainen (2008, 2009) have shown that NEAT can perform poorly on tasks in which the optimal action varies discontinuously across states. They demonstrate that these problems can be mitigated by providing neurons with local receptive fields and constraining topology search to cascade structures

10.4 Hybrids

Many researchers have investigated hybrid methods that combine evolution with supervised or unsupervised learning methods. In such systems, the individuals being evolved do not remain fixed during their fitness evaluations. Instead, they change during their 'lifetimes' by learning from the environments with which they interact.

Much of the research on hybrid methods focuses on analyzing the dynamics that result when evolution and learning interact. For example, several studies (Whitley et al, 1994; Yamasaki and Sekiguchi, 2000; Pereira and Costa, 2001; Whiteson and Stone, 2006a) have used hybrids to compare *Lamarckian* and *Darwinian* systems.

In Lamarckian systems, the phenotypic effects of learning are copied back into the genome before reproduction, allowing new offspring to inherit them. In Darwinian systems, which more closely model biology, learning does not affect the genome. As other hybrid studies (Hinton and Nowlan, 1987; French and Messinger, 1994; Arita and Suzuki, 2000) have shown, Darwinian systems can indirectly transfer the results of learning into the genome by way of the *Baldwin effect* (Baldwin, 1896), in which learning creates selective pressures favoring individuals who innately possess attributes that were previously learned.

Hybrid methods have also been employed to improve performance on supervised learning tasks (Gruau and Whitley, 1993; Boers et al, 1995; Giraud-Carrier, 2000; Schmidhuber et al, 2005, 2007). However, such methods are not directly applicable to reinforcement-learning problems because the labeled data they require is absent.

Nonetheless, many hybrid methods for reinforcement learning have been developed. To get around the problem of missing labels, researchers have employed unsupervised learning (Stanley et al, 2003), trained individuals to resemble their parents (McQuesten and Miikkulainen, 1997), trained them to predict state transitions (Nolfi et al, 1994), and trained them to teach themselves (Nolfi and Parisi, 1997). However, perhaps the most natural hybrids for the reinforcement learning setting are combination of evolution with temporal-difference methods (Ackley and Littman, 1991; Wilson, 1995; Downing, 2001; Whiteson and Stone, 2006a). In this section, we survey two such hybrids: *evolutionary function approximation* and XCS, a type of *learning classifier system*.

10.4.1 Evolutionary Function Approximation

Evolutionary function approximation (Whiteson and Stone, 2006a), is a way to synthesize evolutionary and temporal-difference methods into a single method that automatically selects function approximator representations that enable efficient individual learning. The main idea is that, if evolution is directed to evolve value functions instead of action selectors, then those value functions can be updated, using temporal-difference methods, during each fitness evaluation. In this way, the system can *evolve* function approximators that are better able to *learn* via temporal-difference methods. This biologically intuitive combination, which has been applied to many computational systems (Hinton and Nowlan, 1987; Ackley and Littman, 1991; Boers et al, 1995; French and Messinger, 1994; Gruau and Whitley, 1993; Nolfi et al, 1994), can yield effective reinforcement-learning algorithms. In this section, we briefly describe NEAT+Q, an evolutionary function approximation technique resulting from the combination of NEAT and Q-learning with neural-network function approximation.

To make NEAT optimize value functions instead of action selectors, all that is required is a reinterpretation of its output values. The structure of neural-network action selectors (one input for each state feature and one output for each action) is already identical to that of Q-learning function approximators. Therefore, if the weights of the networks NEAT evolves are updated during their fitness evaluations using Q-learning and backpropagation, they will effectively evolve value functions instead of action selectors. Hence, the outputs are no longer arbitrary values; they represent the long-term discounted values of the associated state-action pairs and are used, not just to select the most desirable action, but to update the estimates of other state-action pairs.

Algorithm 22 shows the inner loop of NEAT+Q, replacing lines 9–13 in Algorithm 21. Each time the agent takes an action, the network is backpropagated towards Q-learning targets (line 7) and ε-greedy selection occurs (lines 4–5). Figure 10.3 illustrates the complete algorithm: networks are selected from the population for evaluation and the Q-values they produce are used to select actions. The resulting feedback from the environment is used both to perform TD updates and to measure the network's fitness, i.e., the total reward it accrues while learning.

```
1: // $\alpha$: learning rate, $\gamma$: discount factor, $\lambda$: eligibility decay rate, $\varepsilon$: exploration rate
2:
3: $Q \leftarrow$ EVAL-NET($N$,$s'$)                      // compute value estimates for current state
4: with-prob($\varepsilon$) $a' \leftarrow$ RANDOM($A$)          // select random exploratory action
5: else $a' \leftarrow \text{argmax}_k Q[k]$                  // or select greedy action
6: if $s \neq$ null then
7:     BACKPROP($N$,$s$,$a$,$(r+\gamma \max_k Q[k])$,$\alpha$,$\gamma$,$\lambda$)    // adjust weights
8: $s,a \leftarrow s',a'$
9: $r,s' \leftarrow$ TAKE-ACTION($a'$)                 // take action and transition to new state
10: $N.fitness \leftarrow N.fitness + r$              // update total reward accrued by N
```

Algorithm 22. NEAT+Q (inner loop)

Like other hybrid methods, NEAT+Q combines the advantages of temporal-difference methods with those of evolution. In particular, it harnesses the ability of NEAT to discover effective representations and uses it to aid neural-network value-function approximation. Unlike traditional neural-network function approximators, which put all their eggs in one basket by relying on a single manually designed network to represent the value function, NEAT+Q explores the space of such networks to increase the chance of finding a high performing representation. As a result, on certain tasks, this approach has been shown to significantly outperform both temporal-difference methods and neuroevolution on their own (Whiteson and Stone, 2006a).

10.4.2 XCS

A different type of hybrid method can be constructed using *learning classifier systems* (LCSs) (Holland, 1975; Holland and Reitman, 1977; Bull and Kovacs, 2005; Butz, 2006; Drugowitsch, 2008). An LCS is an evolutionary system that uses rules,

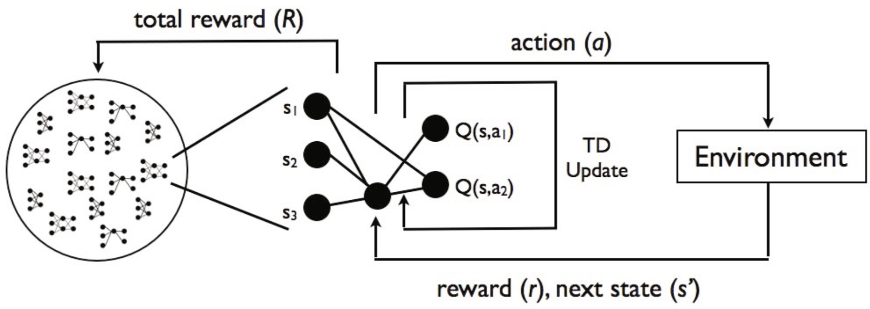

Fig. 10.3 The NEAT+Q algorithm

called *classifiers*, to represent solutions. A wide range of classification, regression, and optimization problems can be tackled using such systems. These include reinforcement learning problems, in which case a set of classifiers describes a policy.

Each classifier contains a *condition* that describes the set of states to which the classifier applies. Conditions can be specified in many ways (e.g., fuzzy conditions (Bonarini, 2000) or neural network conditions (Bull and O'Hara, 2002)) but perhaps the simplest is the ternary alphabet used for binary state features: 0, 1, and #, where # indicates "don't care". For example, a classifier with condition 01#10 applies to the states 01010 and 01110. Classifiers also contain *actions*, which specify what action the agent should take in states to which the classifier applies, and *predictions* which estimate the corresponding action-value function.

