Journal of Machine Learning Research 9 (2008) 937-965 Submitted 8/07; Revised 2/08; Published 5/08

Accelerated Neural Evolution through Cooperatively Coevolved Synapses

Faustino Gomez TINO@IDSIA.CH
Jürgen Schmidhuber[*] JUERGEN@IDSIA.CH
Dalle Molle Institute for Artificial Intelligence (IDSIA)
Galleria 2, Manno (Lugano), Switzerland

Risto Miikkulainen RISTO@CS.UTEXAS.EDU
Department of Computer Sciences
University of Texas, Austin, TX 78712 USA

Editor: Melanie Mitchell

Abstract

Many complex control problems require sophisticated solutions that are not amenable to traditional controller design. Not only is it difficult to model real world systems, but often it is unclear what kind of behavior is required to solve the task. Reinforcement learning (RL) approaches have made progress by using direct interaction with the task environment, but have so far not scaled well to large state spaces and environments that are not fully observable. In recent years, neuroevolution, the artificial evolution of neural networks, has had remarkable success in tasks that exhibit these two properties. In this paper, we compare a neuroevolution method called Cooperative Synapse Neuroevolution (CoSyNE), that uses cooperative coevolution at the level of individual synaptic weights, to a broad range of reinforcement learning algorithms on very difficult versions of the pole balancing problem that involve large (continuous) state spaces and hidden state. CoSyNE is shown to be significantly more efficient and powerful than the other methods on these tasks.

Keywords: coevolution, recurrent neural networks, non-linear control, genetic algorithms, experimental comparison

1. Introduction

In many decision making processes such as manufacturing, aircraft control, and robotics researchers are faced with the problem of controlling systems that are highly complex and unstable. A controller or *agent* must be built that observes the state of the system, or *environment*, and outputs a control signal that affects future states of the environment in some desirable way.

The problem with designing or programming such controllers by direct engineering methods is twofold: (1) The environment is often non-linear and noisy so that it is impossible to obtain the kind of accurate and tractable mathematical model required by these methods. (2) The task is complex enough that there is very little *a priori* knowledge of what constitutes a reasonable, much less optimal, control strategy.

These two problems have compelled researchers to explore methods based on Dynamic Programming, for example Reinforcement Learning (RL; Sutton and Barto, 1998). Instead of trying to pre-program a response to every likely situation, an agent *learns* the utility of being in each state

∗. Also at Technische Universität München Boltzmannstr. 3, 85748 Garching, München, Germany.

(i.e., a value-function) from a reward signal it receives while interacting directly with the environment. In principle, RL methods can solve these problems: they do not require a mathematical model (i.e., the state transition probabilities) of the environment and can solve many problems where examples of correct behavior are not available. However, in practice, they have not scaled well to large state spaces or tasks where the state of the environment is not fully observable to the agent. This is a serious problem because the real world is continuous (i.e., there are an infinite number of states) and artificial agents, like natural organisms, are necessarily constrained in their ability to fully perceive their environment.

More recently, methods for evolving artificial neural networks or *neuroevolution* have shown promising results on continuous, partially observable tasks (Gomez, 2003; Nolfi and Parisi, 1995; Yamauchi and Beer, 1994). Our previous method, Enforced SubPopulations, is a particularly effective neuroevolution algorithm that has been a applied successfully to many domains (Perez-Bergquist, 2001; Lubberts and Miikkulainen, 2001; Greer et al., 2002; Whiteson et al., 2003; Bryant and Miikkulainen, 2003; Gomez et al., 2001; Grasemann and Miikkulainen, 2005), including the real world reinforcement learning task of finless rocket control (Gomez and Miikkulainen, 2003). The goal of this paper is to present a new algorithm that builds on ESP called Cooperative Synapse Neuroevolution (CoSyNE), and compare it to a wide range of other learning systems in a setting that is both challenging and practical. To this end, we have chosen a set of pole balancing tasks ranging from the trivial to versions that are extremely difficult for some of todays most advanced methods.

The paper is organized as follows: in Section 2, we discuss the general neuroevolution paradigm. In Section 3, the underlying approach used by CoSyNE, cooperative coevolution is described. In Section 4, the CoSyNE algorithm is presented. Section 5 presents our experiments comparing CoSyNE with value function, policy search, and other evolutionary methods. Sections 6 and 7 provide some discussion of our overall results, and conclusions.

2. Neuroevolution

The basic idea of Neuroevolution (NE; Yao, 1999) is to search the space of neural network policies directly using a genetic algorithm. In contrast to *ontogenetic* learning involving a single agent that learns incrementally (i.e., value-based RL), NE uses a population of solutions. The individual solutions are not modified during evaluation; instead, adaptation arises through repeatedly recombining the population's most fit individuals in a kind of collective or *phylogenetic* learning. The population gradually improves as a whole until a sufficiently fit individual is found.

In NE, neural network specifications are encoded in string representations or *chromosomes* (see Figure 1). A chromosome can encode any relevant network parameter including synaptic weight values, number of processing units, connectivity (topology), learning rate, etc. These network *genotypes* are then evolved in a sequence of generations. Each generation each genotype is mapped to its network phenotype (i.e., the actual network), and then evaluated in the problem environment and awarded a fitness score that quantifies its performance in some desirable way. After this evaluation phase, genotypes are selected from the population according to fitness through a variety of possible schemes (e.g., fitness proportional, linear ranking, tournament selection, etc.), and then mated through crossover and possibly mutated to form new genotypes that usually replace the least fit members of the population. This cycle repeats until a sufficiently fit network is found, or some other stopping criteria is met.

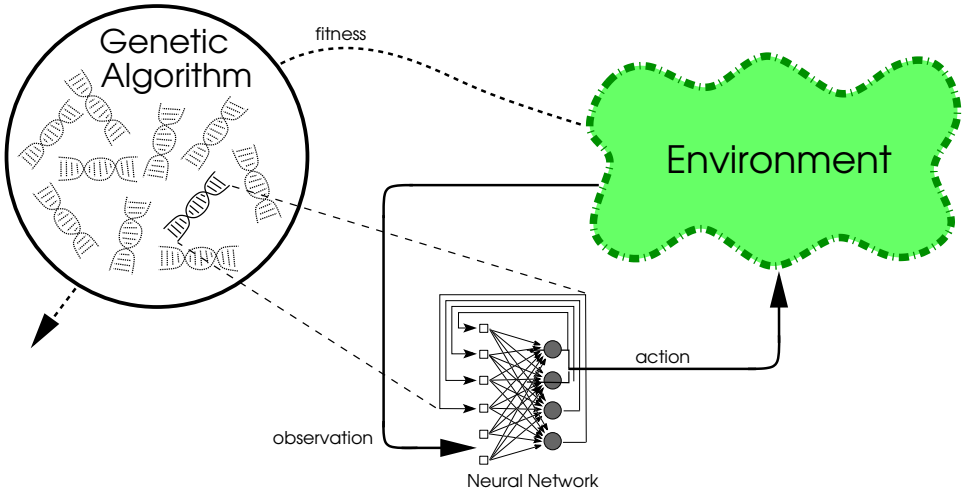

Figure 1: **Neuroevolution.** Each chromosome is transformed into a neural network phenotype and evaluated on the task. The agent receives input from the environment (observation) and propagates it through its neural network to compute an output signal (action) that affects the environment. At the end of the evaluation, the network is assigned a fitness according to its performance. The networks that perform well on the task are mated to generate new networks.