In *Pittsburgh-style* classifier systems (De Jong et al, 1993), each individual in the evolving population consists of an entire rule set describing a complete policy. These methods can be viewed as standard evolutionary approaches to reinforcement learning with rule sets as a way to represent policies, i.e., in lieu of neural networks. In contrast, in *Michigan-style* classifier systems, each individual consists of only one rule. Thus, the entire population represents a single, evolving policy. In the remainder of this section, we give a brief overview of *XCS* (Wilson, 1995, 2001; Butz et al, 2008), one of the most popular Michigan-style classifiers. Since prediction updates in XCS are based on Q-learning, it can be viewed as a hybrid between temporal-difference methods and evolutionary reinforcement-learning algorithms.

In XCS, each classifier's prediction contributes to an estimate of the Q-values of the state-action pairs to which the classifier applies. $Q(s,a)$ is the weighted average of the predictions of all the *matching* classifiers, i.e., those that apply to s and have action a. The fitness of each classifier determines its weight in the average (fitness is defined later in this section). In other words:

$$Q(s,a) = \frac{\sum_{c \in M(s,a)} c.f \cdot c.p}{\sum_{c \in M(s,a)} c.f},$$

where $M(s,a)$ is the set of all classifiers matching s and a; $c.f$ and $c.p$ are the fitness and prediction, respectively, of classifier c.

Each time the agent is in state s, takes action a, receives reward r, and transitions to state s', the following update rule is applied to each $c \in M(s,a)$:

$$c.p \leftarrow c.p + \beta[r + \gamma max_{a'} Q(s',a') - c.p] \frac{c.f}{\sum_{c' \in M(s,a)} c'.f},$$

where β is a learning rate parameter. This is essentially a Q-learning update, except that the size of the update is scaled according to the relative fitness of c, since this determines its contribution to the Q-value.

Many LCS systems use *strength-based* updates, wherein each classifier's fitness is based on its prediction, i.e., classifiers that expect to obtain more reward are favored by evolution. However, this can lead to the problem of *strong over-generals* (Kovacs, 2003), which are general classifiers that have high overall value but select suboptimal actions for a subset of states. With strength-based updates, such classifiers are favored over more specific ones that select better actions for that same subset but have lower overall value.

To avoid this problem, XCS uses *accuracy-based* updates, in which a classifier's fitness is inversely proportional to an estimate of the error in its prediction. This error $c.\varepsilon$ is updated based on the absolute value of the temporal-difference error used in the Q-learning update:

$$c.\varepsilon \leftarrow c.\varepsilon + \beta(|r + \gamma max_{a'} Q(s',a') - c.p| - c.\varepsilon)$$

Classifier accuracy is then defined in terms of this error. Specifically, when $c.\varepsilon > \varepsilon_0$, a minimum error threshold, the accuracy of c is defined as:

$$c.\kappa = \alpha(c.\varepsilon/\varepsilon_0)^{-\eta},$$

where α and η are accuracy parameters. When $c.\varepsilon \leq \varepsilon_0$, $c.\kappa = 1$.

However, fitness is computed, not with this accuracy, but instead with the *set-relative accuracy*: the accuracy divided by the sum of the accuracies of all matching classifiers. This yields the following fitness update:

$$c.f \leftarrow c.f + \beta(\frac{c.\kappa}{\sum_{c' \in M(s,a)} c'.\kappa} - c.f)$$

At every timestep, the prediction, error and fitness of each matching classifier are updated. Since XCS is a *steady-state* evolutionary method, there are no generations. Instead, the population changes incrementally through the periodic selection and reproduction of a few fit classifiers, which replace a few weak classifiers and leave the rest of the population unchanged. When selection occurs, only classifiers that match the current state and action are considered. New classifiers are created from the selected ones using crossover and mutation, and weak classifiers (chosen from the whole population, not just the matching ones) are deleted to make room.

Thanks to the Q-learning updates, the accuracy of the classifiers tends to improve over time. Thanks to steady-state evolution, the most accurate classifiers are selectively bred. General rules tend to have higher error (since they generalize over more states) and thus lower accuracy. It might seem that, as a result, XCS will evolve only highly specific rules. However, more general rules also match more often. Since only matching classifiers can reproduce, XCS balances the pressure for specific rules with pressure for general rules. Thus, it strives to learn a complete, maximally general, and accurate set of classifiers for approximating the optimal Q-function.

Though there are no convergence proofs for XCS on MDPs, it has proven empirically effective on many tasks. For example, on maze tasks, it has proven adept at automatically discovering what state features to ignore (Butz et al, 2005) and solving problems with more than a million states (Butz and Lanzi, 2009). It has also proven adept at complex sensorimotor control (Butz and Herbort, 2008; Butz et al, 2009) and autonomous robotics (Dorigo and Colombetti, 1998).

10.5 Coevolution

Coevolution is a concept from evolutionary biology that refers to the interactions between multiple individuals that are simultaneously evolving. In other words, coevolution occurs when the fitness function of one individual depends on other individuals that are also evolving. In nature, this can occur when different populations interact, e.g., cheetahs and the gazelles they hunt, or within a population, e.g., members of the same species competing for mates. Furthermore, coevolution can be cooperative, e.g., humans and the bacteria in our digestive systems, or competitive, e.g., predator and prey. All these forms of coevolution have been investigated and exploited in evolutionary reinforcement learning, as surveyed in this section.

10.5.1 Cooperative Coevolution

The most obvious application of cooperative coevolution is to cooperative multi-agent systems (Panait and Luke, 2005) which, in the context of reinforcement learning, means evolving teams of agents that coordinate their behavior to solve a sequential decision problem. In principle, such problems can be solved without coevolution by using a monolithic approach: evolving a population in which each individual specifies the policy for every agent on the team. However, such an approach quickly becomes intractable, as the size of the search space grows exponentially with respect to the number of agents.

One of the primary motivations for a coevolutionary approach is that it can help address this difficulty (Wiegand et al, 2001; Jansen and Wiegand, 2004; Panait et al, 2006). As Gomez et al. put it, "many problems may be decomposable into weakly coupled low-dimensional subspaces that can be searched semi-independently by separate species" (Gomez et al, 2008). In general, identifying these low-dimensional subspaces requires a lot of domain knowledge. However, in multi-agent problems, it is often sufficient to divide the problem up by agent, i.e., evolve one population for each agent on the team, in order to make evolution tractable. In this approach, one member of each population is selected, often randomly, to form a team that is then evaluated in the task. The total reward obtained contributes to an estimate of the fitness of each participating agent, which is typically evaluated multiple times.

While this approach often outperforms monolithic evolution and has found success in predator-prey (Yong and Miikkulainen, 2007) and robot-control (Cai and Peng, 2002) applications, it also runs into difficulties when there are large numbers of agents. The main problem is that the contribution of a single agent to the total reward accrued becomes insignificant. Thus, the fitness an agent receives depends more on which teammates it is evaluated with than on its own policy. However, it is possible to construct special fitness functions for individual agents that are much less sensitive to such effects (Agogino and Tumer, 2008). The main idea is to use *difference functions* (Wolpert and Tumer, 2002) that compare the total reward the team obtains when the agent is present to when it is absent or replaced by a fixed baseline agent. While this approach requires access to a model of the environment and increases the computational cost of fitness evaluation (so that the reward in both scenarios can be measured), it can dramatically improve the performance of cooperative coevolution.

Coevolution can also be used to simultaneously evolve multiple components of a single agent, instead of multiple agents. For example, in the task of robot soccer keepaway, domain knowledge has been used to decompose the task into different components, each representing an important skill such as running towards the ball or getting open for a pass (Whiteson et al, 2005). Neural networks for each of these components are then coevolved and together comprise a complete policy. In the keepaway task, coevolution greatly outperforms a monolithic approach.

Cooperative coevolution can also be used in a single-agent setting to facilitate neuroevolution. Rather than coevolving multiple networks, with one for each member of a team or each component of a policy, *neurons* are coevolved, which together form a single network describing the agent's policy (Potter and De Jong, 1995, 2000). Typically, networks have fixed topologies with a single hidden layer and each neuron corresponds to a hidden node, including all the weights of its incoming and outgoing edges. Just as dividing a multi-agent task up by agent often leads to simpler subproblems, so too can breaking up a neuroevolutionary task by neuron. As Moriarty and Mikkulainen say,"neuron-level evolution takes advantage of the *a priori* knowledge that individual neurons constitute basic components of neural networks" (Moriarty and Miikkulainen, 1997).

One example is *symbiotic adaptive neuroevolution* (SANE) (Moriarty and Miikkulainen, 1996, 1997) in which evolution occurs simultaneously at two levels. At the lower level, a single population of neurons is evolved. The fitness of each neuron is the average performance of the networks in which it participates during fitness evaluations. At the higher level, a population of *blueprints* is evolved, with each blueprint consisting of a vector of pointers to neurons in the lower level. The blueprints that combine neurons into the most effective networks tend to survive selective pressure. On various reinforcement-learning tasks such as robot control and pole balancing, SANE has outperformed temporal-difference methods, monolithic neuroevolution, and neuron-level evolution without blueprints.

The *enforced subpopulations* (ESP) method (Gomez and Miikkulainen, 1999) eliminates the blueprint population but segregates neurons into subpopulations, one for each hidden node. One member of each population is selected randomly to form a network for each fitness evaluation. This encourages subpopulations to take on different specialized roles, increasing the likelihood that effective networks will be formed even without blueprints. ESP has performed particularly well on partially observable tasks, solving a non-Markovian version of the double pole-balancing problem. In addition H-ESP, a hierarchical variant that does use blueprints, has proven successful on *deep memory POMDPs*, i.e., those requiring history of hundreds or thousands of timesteps (Gomez and Schmidhuber, 2005a). ESP has even been used to evolve control systems for a finless sounding rocket (Gomez and Miikkulainen, 2003).

In *cooperative synapse neuroevolution* (CoSyNE) (Gomez et al, 2006, 2008), the idea of separate subpopulations is taken even further. Rather than a subpopulation for every neuron, which contains multiple weights, CoSyNE has a subpopulation for each edge, which has only one weight (see Figure 10.4). Thus the problem of finding a complete network is broken down into atomic units, solutions to which are coevolved. On several versions of the pole balancing problem, CoSyNE has been shown to outperform various temporal-difference methods and other policy-search approaches, as well as SANE, ESP, and NEAT (Gomez et al, 2006, 2008). However, in a more recent study, CMA-NeuroES (see Section 10.2) performed even better (Heidrich-Meisner and Igel, 2009b).

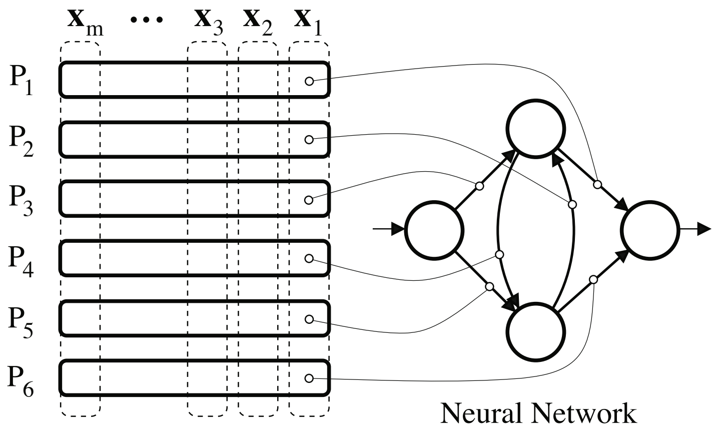
$\mathbf{x}_m$ ... $\mathbf{x}_3$ $\mathbf{x}_2$ $\mathbf{x}_1$
P_1
P_2
P_3
P_4
P_5
P_6
Neural Network

Fig. 10.4 The CoSyNE algorithm, using six subpopulations, each containing m weights. All the weights at a given index i form a genotype $\mathbf{x}_i$. Each weight is taken from a different subpopulation and describes one edge in the neural network. Figure taken with permission from (Gomez et al, 2008).

10.5.2 Competitive Coevolution

Coevolution has also proven a powerful tool for competitive settings. The most common applications are in games, in which coevolution is used to simultaneously evolve strong players and the opponents against which they are evaluated. The hope is to create an *arms race* (Dawkins and Krebs, 1979) in which the evolving agents exert continual selective pressure on each other, driving evolution towards increasingly effective policies.

Perhaps the simplest example of competitive coevolution is the work of Pollack and Blair in the game of backgammon (Pollack and Blair, 1998). Their approach relies on a simple optimization technique (essentially an evolutionary method with a population size of two) wherein a neural network plays against a mutated version of itself and the winner survives. The approach works so well that Pollack and Blair hypothesize that Tesauro's great success with TD-Gammon (Tesauro, 1994) is due more to the nature of backgammon than the power of temporal-difference methods.[2]

Using larger populations, competitive coevolution has also found success in the game of checkers. The Blondie24 program uses the *minimax* algorithm (Von Neumann, 1928) to play checkers, relying on neuroevolution to discover an effective evaluator of board positions (Chellapilla and Fogel, 2001). During fitness evaluations, members of the current population play games against each other. Despite the minimal use of human expertise, Blondie24 evolved to play at a level competitive with human experts.

Competitive coevolution can also have useful synergies with TWEANNs. In fixed-topology neuroevolution, arms races may be cut short when additional improvement requires an expanded representation. Since TWEANNs can automatically expand their representations, coevolution can give rise to *continual complexification* (Stanley and Miikkulainen, 2004a).

The methods mentioned above evolve only a single population. However, as in cooperative coevolution, better performance is sometimes possible by segregating individuals into separate populations. In the *host/parasite* model (Hillis, 1990), one population evolves *hosts* and another *parasites*. Hosts are evaluated based on their robustness against parasites, e.g., how many parasites they beat in games of checkers. In contrast, parasites are evaluated based on their uniqueness, e.g., how many hosts they can beat that other parasites cannot. Such fitness functions can be implemented using *competitive fitness sharing* (Rosin and Belew, 1997).

In *Pareto coevolution*, the problem is treated as a multi-objective one, with each opponent as an objective (Ficici and Pollack, 2000, 2001). The goal is thus to find a *Pareto-optimal solution*, i.e., one that cannot be improved with respect to one objective without worsening its performance with respect to another. Using this approach, many methods have been developed that maintain *Pareto archives* of

[2] Tesauro, however, disputes this claim, pointing out that the performance difference between Pollack and Blair's approach and his own is quite significant, analogous to that between an average human player and a world-class one (Tesauro, 1998).

opponents against which to evaluate evolving solutions (De Jong, 2004; Monroy et al, 2006; De Jong, 2007; Popovici et al, 2010).