NE approaches differ primarily in how they encode neural network specifications into genetic strings. Direct encoding schemes represent the parameters explicitly on the chromosome as binary or real numbers that are mapped directly to the phenotype (Belew et al., 1991; Jefferson et al., 1991; Moriarty, 1997; Gomez, 2003; Stanley and Miikkulainen, 2002). Indirect encodings operate at a higher level of abstraction. Some simply provide a coarse description such as delineating a neuron's receptive field (Mandischer, 1993) or connective density (Harp et al., 1989), while others are more algorithmic, providing growth rules in the form of graph generating grammars (Kitano, 1990; Voigt et al., 1993; Gruau et al., 1996b). These schemes have the advantage that very large networks can be represented without requiring large chromosomes. Our CoSyNE method is a direct encoding method that does not evolve topology.

By searching the space of policies directly, NE can be applied to reinforcement learning problems without using a value function—neural network controllers map observations from the environment directly to actions. This mapping is potentially powerful: neural networks are universal function approximators that can generalize and tolerate noise. Networks with feedback connections (i.e., recurrent networks) can maintain internal state extracted from a history of inputs, allowing them to solve partially observable tasks. By evolving these networks instead of training them, NE avoids the problem of vanishing error gradients that affect recurrent network learning algorithms (Hochreiter et al., 2001). For NE to work, the environment need not satisfy any particular constraints—it can be continuous and partially observable. All that concerns a NE system is that the network representations be large enough to solve the task and that there is an effective way to evaluate the relative quality of candidate solutions.

Algorithm 1: Cooperative Coevolution (n, m)

```
1  Initialize {P_1, ..., P_n}
2  repeat
3      repeat
4          for j = 1 to n do     // construct complete solution
5              x_ij = Select(P_j)
6              x ⇐ x_ij          // add subgenotype to complete solution
7          end
8          Evaluate(x)
9      until enough solutions evaluated
10     for i = 1 to n do         // each subpopulation reproduces independently
11         Recombine(P_i)
12     end
13 until solution is found
```

3. Cooperative Coevolution

In natural ecosystems, organisms of one species compete and/or cooperate with many other different species in their struggle for resources and survival. The fitness of each individual changes over time because it is coupled to that of other individuals inhabiting the environment. As species evolve they specialize and co-adapt their survival strategies to those of other species. This phenomenon of *coevolution* has been used to encourage complex behaviors in GAs.

Most coevolutionary problem solving systems have concentrated on competition between species (Darwen, 1996; Pollack et al., 1996; Paredis, 1994; Miller and Cliff, 1994; Rosin, 1997). These methods rely on establishing an "arms race" where each species produces stronger and stronger strategies for the others to defeat. This is a natural approach for problems such as game-playing where often an optimal opponent is not available.

A very different kind of coevolutionary model emphasizes cooperation. Cooperative coevolution is motivated, in part, by the recognition that the complexity of difficult problems can be reduced through modularization (e.g., the human brain; Grady, 1993). In cooperative coevolutionary algorithms the species represent solution components. Each individual forms a part of a complete solution but need not represent anything meaningful on its own. The components are evolved by measuring their contribution to complete solutions and recombining those that are most beneficial to solving the task.

Algorithm 1 outlines the basic operation of a generic cooperative coevolutionary algorithm. The first parameter n specifies the number of species (components) that will be coevolved. Each species has its own subpopulation $P_i, i = 1..n$, containing m *subgenotypes*, $x_{ij} \in P_i, j = 1..m$ which are initialized with random values (line 1). Assuming complete solutions of fixed size, n determines the granularity at which the coevolutionary search is conducted.

Next, some number of complete solutions are constructed and evaluated (lines 3-9). A complete solution $\mathbf{x}$ is formed by combining one subgenotype, selected according to some policy, from each of the subpopulations. Usually, the string representations of each subgenotype are simply concatenated in a predefined order to form a single chromosome. Each $\mathbf{x}$ is evaluated in the problem environment and a fitness score is assigned to each constituent subgenotype. Since the number of evaluations per

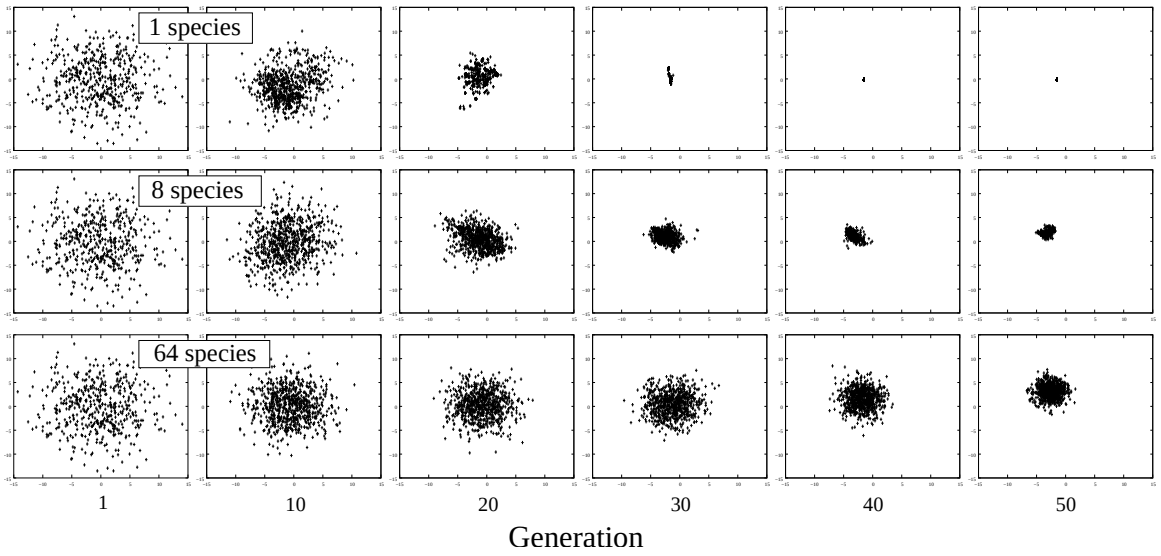

Figure 2: **Convergence speed for varying numbers of species.** Each row shows a PCA projection of 128-dimensional chromosomes at different generations during an evolutionary run optimizing a very simple multi-modal test function. All three runs start with the same set of complete solution (see first column). In row 1, the solutions are not coevolved because each genotype is a complete solution. In row 2, 8 species are coevolved (i.e., have to be combined to form a the complete solutions shown in the plots), and in row 3, 64 species are coevolved. The more species there are to cooperate, the longer it takes for the evolution to converge.

generation can exceed m, subgenotypes can participate in more that one evaluation per generation. Therefore, the fitness score of each x_{ij} at the end of a generation is some function of the raw fitness scores accumulated over multiple evaluations, and is considered a *subjective measure* because it is coupled with that of its *collaborators*, in contrast to an objective measure that only depends on the individual itself (Wiegand, 2003). The exact number of evaluations per subgenotype depends on the collaboration scheme employed by a particular algorithm. One common approach, for example, is simply to evaluate each subgenotype in n trials, and then take the average or best fitness.

Once enough evaluations have been performed, each subpopulation is recombined to form new subgenotypes, as in a normal GA.

Early work in this area was done by Holland and Reitman (1978) in Classifier Systems. A population of rules was evolved by assigning a fitness to each rule based on how well it interacted with other rules. This approach has been used in learning neural network classifiers, in coevolution of cascade correlation networks, and in coevolution of radial basis functions (Eriksson and Olsson, 1997; Horn et al., 1994; Paredis, 1995; Whitehead and Choate, 1995). More recently, Husbands and Mill (1991) and Potter and De Jong (1995) developed a method called Cooperative Coevolutionary GA (CCGA) in which each of the species is evolved independently in its own population. As in Classifier Systems, individuals in CCGA are rewarded for making favorable contributions to complete solutions, but members of different populations (species) are not allowed to mate. A

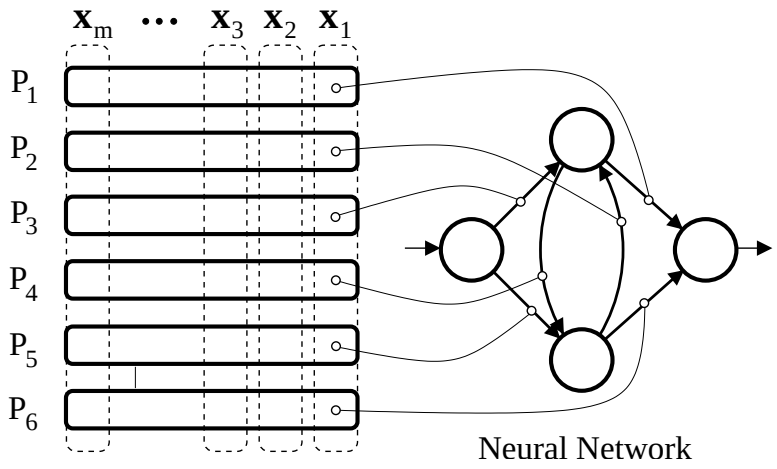

Figure 3: **The CoSyNE method for neuroevolution.** On the left, the figure shows and example population consisting of six subpopulations, $P_1..P_6$, each containing m weight values. To create a network, first the weights at a given index in each subpopulation are collected into a chromosome $\mathbf{x}$, then the weights are mapped to their corresponding synapses in a predefined network architecture with six connections, shown at right.

particularly powerful idea is to combine cooperative coevolution with neuroevolution so that the benefits of evolving neural networks can be enhanced further through improved search efficiency.

Much of the motivation for using the cooperative coevolutionary approach is based on the intuition that many problems may be decomposable into weakly coupled low-dimensional subspaces that can be searched semi-independently by separate species (Wiegand et al., 2001; Jansen and Wiegand, 2003, 2004; Panait et al., 2006). Our experience shows that there may be another, complementary, explanation as to why cooperative coevolution in many cases outperforms single-population algorithms. Figure 2 compares the convergence behavior of the same initial population of complete solutions using different number of species: 1, 8, and 64. Each point represents a 128-dimensional chromosome projected onto 2-D using Principal Component Analysis. The chromosomes are coevolved to optimize a continuous multi-modal test function[1] with 128 randomly distributed maxima that represent valid solutions. As the number of species increases, the selection of subgenotypes for reproduction becomes less greedy, causing the search points that are evaluated each generation to converge more slowly, providing more paths toward better solutions (not shown). In a normal evolutionary algorithm, a subgenotype suffers the fate of the complete solution to which it is attached. If the complete solution performs with high fitness, the subgenotype is retained in the population, even if it is not ultimately beneficial to the search; if it is less fit then this potentially useful component (if combined with other subgenotypes in the population) is lost. This diversity sustaining mechanism is exploited fully in the CoSyNE algorithm, introduced next.

1. The URL is http://www.cs.uwyo.edu/˜wspears/multi.kennedy.html.

Algorithm 2: CoSyNE (n, m, Ψ)

```
1  Initialize P = {P_1, ..., P_n}
2  repeat
3      for j = 1 to m do
4          x_j ⇐ (x_1j, ..., x_nj)     // form complete solution
5          Evaluate(x_j, Ψ)
6      end
7      O ⇐ Recombine(P)
8      for i = 1 to n do
9          Sort(P_i)
10         for k = 1 to l do     // replace least fit weights with
11             x_i,m−k ⇐ o_ik     // weights from offspring nets
12         end
13         for j = 1 to m do
14             prob(x_ij) ⇐ F(P, i, j)     // assign probability to each weight
15             if rand() < prob(x_ij) then
16                 mark(x_ij)     // mark weight for permutation probabilistically
17             end
18         end
19         PermuteMarked(P_i)     // see Figure 4
20     end
21 until solution is found
```

4. Cooperative Synapse Neuroevolution (CoSyNE)

Previous Cooperative Coevolution NE methods decomposed networks at the neuron level (Moriarty, 1997; Potter and De Jong, 1995; Gomez, 2003). This is a natural approach dictated by phenotypic structure: networks consist of multiple processing units that function in parallel. In contrast, CoSyNE evolves at the lowest possible level of granularity, the level of the individual synaptic weight. For each network connection, there is a separate subpopulation consisting of real valued weights. Like neuron-level methods such as ESP, networks are constructed by selecting one member from each subpopulation and plugging them into a predefined network topology.

Algorithm 2 describes the CoSyNE algorithm in pseudocode. First (line 1), a population $\mathcal{P}$ consisting of n subpopulations $P_i, i = 1..n$, is created, where n is the number of synaptic weights in the networks to be evolved, determined by a user-specified network architecture Ψ. Each subpopulation is initialized to contain m real numbers, $x_{ij} = \mathcal{P}_{ij} \in P_i, j = 1..m$, chosen from a uniform probability distribution in the interval $[-\alpha, \alpha]$. The population is thereby represented by an $n \times m$ matrix.

CoSyNE then loops through a sequence of *generations* until a sufficiently good network is found (lines 2-21). Each generation starts by constructing a complete network chromosome $\mathbf{x}_j = (x_{1j}, x_{2j}, \ldots, x_{nj})$ from each row in $\mathcal{P}$. The m resulting chromosomes are transformed into networks by assigning their weights to their corresponding synapses, in Ψ (line 4; see Figure 3).