10.6 Generative and Developmental Systems

All of the evolutionary reinforcement-learning methods described above rely on *direct encodings* to represent policies. In such representations, evolution optimizes a *genotype* (e.g., a vector of numbers specifying the weights of a neural network) that can be trivially transformed into a *phenotype* (e.g., the neural network itself). While the simplicity of such an approach is appealing, it has limited scalability. Since the genotype is always as large as the phenotype, evolving the complex policies required to solve many realistic tasks requires searching a high dimensional space.

Generative and developmental systems (Gruau, 1994; Hornby and Pollack, 2002; Stanley and Miikkulainen, 2003; Stanley et al, 2009) is a subfield of evolutionary computation that focuses on evolving *indirect encodings*. In such representations, the phenotype is ‘grown’ via a developmental process specified by the genotype. In many cases, good policies possess simplifying regularities such as symmetry and repetition, which allow for genotypes that are much smaller than the phenotypes they produce. Consequently, searching genotype space is more feasible, making it possible to scale to larger reinforcement-learning tasks.

Furthermore, indirect encodings often provide a natural way to exploit a task’s *geometry*, i.e., the spatial relationship between state features. In most direct encodings, such geometry cannot be exploited because it is not captured in the representation. For example, consider a neural network in which each input describes the current state of one square on a chess board. Since these inputs are treated as an unordered set, the distance between squares is not captured in the representation. Thus, structures for exploiting the relationship between squares must be evolved, complicating the task. In contrast, an indirect encoding can describe a network where the structure for processing each square’s state is a function of that square’s position on the board, with the natural consequence that nearby squares are processed similarly.

Like other evolutionary methods, systems using indirect encodings are inspired by analogies with biological systems: e.g., human beings have trillions of cells in their bodies but develop from genomes containing only tens of thousands of genes. Thus, it is not surprising that many indirect encodings are built on models of natural development. For example, *L-systems* (Lindenmayer, 1968), which are formal grammars for describing complex recursive structures, have been used to evolve both the morphology and control system for autonomous robots, greatly outperforming direct encodings (Hornby and Pollack, 2002). Similarly, *cellular encodings* (Gruau and Whitley, 1993; Gruau, 1994) evolve graph grammars for generating modular neural networks composed of simpler subnetworks. This approach allows evolution to exploit regularities in the solution structure by instantiating multiple copies of the same subnetwork in order to build a complete network.

More recently, the HyperNEAT method (Stanley et al, 2009) has been developed to extend NEAT to use indirect encodings. This approach is based on *compositional patten producing networks* (CPPNs). CPPNs are neural networks for describing complex patterns. For example, a two-dimensional image can be described by a CPPN whose inputs correspond to an *x*-*y* position in the image and whose output corresponds to the color that should appear in that position. The image can then be generated by querying the CPPN at each *x*-*y* position and setting that position's color based on the output. Such CPPNs can be evolved by NEAT, yielding a developmental system with the CPPN as the genotype and the image as the phenotype.

In HyperNEAT, the CPPN is used to describe a neural network instead of an image. Thus, both the genotype and phenotype are neural networks. As illustrated in Figure 10.5, the nodes of the phenotypic network are laid out on a *substrate*, i.e., a grid, such that each has a position. The CPPN takes as input two positions instead of one and its output specifies the weight of the edge connecting the two corresponding nodes. As before, these CPPNs can be evolved by NEAT based on the fitness of the resulting phenotypic network, e.g., its performance as a policy in a reinforcement learning task. The CPPNs can be interpreted as describing a spatial pattern in a four-dimensional hypercube, yielding the name HyperNEAT. Because the developmental approach makes it easy to specify networks that exploit symmetries and regularities in complex tasks, HyperNEAT has proven an effective tool for reinforcement learning, with successful applications in domains such as checkers (Gauci and Stanley, 2008, 2010), keepaway soccer (Verbancsics and Stanley, 2010), and multi-agent systems (D'Ambrosio et al, 2010).

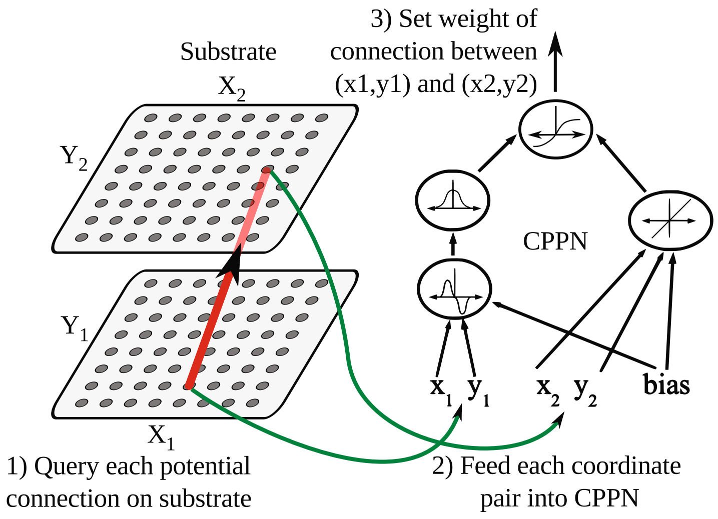
3) Set weight of connection between (x1,y1) and (x2,y2)
Substrate
X2
Y2
Y1
X1
CPPN
x1 y1
x2 y2
bias
1) Query each potential connection on substrate
2) Feed each coordinate pair into CPPN

Fig. 10.5 The HyperNEAT algorithm. Figure taken with permission from (Gauci and Stanley, 2010).

10.7 On-Line Methods

While evolutionary methods have excelled in many challenging reinforcement-learning problems, their empirical success is largely restricted to *off-line* scenarios, in which the agent learns, not in the real-world, but in a 'safe' environment like a simulator. In other words, the problem specification usually includes a fitness function that requires only computational resources to evaluate, as in other optimization problems tackled with evolutionary computation.

In off-line scenarios, an agent's only goal is to learn a good policy as quickly as possible. How much reward it obtains *while it is learning* is irrelevant because those rewards are only hypothetical and do not correspond to real-world costs. If the agent tries disastrous policies, only computation time is lost.

While efficient off-line learning is an important goal, it has limited practical applicability. In many cases, no simulator is available because the dynamics of the task are unknown, e.g., when a robot explores an unfamiliar environment or a chess player plays a new opponent. Other times, the dynamics of the task are too complex to accurately simulate, e.g., user behavior on a large computer network or the noise in a robot's sensors and actuators.

Therefore, many researchers consider *on-line* learning a fundamental challenge in reinforcement learning. In an on-line learning scenario, the agent must maximize the reward it accrues while it is learning because those rewards correspond to real-world costs. For example, if a robot learning on-line tries a policy that causes it to drive off a cliff, then the negative reward the agent receives is not hypothetical; it corresponds to the real cost of fixing or replacing the robot.

Evolutionary methods have also succeeded on-line (Steels, 1994; Nordin and Banzhaf, 1997; Schroder et al, 2001), especially in evolutionary robotics (Meyer et al, 1998; Floreano and Urzelai, 2001; Floreano and Mondada, 2002; Pratihar, 2003; Kernbach et al, 2009; Zufferey et al, 2010), and some research has investigated customizing such methods to on-line settings (Floreano and Urzelai, 2001; Whiteson and Stone, 2006a,b; Priesterjahn et al, 2008; Tan et al, 2008; Cardamone et al, 2009, 2010).

Nonetheless, in most applications, researchers typically report performance using only the off-line measures common for optimization problems, e.g., the number of fitness evaluations needed to find a policy achieving a threshold performance or the performance of the best policy found after a given number of fitness evaluations. Therefore, determining how best to use evolutionary methods for reinforcement learning in on-line settings, i.e., how to maximize cumulative reward during evolution, remains an important and under-explored research area.

10.7.1 Model-Based Methods

One possible approach is to use evolution, not as a complete solution method, but as a component in a model-based method. In model-based algorithms, the agent's interactions with its environment are used to learn a model, to which planning methods are then applied. As the agent gathers more samples from the environment, the quality of the model improves, which, in turn, improves the quality of the policy produced via planning. Because planning is done off-line, the number of interactions needed to find a good policy is minimized, leading to strong on-line performance.

In such an approach, planning is typically conducted using dynamic programming methods like value iteration. However, many other methods can be used instead; if the model is continuous and/or high dimensional, evolutionary or other policy-search methods may be preferable. Unfortunately, most model-based methods are designed only to learn tabular models for small, discrete state spaces. Still, in some cases, especially when considerable domain expertise is available, more complex models can be learned.

For example, linear regression has been used learn models of helicopter dynamics, which can then be used for policy-search reinforcement learning (Ng et al, 2004). The resulting policies have successfully controlled real model helicopters. A similar approach was used to maximize on-line performance in the helicopter-hovering events in recent Reinforcement Learning Competitions (Whiteson et al, 2010a): models learned via linear regression were used as fitness functions for policies evolved off-line via neuroevolution (Koppejan and Whiteson, 2009).

Alternatively, evolutionary methods can be used for the model-learning component of a model-based solution. In particular, *anticipatory learning classifier systems* (Butz, 2002; Gerard et al, 2002, 2005; Sigaud et al, 2009), a type of LCS, can be used to evolve models of the environment that are used for planning in a framework similar to Dyna-Q (Sutton, 1990).

10.7.2 On-Line Evolutionary Computation

Another possible solution is *on-line evolutionary computation* (Whiteson and Stone, 2006a,b). The main idea is to borrow exploration strategies commonly used to select actions in temporal-difference methods and use them to select policies for evaluation in evolution. Doing so allows evolution to balance exploration and exploitation in a way that improves on-line performance.

Of course, evolutionary methods already strive to balance exploration and exploitation. In fact, this is one of the main motivations originally provided for genetic algorithms (Holland, 1975). However, this balance typically occurs only *across* generations, not *within* them. Once the members of each generation have been determined, they all typically receive the same evaluation time.

This approach makes sense in deterministic domains, where each member of the population can be accurately evaluated in a single episode. However, many real-world domains are stochastic, in which case fitness evaluations must be averaged over many episodes. In these domains, giving the same evaluation time to each member of the population can be grossly suboptimal because, within a generation, it is purely exploratory.

Instead, on-line evolutionary computation exploits information gained earlier in the generation to systematically give more evaluations to more promising policies and avoid re-evaluating weaker ones. This is achieved by employing temporal-difference exploration strategies to select policies for evaluation in each generation.

For example, ε-greedy selection can be used at the beginning of each episode to select a policy for evaluation. Instead of iterating through the population, evolution selects a policy randomly with probability ε. With probability $1-\varepsilon$, the algorithm selects the best policy discovered so far in the current generation. The fitness of each policy is just the average reward per episode it has received so far. Each time a policy is selected for evaluation, the total reward it receives is incorporated into that average, which can cause it to gain or lose the rank of best policy.

For the most part, ε-greedy selection does not alter evolution's search but simply interleaves it with exploitative episodes that increase average reward during learning. However, softmax selection can also be used to focus exploration on the most promising alternatives. At the beginning of each generation, each individual is evaluated for one episode, to initialize its fitness. Then, the remaining $e-|P|$ episodes are allocated according to a Boltzmann distribution.

Neither ε-greedy nor softmax consider the uncertainty of the estimates on which they base their selections, a shortcoming that can be addressed with interval estimation (Kaelbling, 1993). When used in temporal-difference methods, interval estimation computes a $(100-\alpha)\%$ confidence interval for the value of each available action. The agent always takes the action with the highest upper bound on this interval. This strategy favors actions with high estimated value and also focuses exploration on promising but uncertain actions. The α parameter controls the balance between exploration and exploitation, with smaller values generating greater exploration.

The same strategy can be employed within evolution to select policies for evaluation. At the beginning of each generation, each individual is evaluated for one episode, to initialize its fitness. Then, the remaining $e-|P|$ episodes are allocated to the policy that currently has the highest upper bound on its confidence interval.

All three of these implementations of on-line evolutionary computation have been shown to substantially improve on-line performance, both in conjunction with NEAT (Whiteson and Stone, 2006b; Cardamone et al, 2009, 2010) and NEAT+Q (Whiteson and Stone, 2006a).

10.8 Conclusion

Evolutionary methods are a powerful tool for tackling challenging reinforcement learning problems. They are especially appealing for problems that include partial observability, have continuous action spaces, or where effective representations cannot be manually specified. Particularly in the area of neuroevolution, sophisticated methods exist for evolving neural-network topologies, decomposing the task based on network structure, and exploiting indirect encodings. Thanks to hybrid methods, the use of evolutionary computation does not require forgoing the power of temporal-difference methods. Furthermore, coevolutionary approaches extend the reach of evolution to multi-agent reinforcement learning, both cooperative and competitive. While most work in evolutionary computation has focused on off-line settings, promising research exists in developing evolutionary methods for on-line reinforcement learning, which remains a critical and exciting challenge for future work.

Acknowledgements. Thanks to Ken Stanley, Risto Miikkulainen, Jürgen Schmidhuber, Martin Butz, Julian Bishop, and the anonymous reviewers for their invaluable input regarding the state of the art in evolutionary reinforcement learning.

References

Ackley, D., Littman, M.: Interactions between learning and evolution. Artificial Life II, SFI Studies in the Sciences of Complexity 10, 487–509 (1991)

Agogino, A.K., Tumer, K.: Efficient evaluation functions for evolving coordination. Evolutionary Computation 16(2), 257–288 (2008)

Arita, T., Suzuki, R.: Interactions between learning and evolution: The outstanding strategy generated by the Baldwin Effect. Artificial Life 7, 196–205 (2000)

Baldwin, J.M.: A new factor in evolution. The American Naturalist 30, 441–451 (1896)

Boers, E., Borst, M., Sprinkhuizen-Kuyper, I.: Evolving Artificial Neural Networks using the “Baldwin Effect”. In: Proceedings of the International Conference Artificial Neural Nets and Genetic Algorithms in Ales, France (1995)

Bonarini, A.: An introduction to learning fuzzy classifier systems. Learning Classifier Systems, 83–104 (2000)

Bull, L., Kovacs, T.: Foundations of learning classifier systems: An introduction. Foundations of Learning Classifier Systems, 1–17 (2005)

Bull, L., O’Hara, T.: Accuracy-based neuro and neuro-fuzzy classifier systems. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 905–911 (2002)

Butz, M.: Anticipatory learning classifier systems. Kluwer Academic Publishers (2002)

Butz, M.: Rule-based evolutionary online learning systems: A principled approach to LCS analysis and design. Springer, Heidelberg (2006)