After all of the networks have been evaluated (line 5) and assigned a fitness, the top quarter with the highest fitness (i.e., the parents) are recombined (line 7) using crossover and mutation. Recombination produces a pool of offspring O consisting of l new network chromosomes $\mathbf{o}_k$, where

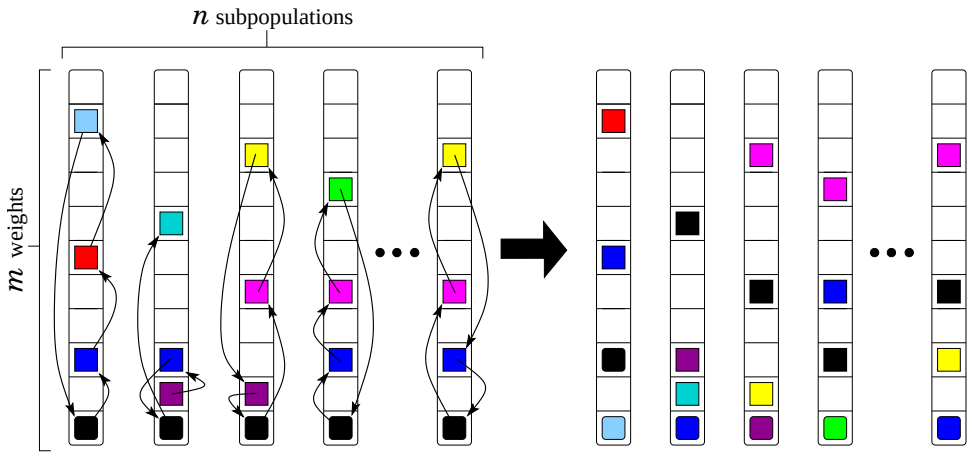

Figure 4: **Probabilistic permutations.** On the left is the set of subpopulations before permutation. The colored boxes are denote those genotypes that have been marked for permutation based on Equation 1. As the individuals are sorted by fitness within each subpopulation, notice that the less fit individuals have a higher probability of being permuted. On the right, the marked individuals have been permuted among themselves with each subpopulation. All unmarked genotypes remain part of the same complete solution.

$o_{ik} = O_{ik} \in O_i, i = 1..n, k = 1..l$, and O_i is the offspring subpopulation corresponding to P_i. The subpopulations are then sorted by fitness (line 9), and the weights from the new networks are added to $\mathcal{P}$ by replacing the least fit weights in their corresponding subpopulation (i.e., the P with the same index i; lines 10-11).

At this point the algorithm functions as a conventional neuroevolution system that evolves complete network chromosomes. In order to *coevolve* the synaptic weights, the subpopulations are permuted so that each weight forms part of a potentially different network in the next generation. Permutation is performed probabilistically. First, weights are marked randomly according to probabilities assigned by a user-defined function $F()$ (lines 14-17). Then the marked weights are permuted amongst themselves (see Figure 4). The function $F()$ can be anything from as simple as $\text{prob}(x_{ij}) = 1.0, \forall i, j$, in which case all weights are permuted, or more sophisticated:

$$\text{prob}(x_{ij}) = 1 - \sqrt[n]{\frac{f(x_{ij}) - f_i^{min}}{f_i^{max} - f_i^{min}}} \tag{1}$$

where $f(x_{ij})$ is the fitness of subgenotype (weight) x_{ij}, and f_j^{min} and f_j^{max} are, respectively, the fitness of the least and most fit individuals in subpopulation i. In this case, the probability of disrupting the network $\mathbf{x}_j$ is inversely proportional to its relative fitness, so that weight combinations that receive high fitness are more likely to be preserved, while those with low fitness are more likely to be disrupted and their constituents used to search for new complete solutions. In the experiments below, the simpler function that permutes all weights, except for the newly inserted offspring weights, was found to work well.

The basic CoSyNE framework does not specify how the weights are grouped in the complete solution chromosomes (i.e., which entry in the chromosome corresponds to which synapse) or which

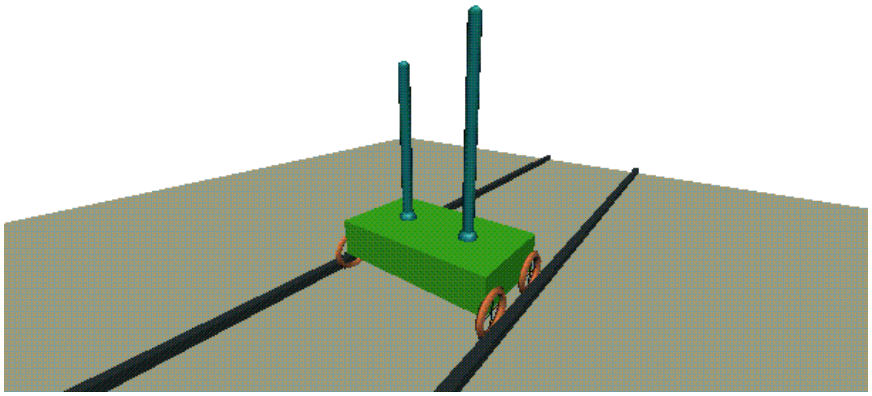

Figure 5: **The double pole balancing system.** Both poles must be balanced simultaneously by applying a continuous force to the cart. The system becomes more difficult to control as the poles assume similar lengths and if the velocities are not provided to the controller. The figure is a snapshot of a 3D real-time simulation.

genetic operators are used. In the implementation used in this paper, the weights of each neuron are grouped together (i.e., form a substring) and are separated into adjacent input, output, and recurrent weight segments, and the neuron substrings are concatenated together in a fixed order. For the genetic operators, we use multi-point crossover where 1-point crossover is applied to each neuron segment of the chromosome to generate two offspring, and mutation where each weight in $\mathcal{P}$ has a probability of being perturbed by Cauchy distributed noise with zero mean $\alpha = 0.3$.

5. Experiments

We compared CoSyNE to a broad range of learning algorithms on a sequence of increasingly difficult versions of the pole balancing task. This scheme allows us to compare methods at different levels of task complexity, exposing the strengths and limitations of each method with respect to specific challenges introduced by each succeeding task.

5.1 The Pole Balancing Problem

The basic pole balancing or inverted pendulum system consists of a pole hinged to a wheeled cart on a finite stretch of track. The objective is to apply a force to the cart at regular intervals such that the pole is balanced indefinitely and the cart stays within the track boundaries. This task has been a popular artificial learning testbed for over 30 years (Michie and Chambers, 1968; Anderson, 1989; Jang, 1992; Lin and Mitchell, 1992; Whitley et al., 1993) because it requires solving the temporal credit assignment problem, and is a good surrogate for a more general class of unstable control problems such as bipedal robot walking, and rocket guidance.

This long history notwithstanding, it turns out that the basic pole balancing problem can be solved easily by random search. To make it challenging for artificial learners, a variety of extensions to the basic pole-balancing task have been suggested. (Wieland, 1991) presented several variations that can be grouped into two categories: (1) modifications to the mechanical system itself, such as

adding a second pole either next to or on top of the other, and (2) restricting the amount of state information that is given to the controller; for example, only providing the cart position and the pole angle. The first category makes the task more difficult by introducing non-linear interactions between the poles. The second makes the task non-Markov, requiring the controller to employ short term memory to disambiguate underlying process states. Together, these extensions represent a family of tasks that can effectively test algorithms designed to learn control policies.

The sequence of comparisons presented below begins with a single pole version and then moves on to progressively more challenging variations culminating in a version where two separate poles of different length must be balanced simultaneously without the benefit of velocity information (see Appendix A for the equations of motion).

5.2 Other Methods

CoSyNE was compared to eight *ontogenetic* methods and seven *phylogenetic* methods in the pole balancing domain:

5.2.1 ONTOGENETIC METHODS

Random Weight Guessing (RWG) where the network weights are chosen at random (i.i.d.) from a uniform distribution. This approach is used to give an idea of how difficult each task is to solve by simply guessing a good set of weights.

Policy Gradient RL (PGRL; Sutton et al., 2000) where sampled Q-values are used to differentiate the performance of a given policy with respect to its parameters. The policy was implemented using a feed-forward neural network with one hidden layer.