Butz, M., Herbort, O.: Context-dependent predictions and cognitive arm control with XCSF. In: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation, pp. 1357–1364. ACM (2008)

Butz, M., Lanzi, P.: Sequential problems that test generalization in learning classifier systems. Evolutionary Intelligence 2(3), 141–147 (2009)

Butz, M., Goldberg, D., Lanzi, P.: Gradient descent methods in learning classifier systems: Improving XCS performance in multistep problems. IEEE Transactions on Evolutionary Computation 9(5) (2005)

Butz, M., Lanzi, P., Wilson, S.: Function approximation with XCS: Hyperellipsoidal conditions, recursive least squares, and compaction. IEEE Transactions on Evolutionary Computation 12(3), 355–376 (2008)

Butz, M., Pedersen, G., Stalph, P.: Learning sensorimotor control structures with XCSF: Redundancy exploitation and dynamic control. In: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, pp. 1171–1178 (2009)

Cai, Z., Peng, Z.: Cooperative coevolutionary adaptive genetic algorithm in path planning of cooperative multi-mobile robot systems. Journal of Intelligent and Robotic Systems 33(1), 61–71 (2002)

Cardamone, L., Loiacono, D., Lanzi, P.: On-line neuroevolution applied to the open racing car simulator. In: Proceedings of the Congress on Evolutionary Computation (CEC), pp. 2622–2629 (2009)

Cardamone, L., Loiacono, D., Lanzi, P.L.: Learning to drive in the open racing car simulator using online neuroevolution. IEEE Transactions on Computational Intelligence and AI in Games 2(3), 176–190 (2010)

Chellapilla, K., Fogel, D.: Evolving an expert checkers playing program without using human expertise. IEEE Transactions on Evolutionary Computation 5(4), 422–428 (2001)

Coello, C., Lamont, G., Van Veldhuizen, D.: Evolutionary algorithms for solving multi-objective problems. Springer, Heidelberg (2007)

D'Ambrosio, D., Lehman, J., Risi, S., Stanley, K.O.: Evolving policy geometry for scalable multiagent learning. In: Proceedings of the Ninth International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010), pp. 731–738 (2010)

Darwen, P., Yao, X.: Automatic modularization by speciation. In: Proceedings of the 1996 IEEE International Conference on Evolutionary Computation (ICEC 1996), pp. 88–93 (1996)

Dasgupta, D., McGregor, D.: Designing application-specific neural networks using the structured genetic algorithm. In: Proceedings of the International Conference on Combinations of Genetic Algorithms and Neural Networks, pp. 87–96 (1992)

Dawkins, R., Krebs, J.: Arms races between and within species. Proceedings of the Royal Society of London Series B, Biological Sciences 205(1161), 489–511 (1979)

de Jong, E.D.: The Incremental Pareto-coevolution Archive. In: Deb, K., et al. (eds.) GECCO 2004. LNCS, vol. 3102, pp. 525–536. Springer, Heidelberg (2004)

de Jong, E.: A monotonic archive for Pareto-coevolution. Evolutionary Computation 15(1), 61–93 (2007)

de Jong, K., Spears, W.: An analysis of the interacting roles of population size and crossover in genetic algorithms. In: Parallel Problem Solving from Nature, pp. 38–47 (1991)

de Jong, K., Spears, W., Gordon, D.: Using genetic algorithms for concept learning. Machine learning 13(2), 161–188 (1993)

Deb, K.: Multi-objective optimization using evolutionary algorithms. Wiley (2001)

Dorigo, M., Colombetti, M.: Robot shaping: An experiment in behavior engineering. The MIT Press (1998)

Downing, K.L.: Reinforced genetic programming. Genetic Programming and Evolvable Machines 2(3), 259–288 (2001)

Doya, K.: Reinforcement learning in continuous time and space. Neural Computation 12(1), 219–245 (2000)

Drugowitsch, J.: Design and analysis of learning classifier systems: A probabilistic approach. Springer, Heidelberg (2008)

Ficici, S., Pollack, J.: A game-theoretic approach to the simple coevolutionary algorithm. In: Parallel Problem Solving from Nature PPSN VI, pp. 467–476. Springer, Heidelberg (2000)

Ficici, S., Pollack, J.: Pareto optimality in coevolutionary learning. Advances in Artificial Life, 316–325 (2001)

Floreano, D., Mondada, F.: Evolution of homing navigation in a real mobile robot. IEEE Transactions on Systems, Man, and Cybernetics, Part B 26(3), 396–407 (2002)

Floreano, D., Urzelai, J.: Evolution of plastic control networks. Autonomous Robots 11(3), 311–317 (2001)

French, R., Messinger, A.: Genes, phenes and the Baldwin effect: Learning and evolution in a simulated population. Artificial Life 4, 277–282 (1994)

Gaskett, C., Wettergreen, D., Zelinsky, A.: Q-learning in continuous state and action spaces. Advanced Topics in Artificial Intelligence, 417–428 (1999)

Gauci, J., Stanley, K.O.: A case study on the critical role of geometric regularity in machine learning. In: Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008 (2008)

Gauci, J., Stanley, K.O.: Autonomous evolution of topographic regularities in artificial neural networks. Neural Computation 22(7), 1860–1898 (2010)

Gerard, P., Stolzmann, W., Sigaud, O.: YACS: a new learning classifier system using anticipation. Soft Computing-A Fusion of Foundations, Methodologies and Applications 6(3), 216–228 (2002)

Gerard, P., Meyer, J., Sigaud, O.: Combining latent learning with dynamic programming in the modular anticipatory classifier system. European Journal of Operational Research 160(3), 614–637 (2005)

Giraud-Carrier, C.: Unifying learning with evolution through Baldwinian evolution and Lamarckism: A case study. In: Proceedings of the Symposium on Computational Intelligence and Learning (CoIL 2000), pp. 36–41 (2000)

Goldberg, D.: Genetic Algorithms in Search. In: Optimization and Machine Learning, Addison-Wesley (1989)

Goldberg, D., Deb, K.: A comparative analysis of selection schemes used in genetic algorithms. Foundations of genetic algorithms 1, 69–93 (1991)

Goldberg, D., Richardson, J.: Genetic algorithms with sharing for multimodal function optimization. In: Proceedings of the Second International Conference on Genetic Algorithms and their Application, p. 49 (1987)

Gomez, F., Miikkulainen, R.: Solving non-Markovian control tasks with neuroevolution. In: Proceedings of the International Joint Conference on Artificial Intelligence, pp. 1356–1361 (1999)

Gomez, F., Miikkulainen, R.: Active guidance for a finless rocket using neuroevolution. In: GECCO 2003: Proceedings of the Genetic and Evolutionary Computation Conference (2003)

Gomez, F., Schmidhuber, J.: Co-evolving recurrent neurons learn deep memory POMDPs. In: GECCO 2005: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 491–498 (2005a)

Gomez, F.J., Schmidhuber, J.: Evolving Modular Fast-Weight Networks for Control. In: Duch, W., Kacprzyk, J., Oja, E., Zadrożny, S. (eds.) ICANN 2005. LNCS, vol. 3697, pp. 383–389. Springer, Heidelberg (2005b)

Gomez, F.J., Schmidhuber, J., Miikkulainen, R.: Efficient Non-Linear Control Through Neuroevolution. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 654–662. Springer, Heidelberg (2006)

Gomez, F., Schmidhuber, J., Miikkulainen, R.: Accelerated neural evolution through cooperatively coevolved synapses. Journal of Machine Learning Research 9, 937–965 (2008)