Recurrent Policy Gradients (RPG; Wierstra et al., 2007) where a stochastic policy is represented by a Long Short-Term Memory network (LSTM; Hochreiter and Schmidhuber, 1997) trained with BackPropagation Through Time (Werbos, 1990). The gradient of the expected future reward over all possible state trajectories with respect to the policy parameters is calculated by Monte Carlo approximation. To reduce variance in the approximation, a *baseline* representing the expected average reward is used.

Value and Policy Search (VAPS; Meuleau et al., 1999) extends the work of Baird and Moore (1999) to policies that can make use of memory. The algorithm uses stochastic gradient descent to search the space of finite policy graph parameters. A policy graph is a state automaton that consists of nodes labeled with actions that are connected by arcs labeled with observations. When the system is in a particular node, the action associated with that node is taken and the underlying Markov environment transitions to the next observation that determines which arc is followed to the next action node.

Q-learning with MLP (Q-MLP): This method is the basic Q-learning algorithm (Watkins and Dayan, 1992) that uses a Multi-Layer Perceptron (i.e., a feed-forward artificial neural network) to map state-action pairs to values $Q(s,a)$. The input layer of the network has one unit per state variable and one unit per action variable. The output layer consists of a single unit indicating the Q-value. Values are learned through gradient descent on the prediction error using the backpropagation algorithm. This kind of approach has been studied widely with

success in tasks such as pole-balancing (Lin and Mitchell, 1992), pursuit-evasion games (Lin, 1992), and backgammon (Tesauro, 1992).

Sarsa(λ) with Case-Based function approximator (SARSA-CABA; Santamaria et al., 1998): This method consists of the Sarsa on-policy Temporal Difference control algorithm with eligibility traces that uses a case-based memory to approximate the *Q*-function. The memory explicitly records state-action pairs (i.e., cases) that have been experienced by the controller. The value of a new state-action pair not in the memory is calculated by combining the values of the *k*-nearest neighbors. A new case is added to the memory whenever the current query point is further than a specified *density threshold*, t_d away from all cases already in the memory. The case-based memory provides a locally-linear model of the Q-function that concentrates its resources on the regions of the state space that are most relevant to the task and expands its coverage dynamically according to t_d.

Sarsa(λ) with CMAC function approximator (SARSA-CMAC; Santamaria et al., 1998): This is the same as SARSA-CABA except that it uses a Cerebellar Model Articulation Controller (CMAC; Albus, 1975; Sutton, 1996) instead of a case-based memory to represent the Q-function. The CMAC partitions the state-action space with a set of overlapping tilings. Each tiling divides the space into a set of discrete *features* which maintain a value. When a query is made for a particular state-action pair, its *Q*-value is returned as the sum of the value in each tiling corresponding to the feature containing the query point. SARSA-CABA and SARSA-CMAC have both been applied to the pendulum swing-up task and the double-integrator task.

Adaptive Heuristic Critic (AHC; Anderson, 1987): uses a learning agent composed of two components: an *actor* (policy) and a *critic* (value-function), both of which are implemented using a feed-forward neural network trained with a variant of backpropagation.

The three value-function based methods (SARSA-CABA, SARSA-CMAC, and Q-MLP) each use a different kind of function approximator to represent a *Q*-function that can generalize across the continuous space of state-action pairs. Although these approximators can compute a value for any state-action pair, they do not implement true continuous control since the policy is not explicitly stored. Instead, continuous control is approximated by discretizing the action space at a resolution that is adequate for the problem. In order to select the optimal action *a* for a given state *s*, a *one-step* search in the action space is performed. The control agent selects actions according to an ε-*greedy* policy: with probability $1-\varepsilon$, $0 \leq \varepsilon < 1$, the action with the highest value is selected, and with probability ε, the action is random. This policy allows some exploration so that information can be gathered for all actions. In all simulations the controller was tested every 20 trials with ε=0 and learning turned off to determine whether a solution had been found.

5.2.2 PHYLOGENETIC METHODS

Symbiotic, Adaptive Neuro-Evolution (SANE; Moriarty, 1997) is a cooperative coevolutionary method that evolves two different populations simultaneously: a population of neurons and a population of *network blueprints* that specify how the neurons are combined to form complete networks. Each generation of networks is formed both using the blueprints and at random. Neurons that combine to form good networks receive high fitness, and are recombined in a

single population. Blueprints that result in favorable neuron combinations are also recombined to search for even better combinations.

Conventional Neuroevolution (CNE) is our implementation of single-population Neuroevolution similar to the algorithm used in Wieland (1991). In this approach, each chromosome in the population represents a complete neural network. CNE differs from Wieland's algorithm in that (1) the network weights are encoded with real instead of binary numbers, (2) it uses rank selection, and (3) it uses burst mutation. CNE is like ESP except that it evolves at the network level instead of the neuron level, and therefore provides a way to isolate the performance advantage of cooperative coevolution (ESP) over a single population approach (CNE).

Evolutionary Programming (EP; Saravanan and Fogel, 1995) is a general mutation-based evolutionary method that can be used to search the space of neural networks. Individuals are represented by two n-dimensional vectors (where n is the number of weights in the network): $\vec{x}$ contains the synaptic weight values for the network, and $\vec{\delta}$ is a vector of standard deviation values of $\vec{x}$. A network is constructed using the weights in $\vec{x}$, and offspring are produced by applying Gaussian noise to each element $\vec{x}(i)$ with standard deviation $\vec{\delta}(i), i \in \{1..n\}$.

Cellular Encoding (CE; Gruau et al., 1996a,b) uses Genetic Programming (GP; Koza, 1991) to evolve graph-rewriting programs. The programs control how neural networks are constructed out of "cells." A cell represents a neural network processing unit (neuron) with its input and output connections and a set of registers that contain synaptic weight values. A network is built through a sequence of operations that either copy cells or modify the contents of their registers. CE uses the standard GP crossover and mutation to recombine the programs allowing evolution to automatically determine an appropriate architecture for the task and relieve the investigator from this often trial-and-error undertaking.

Covariance Matrix Adaptation Evolutionary Strategies (CMA-ES; Hansen and Ostermeier 2001) evolves the covariance matrix of the mutation operator in evolutionary strategies. The results in the pole-balancing domain were obtained from Igel (2003).

NeuroEvolution of Augmenting Topologies (NEAT; Stanley and Miikkulainen, 2002; Stanley 2004) is another NE method that evolves topology as well as synaptic weights, but unlike CE it uses a direct encoding. NEAT starts with a population of minimal networks (i.e., no hidden units) that can increase in complexity by adding either new connections or units through mutation. Every time a new gene appears, a *global innovation number* is incremented and assigned to that gene. Innovation numbers allow NEAT to keep track of the historical origin of every gene in the population so that (1) crossover can be performed between networks with different topologies, and (2) the networks can be grouped into "species" based on topological similarity.

Whenever two networks are recombined, the genes in both chromosomes with the same innovation numbers are lined up. Those 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, and are inherited from the more fit parent.