Gruau, F.: Automatic definition of modular neural networks. Adaptive Behavior 3(2), 151 (1994)

Gruau, F., Whitley, D.: Adding learning to the cellular development of neural networks: Evolution and the Baldwin effect. Evolutionary Computation 1, 213–233 (1993)

Hansen, N., Müller, S., Koumoutsakos, P.: Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES). Evolutionary Computation 11(1), 1–18 (2003)

van Hasselt, H., Wiering, M.: Reinforcement learning in continuous action spaces. In: IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning, ADPRL, pp. 272–279 (2007)

Haykin, S.: Neural networks: a comprehensive foundation. Prentice-Hall (1994)

Heidrich-Meisner, V., Igel, C.: Variable metric reinforcement learning methods applied to the noisy mountain car problem. Recent Advances in Reinforcement Learning, 136–150 (2008)

Heidrich-Meisner, V., Igel, C.: Hoeffding and Bernstein races for selecting policies in evolutionary direct policy search. In: Proceedings of the 26th Annual International Conference on Machine Learning, pp. 401–408 (2009a)

Heidrich-Meisner, V., Igel, C.: Neuroevolution strategies for episodic reinforcement learning. Journal of Algorithms 64(4), 152–168 (2009b)

Heidrich-Meisner, V., Igel, C.: Uncertainty handling CMA-ES for reinforcement learning. In: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, pp. 1211–1218 (2009c)

Hillis, W.: Co-evolving parasites improve simulated evolution as an optimization procedure. Physica D: Nonlinear Phenomena 42(1-3), 228–234 (1990)

Hinton, G.E., Nowlan, S.J.: How learning can guide evolution. Complex Systems 1, 495–502 (1987)

Holland, J., Reitman, J.: Cognitive systems based on adaptive algorithms. ACM SIGART Bulletin 63, 49–49 (1977)

Holland, J.H.: Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology. In: Control and Artificial Intelligence. University of Michigan Press (1975)

Hornby, G., Pollack, J.: Creating high-level components with a generative representation for body-brain evolution. Artificial Life 8(3), 223–246 (2002)

Igel, C.: Neuroevolution for reinforcement learning using evolution strategies. In: Congress on Evolutionary Computation, vol. 4, pp. 2588–2595 (2003)

Jansen, T., Wiegand, R.P.: The cooperative coevolutionary (1+1) EA. Evolutionary Computation 12(4), 405–434 (2004)

Kaelbling, L.P.: Learning in Embedded Systems. MIT Press (1993)

Kernbach, S., Meister, E., Scholz, O., Humza, R., Liedke, J., Ricotti, L., Jemai, J., Havlik, J., Liu, W.: Evolutionary robotics: The next-generation-platform for on-line and on-board artificial evolution. In: CEC 2009: IEEE Congress on Evolutionary Computation, pp. 1079–1086 (2009)

Kohl, N., Miikkulainen, R.: Evolving neural networks for fractured domains. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1405–1412 (2008)

Kohl, N., Miikkulainen, R.: Evolving neural networks for strategic decision-making problems. Neural Networks 22, 326–337 (2009); (special issue on Goal-Directed Neural Systems)

Koppejan, R., Whiteson, S.: Neuroevolutionary reinforcement learning for generalized helicopter control. In: GECCO 2009: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 145–152 (2009)

Kovacs, T.: Strength or accuracy: credit assignment in learning classifier systems. Springer, Heidelberg (2003)

Larranaga, P., Lozano, J.: Estimation of distribution algorithms: A new tool for evolutionary computation. Springer, Netherlands (2002)

Lindenmayer, A.: Mathematical models for cellular interactions in development II. Simple and branching filaments with two-sided inputs. Journal of Theoretical Biology 18(3), 300–315 (1968)

Littman, M.L., Dean, T.L., Kaelbling, L.P.: On the complexity of solving Markov decision processes. In: Proceedings of the Eleventh International Conference on Uncertainty in Artificial Intelligence, pp. 394–402 (1995)

Lucas, S.M., Runarsson, T.P.: Temporal difference learning versus co-evolution for acquiring othello position evaluation. In: IEEE Symposium on Computational Intelligence and Games (2006)

Lucas, S.M., Togelius, J.: Point-to-point car racing: an initial study of evolution versus temporal difference learning. In: Symposium, I.E.E.E. (ed.) on Computational Intelligence and Games, pp. 260–267 (2007)

Mahadevan, S., Maggioni, M.: Proto-value functions: A Laplacian framework for learning representation and control in Markov decision processes. Journal of Machine Learning Research 8, 2169–2231 (2007)

Mahfoud, S.: A comparison of parallel and sequential niching methods. In: Conference on Genetic Algorithms, vol. 136, p. 143 (1995)

McQuesten, P., Miikkulainen, R.: Culling and teaching in neuro-evolution. In: Proceedings of the Seventh International Conference on Genetic Algorithms, pp. 760–767 (1997)

Meyer, J., Husbands, P., Harvey, I.: Evolutionary robotics: A survey of applications and problems. In: Evolutionary Robotics, pp. 1–21. Springer, Heidelberg (1998)

Millán, J., Posenato, D., Dedieu, E.: Continuous-action Q-learning. Machine Learning 49(2), 247–265 (2002)

Monroy, G., Stanley, K., Miikkulainen, R.: Coevolution of neural networks using a layered Pareto archive. In: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, p. 336 (2006)

Moriarty, D., Miikkulainen, R.: Forming neural networks through efficient and adaptive coevolution. Evolutionary Computation 5(4), 373–399 (1997)

Moriarty, D.E., Miikkulainen, R.: Efficient reinforcement learning through symbiotic evolution. Machine Learning 22(11), 11–33 (1996)

Moriarty, D.E., Schultz, A.C., Grefenstette, J.J.: Evolutionary algorithms for reinforcement learning. Journal of Artificial Intelligence Research 11, 199–229 (1999)

Ng, A.Y., Coates, A., Diel, M., Ganapathi, V., Schulte, J., Tse, B., Berger, E., Liang, E.: Inverted autonomous helicopter flight via reinforcement learning. In: Proceedings of the International Symposium on Experimental Robotics (2004)

Nolfi, S., Parisi, D.: Learning to adapt to changing environments in evolving neural networks. Adaptive Behavior 5(1), 75–98 (1997)

Nolfi, S., Elman, J.L., Parisi, D.: Learning and evolution in neural networks. Adaptive Behavior 2, 5–28 (1994)

Nordin, P., Banzhaf, W.: An on-line method to evolve behavior and to control a miniature robot in real time with genetic programming. Adaptive Behavior 5(2), 107 (1997)

Panait, L., Luke, S.: Cooperative multi-agent learning: The state of the art. Autonomous Agents and Multi-Agent Systems 11(3), 387–434 (2005)

Panait, L., Luke, S., Harrison, J.F.: Archive-based cooperative coevolutionary algorithms. In: GECCO 2006: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, pp. 345–352 (2006)

Parr, R., Painter-Wakefield, C., Li, L., Littman, M.: Analyzing feature generation for value-function approximation. In: Proceedings of the 24th International Conference on Machine Learning, p. 744 (2007)

Pereira, F.B., Costa, E.: Understanding the role of learning in the evolution of busy beaver: A comparison between the Baldwin Effect and a Lamarckian strategy. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2001 (2001)

Peters, J., Schaal, S.: Natural actor-critic. Neurocomputing 71(7-9), 1180–1190 (2008)

Pollack, J., Blair, A.: Co-evolution in the successful learning of backgammon strategy. Machine Learning 32(3), 225–240 (1998)

Popovici, E., Bucci, A., Wiegand, P., De Jong, E.: Coevolutionary principles. In: Rozenberg, G., Baeck, T., Kok, J. (eds.) Handbook of Natural Computing. Springer, Berlin (2010)

Potter, M.A., De Jong, K.A.: Evolving neural networks with collaborative species. In: Summer Computer Simulation Conference, pp. 340–345 (1995)

Potter, M.A., De Jong, K.A.: Cooperative coevolution: An architecture for evolving coadapted subcomponents. Evolutionary Computation 8, 1–29 (2000)

Pratihar, D.: Evolutionary robotics: A review. Sadhana 28(6), 999–1009 (2003)

Priesterjahn, S., Weimer, A., Eberling, M.: Real-time imitation-based adaptation of gaming behaviour in modern computer games. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1431–1432 (2008)

Radcliffe, N.: Genetic set recombination and its application to neural network topology optimisation. Neural Computing & Applications 1(1), 67–90 (1993)

Rosin, C.D., Belew, R.K.: New methods for competitive coevolution. Evolutionary Computation 5(1), 1–29 (1997)

Rubinstein, R., Kroese, D.: The cross-entropy method: a unified approach to combinatorial optimization. In: Monte-Carlo Simulation, and Machine Learning. Springer, Heidelberg (2004)

Runarsson, T.P., Lucas, S.M.: Co-evolution versus self-play temporal difference learning for acquiring position evaluation in small-board go. IEEE Transactions on Evolutionary Computation 9, 628–640 (2005)

Schmidhuber, J., Wierstra, D., Gomez, F.J.: Evolino: Hybrid neuroevolution / optimal linear search for sequence learning. In: Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, pp. 853–858 (2005)

Schmidhuber, J., Wierstra, D., Gagliolo, M., Gomez, F.: Training recurrent networks by evolino. Neural Computation 19(3), 757–779 (2007)

Schroder, P., Green, B., Grum, N., Fleming, P.: On-line evolution of robust control systems: an industrial active magnetic bearing application. Control Engineering Practice 9(1), 37–49 (2001)

Sigaud, O., Butz, M., Kozlova, O., Meyer, C.: Anticipatory Learning Classifier Systems and Factored Reinforcement Learning. Anticipatory Behavior in Adaptive Learning Systems, 321–333 (2009)

Stanley, K., Miikkulainen, R.: A taxonomy for artificial embryogeny. Artificial Life 9(2), 93–130 (2003)

Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evolutionary Computation 10(2), 99–127 (2002)

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

Stanley, K.O., Miikkulainen, R.: Evolving a Roving Eye for Go. In: Deb, K., et al. (eds.) GECCO 2004. LNCS, vol. 3103, pp. 1226–1238. Springer, Heidelberg (2004b)

Stanley, K.O., Bryant, B.D., Miikkulainen, R.: Evolving adaptive neural networks with and without adaptive synapses. In: Proceeedings of the 2003 Congress on Evolutionary Computation (CEC 2003), vol. 4, pp. 2557–2564 (2003)

Stanley, K.O., D'Ambrosio, D.B., Gauci, J.: A hypercube-based indirect encoding for evolving large-scale neural networks. Artificial Life 15(2), 185–212 (2009)

Steels, L.: Emergent functionality in robotic agents through on-line evolution. In: Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems, pp. 8–16 (1994)

Sutton, R.S.: Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In: Proceedings of the Seventh International Conference on Machine Learning, pp. 216–224 (1990)

Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press (1998)

Sywerda, G.: Uniform crossover in genetic algorithms. In: Proceedings of the Third International Conference on Genetic Algorithms, pp. 2–9 (1989)

Tan, C., Ang, J., Tan, K., Tay, A.: Online adaptive controller for simulated car racing. In: Congress on Evolutionary Computation (CEC), pp. 2239–2245 (2008)

Taylor, M.E., Whiteson, S., Stone, P.: Comparing evolutionary and temporal difference methods in a reinforcement learning domain. In: GECCO 2006: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1321–1328 (2006)

Tesauro, G.: TD-gammon, a self-teaching backgammon program achieves master-level play. Neural Computation 6, 215–219 (1994)

Tesauro, G.: Comments on co-evolution in the successful learning of backgammon strategy. Machine Learning 32(3), 241–243 (1998)

Verbancsics, P., Stanley, K.: Evolving Static Representations for Task Transfer. Journal of Machine Learning Research 11, 1737–1769 (2010)

Von Neumann, J.: Zur Theorie der Gesellschaftsspiele Math. Annalen 100, 295–320 (1928)

Whiteson, S., Stone, P.: Evolutionary function approximation for reinforcement learning. Journal of Machine Learning Research 7, 877–917 (2006a)

Whiteson, S., Stone, P.: On-line evolutionary computation for reinforcement learning in stochastic domains. In: GECCO 2006: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1577–1584 (2006b)

Whiteson, S., Kohl, N., Miikkulainen, R., Stone, P.: Evolving keepaway soccer players through task decomposition. Machine Learning 59(1), 5–30 (2005)

Whiteson, S., Tanner, B., White, A.: The reinforcement learning competitions. AI Magazine 31(2), 81–94 (2010a)

Whiteson, S., Taylor, M.E., Stone, P.: Critical factors in the empirical performance of temporal difference and evolutionary methods for reinforcement learning. Autonomous Agents and Multi-Agent Systems 21(1), 1–27 (2010b)

Whitley, D., Dominic, S., Das, R., Anderson, C.W.: Genetic reinforcement learning for neurocontrol problems. Machine Learning 13, 259–284 (1993)

Whitley, D., Gordon, S., Mathias, K.: Lamarckian evolution, the Baldwin effect and function optimization. In: Parallel Problem Solving from Nature - PPSN III, pp. 6–15 (1994)

Wiegand, R., Liles, W., De Jong, K.: An empirical analysis of collaboration methods in cooperative coevolutionary algorithms. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), pp. 1235–1242 (2001)

Wieland, A.: Evolving neural network controllers for unstable systems. In: International Joint Conference on Neural Networks, vol 2, pp. 667–673 (1991)

Wilson, S.: Classifier fitness based on accuracy. Evolutionary Computation 3(2), 149–175 (1995)

Wilson, S.: Function approximation with a classifier system. In: GECCO 2001: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 974–982 (2001)

Wolpert, D., Tumer, K.: Optimal payoff functions for members of collectives. Modeling Complexity in Economic and Social Systems, 355 (2002)

Yamasaki, K., Sekiguchi, M.: Clear explanation of different adaptive behaviors between Darwinian population and Lamarckian population in changing environment. In: Proceedings of the Fifth International Symposium on Artificial Life and Robotics, vol. 1, pp. 120–123 (2000)

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

Yong, C.H., Miikkulainen, R.: Coevolution of role-based cooperation in multi-agent systems. Tech. Rep. AI07-338, Department of Computer Sciences, The University of Texas at Austin (2007)

Zhang, B., Muhlenbein, H.: Evolving optimal neural networks using genetic algorithms with Occam's razor. Complex Systems 7(3), 199–220 (1993)

Zufferey, J.-C., Floreano, D., van Leeuwen, M., Merenda, T.: Evolving vision-based flying robots. In: Bülthoff, H.H., Lee, S.-W., Poggio, T.A., Wallraven, C. (eds.) BMCV 2002. LNCS, vol. 2525, pp. 592–600. Springer, Heidelberg (2002)