The number of disjoint and excess genes is used to measure the distance between genomes. Using this distance, the population is divided into species so that individuals compete primarily within their own species instead of with the population at large. This way, topological

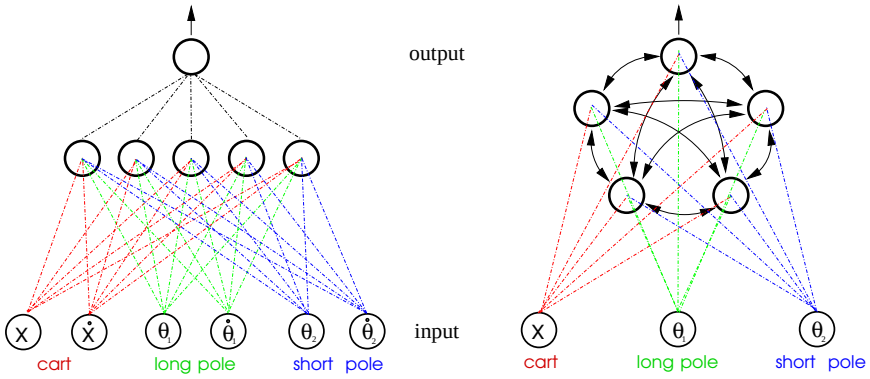

Figure 6: **Neural network control of the pole balancing system.** At each time step the network receives the current state of the cart-pole system $(x, \dot{x}, \theta_1, \dot{\theta_1}, \theta_2, \dot{\theta_2})$ through its input layer. For the feed-forward networks (a) used in the Markov tasks (1a and 2a), the input layer activation is propagated forward through the hidden layer of neurons to the output unit which indicates force to be applied to the cart. For the recurrent networks (b) used in the non-Markov tasks (1b and 2b), the neurons do not receive the velocities $(\dot{x}, \dot{\theta_1}, \dot{\theta_2})$, instead they must use their feedback connections to determine which direction the poles are moving. For the single pole version the network only has inputs for the cart and long pole.

innovations are protected and have time to optimize their structure before they have to compete with other species in the population.

Enforced SubPopulations (ESP; Gomez and Miikkulainen, 1997) is similar to SANE in that it uses cooperative coevolution at the neuron level, but, instead of using blueprints, the neuron population is split into disjoint subpopulations, one for each hidden unit in the network architecture being evolved. Instead of selecting neurons from a single population, as in SANE, to form networks, networks consist of one neuron from each subpopulation. During reproduction, neuron genotypes are only mated with members of their own subpopulation, and offspring remain in their parents' subpopulation.

For Q-MLP, SANE, CNE, ESP, and CoSyNE, experiments were run using our own code. For PGRL, AHC, SARSA, publicly available code from Grudic (2000), Anderson (1987), and Santamaria et al. (1998), was used respectively, modified for the pole-balancing domain. The parameter settings for each of these methods are listed in Appendix B. For VAPS, EP, CMA-ES, NEAT, and CE, the results were taken from the papers cited above. Data was not available for all methods on all tasks: however, in all such cases the method is shown to be significantly weaker already in a previous, easier task.

5.3 Task Setup

The pole balancing environment was implemented using a realistic physical model with friction, and fourth-order Runge-Kutta integration with a step size of 0.01s (see Appendix A for the equations of motion and parameters used). The state variables for the system are the following:

x : position of the cart.
$\dot{x}$: velocity of the cart.
θ_i : angle of the *i*-th pole $(i = 1, 2)$.
$\dot{\theta}_i$: angular velocity of the *i*-th pole.

Figure 6 shows how the network controllers interact with the pole balancing environment. At each time-step (0.02 seconds of simulated time) the network receives the state variable values scaled to [-1.0, 1.0]. This input activation is propagated through the network to produce a signal from the output unit that represents the amount of force used to push the cart. The force is then applied and the system transitions to the next state which becomes the new input to the controller. This cycle is repeated until a pole falls or the cart goes off the end of the track. In keeping with the setup in prior work (e.g., Wieland, 1991; Gruau et al., 1996b) we restrict the force to be no less than $\pm 1/256 \times 10$ Newtons so that the controllers cannot maintain the system in unstable equilibrium by outputting a force of zero when the poles are vertical.

The following four task configurations of increasing difficulty were used:

1. One Pole
   (a) Complete state information
   (b) Incomplete state information
2. Two Poles
   (a) Complete state information
   (b) Incomplete state information

Task 1a is the classic one-pole configuration. In 1b, the controller only has access to two of the four state variables: it does not receive the velocities $(\dot{x}, \dot{\theta})$. In 2a, the system now has a second pole next to the first, making the state-space 6-dimensional. Task 2b, like 1b, is non-Markov with the controller only seeing x, θ_1, and θ_2. Fitness was determined by the number of time steps a network could keep both poles within a specified failure angle from vertical and the cart between the ends of the track. The failure angle was 12° and 36° for the one and two pole tasks, respectively. For the one-pole tasks, the initial pole angle was set to 4.0° from vertical. For the two-pole tasks, the initial angle of the long pole was 4.0°, and the short pole was vertical. A task was considered solved if a network could do this for 100,000 time steps, which is equal to over 30 minutes in simulated time. CoSyNE evolved networks with one hidden unit, 20 weights per subpopulation for the 1-pole tasks, and 30 weights for the 2-pole tasks. Mutation was set to 0.3 for all experiments, which means that each weight in a new network have a 30% chance of being perturbed with Cauchy distributed noise. The initial weight range was $[-10, 10]$. All simulations were run on a 1.5GHz Intel Xeon.

5.4 Results: Balancing One Pole

Balancing one pole is a relatively easy problem that gives us a performance baseline before moving on to the much harder two-pole task. It has also been solved with many other methods and therefore serves to put the results in perspective with prior literature.

5.4.1 COMPLETE STATE INFORMATION

Table 1 shows the results for the single pole balancing task with complete state information. The results show that simply choosing weights at random (RWG) is sufficient to solve this task efficiently. CoSyNE was the only method that solved the task in fewer evaluations.

Method	Evaluations	CPU time (sec)
AHC	189,500	95
PGRL	28,779	1,163
Q-MLP	2,056	53
SARSA-CMAC	540	487
SARSA-CABA	965	1,713
RPG	(863)	—
CMA-ES	283	—
CNE	352	5
SANE	302	5
NEAT	743	7
ESP	289	4
RWG	199	2
CoSyNE	98	1

Table 1: **One pole with complete state information.** Comparison of various learning methods on the basic pole balancing problem with continuous control. Results for all methods are averages of 50 runs.

With the exception of RWG, there is a clear divide between the performance of the ontogenetic and phylogenetic methods, especially in terms of CPU time. For the value-based, ontogenetic methods, evaluating and updating values can be computationally expensive. The value-function approximator must be evaluated $O(|A|)$ times per state transition to determine the best action-value estimate, where A is a finite set of actions. Q-MLP and AHC have a notable CPU time advantage over SARSA because their value functions are represented compactly by neural networks which can be evaluated quickly, while the CMAC and case-based memory are coarse-codings have memory requirements and evaluation cost grow exponentially with the dimensionality of the state space.

In contrast, evolutionary methods do not update any agent parameters during interaction with the environment and only need to evaluate a function approximator once per state transition since the policy is represented explicitly.

PGRL is also quite slow as each update to the policy requires sampling $O(|A|T)$ trajectories, where T is the number of state transitions in the initial trajectory of each update. RPG performed best of the ontogenetic methods, but it must be noted that the criteria for success in the referenced work (Wierstra et al., 2007) was 10K steps instead of the 100K steps used with all the other methods (hence the parentheses in all tables for this method).

This task poses very little difficulty for the NE methods. However, NEAT required more than twice as many evaluations as CNE, SANE, and ESP because it explores different topologies that initially behave poorly and require time to develop. For this task the speciation process is an overkill—the task can be solved more efficiently by devoting resources to searching for weights only. All observed performance differences are statistically significant ($p < 0.01$) except between CNE, SANE and ESP.

Method	Evaluations	CPU time
VAPS	(500,000)	(5days)
SARSA-CABA	15,617	6,754
SARSA-CMAC	13,562	2,034
Q-MLP	11,331	340
RWG	8,557	3
RPG	(1,893)	—
NEAT	1,523	15
SANE	1,212	6
CNE	724	15
ESP	589	11
CoSyNE	127	2

Table 2: **One pole with incomplete state information.** The table shows the number of evaluations, CPU time, and success rate of the various methods. Results are the average of 50 simulations, and all differences are statistically significant ($p < 0.01$). The results for VAPS are in parenthesis since only a single unsuccessful run according to our criteria was reported by Meuleau et al. (1999).

5.4.2 INCOMPLETE STATE INFORMATION

This task is identical to the first task except the controller only senses the cart position x and pole angle θ. Therefore, the underlying states $\{x, \dot{x}, \theta, \dot{\theta}\}$ are hidden and the networks need to be recurrent so that the velocities can be computed internally using feedback connections. This makes the task significantly harder since it is more difficult to control the system when the concomitant problem of velocity calculation must also be solved. We were unable to solve this task with AHC and PGRL.

To allow Q-MLP and the SARSA methods to solve this task, we extended their inputs to include the immediately previous cart position, pole angle, and action $(x_{t-1}, \theta_{t-1}, a_{t-1})$ in addition to x_t, θ_t, and a_t. This *delay window* of depth 1 is sufficient to disambiguate process states (Lin and Mitchell, 1992). For VAPS, the state-space was partitioned into unequal intervals, 8 for x and 6 for θ, with the smaller intervals being near the center of the value ranges (Meuleau et al., 1999).

Table 2 compares the various methods in this task. The table shows the number of evaluations and average CPU time for the successful runs.

The results for VAPS are in parenthesis in the table because only a single run was reported by Meuleau et al. (1999). It is clear, however, that VAPS is the slowest method in this comparison, only being able to balance the pole for around 1 minute of simulated time after several days of computation (Meuleau et al., 1999). The evaluations and CPU time for the SARSA methods are the average of the successful runs only (out 29 of 50 for SARSA-CMAC and 35 out of 50 for SARSA-CABA). Of the value-function methods, Q-MLP fared the best, reliably solving the task and doing so much more rapidly than SARSA. Since both the CMAC and the cased-based memory are local function approximators, they require a dense sampling of the state space to obtain good value estimate. The MLP, being a global function approximator, is able to learn values for a whole set of states every time a state is updated. This property has been considered undesirable in some domains because updates at one state can disrupt or unlearn values at distant states. Because the

relatively simple form of the optimal value function for this task, the MLP accelerates learning by providing "useful" information about more of the state space on each update which is especially useful to bootstrap learning at the beginning when there is virtually no information about the value of most states. For more complicated value functions, the potential for instability in the MLP could give local representations the advantage (Boyan and Moore, 1995).

The performance of the five evolutionary methods degrades only slightly compared to the previous task. CoSyNE, CNE, and ESP were two orders of magnitude faster than VAPS and SARSA, one order of magnitude faster than Q-MLP, and approximately twice as fast as SANE and NEAT. CoSyNE was able to balance the pole for over 30 minutes of simulated time usually within 2 seconds of learning CPU time, and do so reliably.

The results on these first two tasks show that the single pole environment is not very challenging. A large part of the search space represents successful solutions, so that simply choosing points at random (i.e., RWG) can compete favorably with other ontogenetic approaches that start at one point and then must make relatively small incremental changes to reach a solution, without not getting stuck in a local minimum.

5.5 Results: Balancing Two Poles

The double pole problem is a better test environment for these methods, representing a significant jump in difficulty. Here the controller must balance two poles of different lengths (1m and 0.1m) simultaneously. The second pole adds two more dimensions to the state-space $(\theta_2, \dot{\theta}_2)$ and non-linear interactions between the poles.

5.5.1 COMPLETE STATE INFORMATION

For this task, CoSyNE was compared with Q-MLP, CNE, SANE, ESP, NEAT, and the published results of RPG, EP, and CMA-ES. Despite extensive experimentation with many different parameter settings, we were unable to get the SARSA methods to solve this task within 12 hours of computation.

Table 3 shows the results for the two-pole configuration with complete state information. Q-MLP compares very well to the NE methods with respect to evaluations, in fact, better than on task 1b, but again lags behind SANE, ESP and NEAT by nearly an order of magnitude in CPU time. ESP and NEAT are statistically even in terms of evaluations, requiring roughly three times fewer evaluations than SANE. In terms of CPU time, ESP has a slight but statistically significant ($p < 0.01$) advantage over NEAT. This is an interesting result because the two methods take such different approaches to evolving neural networks. NEAT is based on searching for an optimal topology, whereas ESP, like CoSyNE, optimizes a single, general topology (i.e., fully recurrent networks). At least in the difficult versions of the pole balancing task, the performance of these two approaches is very similar.

CMA-ES required the fewest number of evaluations, 59 less than CoSyNE on average, although we do not have the CMA-ES run data to test for statistical significance.

5.5.2 INCOMPLETE STATE INFORMATION

Although the previous task is difficult, the controller has the benefit of complete state information. In this task, as in task 1b, the controller does not have access to the velocities, that is, it does not know how fast or in which direction the poles are moving.

Method	Evaluations	CPU time
RWG	474,329	70
EP	307,200	—
CNE	22,100	73
SANE	12,600	37
Q-MLP	10,582	153
RPG	(4,981)	—
NEAT	3,600	31
ESP	3,800	22
CoSyNE	954	4
CMA-ES	895	—

Table 3: **Two poles with complete state information.** The table shows the number of pole balancing attempts (evaluations) and CPU time required by each method to solve the task. Evolutionary Programming data is taken from Saravanan and Fogel (1995), CMA-ES from Igel (2003). Q-MLP, CNE, SANE, NEAT, ESP, CoSyNE data are the average of 50 simulations, and all differences are statistically significant ($p < 0.01$) except the number of evaluations for NEAT and ESP.

Gruau et al. (1996b) were the first to tackle the two-pole problem without velocity information. Although they report the performance for only one simulation, we include their results to put the performance of the other methods in greater perspective. None of the value-function methods we tested made noticeable progress on the task after approximately 12 hours of computation. Therefore, in this task, only the evolutionary methods are compared.

To accommodate a comparison with CE, controllers were evolved using both the standard fitness function used in the previous tasks and also the "damping" fitness function used by Gruau et al. (1996b). The damping fitness is the weighted sum of two separate fitness measurements ($0.1 f_1 + 0.9 f_2$) taken over a simulation of 1000 time steps:

$$f_1 = t/1000,$$

$$f_2 = \begin{cases} 0 & \text{if } t < 100 \\ \left(\dfrac{0.75}{\sum_{i=t-100}^{t} (|x^i| + |\dot{x}^i| + |\theta_1^i| + |\dot{\theta}_1^i|)} \right) & \text{otherwise,} \end{cases}$$

where t is the number of time steps the poles were balanced out of the first 1000 steps. This complex fitness is intended to force the network to compute the pole velocities, and avoid solutions that balance the poles by merely swinging them back and forth (i.e., without calculating the velocities).

Table 4 compares the "surviving" methods for both fitness functions. To determine when the task was solved for the damping fitness function, the best controller from each generation was tested using the standard fitness to see if it could balance the poles for 100K time steps. The results for CE are in parenthesis in the table because only a single run was reported by Gruau et al. (1996b).

Using the damping fitness, CMA-ES, ESP, CNE, NEAT, and CoSyNE required two orders of magnitude fewer evaluations than CE. ESP was three times faster than CNE using either fitness

Method	Evaluations	
	Standard fitness	Damping fitness
RWG	415,209	1,232,296
CE	—	(840,000)
SANE	262,700	451,612
CNE	76,906	87,623
ESP	7,374	26,342
NEAT	—	6,929
RPG	(5,649)	—
CMA-ES	3,521	6,061
CoSyNE	1,249	3,416

Table 4: **Two poles with incomplete state information.** The table shows the number of evaluations for CNE, NEAT, and ESP using the standard fitness function (middle column), and using the damping fitness function (right column). Results are the average of 50 simulations for all methods except CE which is from a single run. All results are statistically significant ($p < 0.01$) .

function, with CNE failing to solve the task about 40% of the time, and NEAT, using small populations of size 16 (Stanley, 2004) performed nearly as well as CMA-ES (damping function). RPG was the only ontogenetic method to make significant progress in this task, again, however, only up to 10K time-steps of balancing.

On this most difficult task CoSyNE outperformed the next best method, CMA-ES, by a factor of two on both fitness functions.

6. Discussion

The results of the comparisons show that the phylogenetic methods (i.e., neuroevolution) are more efficient on this set of tasks than the ontogenetic methods. In the single pole tasks, the value-based ontogenetic methods were outperformed by random search. Our hope is that these results will help put an end to the use of this task for evaluating artificial learning systems. On the more difficult two-pole tasks, only Q-MLP was able to solve the completely observable version (task 2a), and none of the ontogenetic methods could solve the partially observable one (task 2b). In contrast, all of the neuroevolution methods scaled up to the most difficult tasks, with CMA-ES and CoSyNE leading the pack.

The most challenging of the tasks exhibit many of the dimensions of difficulty found in real world control problems: (1) continuous state and action spaces, (2) partial observability, and (3) non-linearity. The first two are problematic for value-based reinforcement learning methods because they either complicate the representation of the value function or the access to it. Neuroevolution deals with them by evolving recurrent networks; the networks can compactly represent arbitrary temporal, non-linear mappings. The success of CoSyNE on tasks of this complexity suggests that it can be applied to the control of real systems that manifest similar properties—specifically, non-linear, continuous systems such as aircraft control, satellite detumbling, and robot bipedal walking.

Other types of environments that are discrete or discontinuous, such as game-playing, job-shop scheduling, and resource allocation may be better served by other learning or optimization strategies.

The CoSyNE implementation used in this paper permuted all members of a subpopulation each generation. This means that it is possible for the networks evaluated in a given generation to not contain any combinations of weights found in the networks of the previous generation. While this maximizes the amount of exploration performed by sampling new networks, good weight combinations may be lost that could lead to a solution more efficiently. This aggressive exploration could become a problem for large networks, such as those that use very high-dimensional vision inputs. Future work will begin by investigating schemes for assigning permutation probabilities to weights (e.g., fitness proportional) in order to retain potential useful building blocks in the system and facilitate search in larger network spaces.

7. Conclusion

Reinforcement learning can in principle be used to control real world systems, but conventional methods scale poorly to large state-spaces and non-Markov environments. In this paper, we have shown that for a set of benchmark tasks that exhibit many of the key dimensions of difficulty found in real world control problems, neuroevolution in general, and CoSyNE in particular, can solve these problems much more reliably and efficiently than non-evolutionary reinforcement learning approaches.

Appendix A. Pole-balancing Equations

The equations of motion for N unjointed poles balanced on a single cart are

$$\ddot{x} = \frac{F - \mu_c sgn(\dot{x}) + \sum_{i=1}^{N} \tilde{F}_i}{M + \sum_{i=1}^{N} \tilde{m}_i},$$

$$\ddot{\theta}_i = -\frac{3}{4l_i}(\ddot{x}\cos\theta_i + g\sin\theta_i + \frac{\mu_{pi}\dot{\theta}_i}{m_i l_i}),$$

where $\tilde{F}_i$ is the effective force from the ith pole on the cart,

$$\tilde{F}_i = m_i l_i \dot{\theta}_i^2 \sin\theta_i + \frac{3}{4} m_i \cos\theta_i (\frac{\mu_{pi}\dot{\theta}_i}{m_i l_i} + g\sin\theta_i),$$

and $\tilde{m}_i$ is the effective mass of the ith pole,

$$\tilde{m}_i = m_i(1 - \frac{3}{4}\cos^2\theta_i).$$

Parameters used for the single pole problem:

Sym.	Description	Value
x	Position of cart on track	[-2.4,2.4] m
θ	Angle of pole from vertical	[-12,12] deg.
F	Force applied to cart	-10,10 N
l	Half length of pole	0.5m
M	Mass of cart	1.0 kg
m	Mass of pole	0.1 kg

Parameters for the double pole problem.

Sym.	Description	Value
x	Position of cart on track	[-2.4,2.4] m
θ	Angle of pole from vertical	[-36,36] deg.
F	Force applied to cart	[-10,10] N
l_i	Half length of ith pole	l_1 = 0.5m l_2 = 0.05m
M	Mass of cart	1.0 kg
m_i	Mass of ith pole	m_1 = 0.1 kg m_2 = 0.01 kg
μ_c	Coefficient of friction of cart on track	0.0005
μ_p	Coefficient of friction if ith pole's hinge	0.000002

Appendix B. Parameter Settings Used in Pole Balancing Comparisons

Below are the parameters used to obtain the results for Q-MLP, SARSA-CABA, SARSA-CMAC, CNE, SANE, ESP, and NEAT. The parameters for VAPS (Meuleau et al., 1999), RPG (Wierstra et al., 2007), CMA-ES (Igel, 2003), EP (Saravanan and Fogel, 1995), and CE2 (Gruau et al., 1996b) along with a detailed description of each method can be found in the cited papers.

Table 5 describes the parameters common to all of the value function methods.

Parameter	Description
ε	greediness of policy
α	learning rate
γ	discount rate
λ	eligibility

Table 5: All parameters have a range of (0,1).

Q-MLP

Parameter	Task		
	1a	1b	2a
ε	0.1	0.1	0.05
α	0.4	0.4	0.2
γ	0.9	0.9	0.9
λ	0	0	0

For all Q-MLP experiments the Q-function network had 10 hidden units and the action space was quantized into 26 possible actions: $\pm 0.1, 0.25, 0.5, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10$.

SARSA-MLP

Parameter	Task		
	1a	1b	2a
ε	0.1	0.1	0.05
α	0.4	0.4	0.1
γ	0.9	0.9	0.9
λ	0	0	0.3

For all Q-MLP experiments the Q-function network had 10 hidden units and the action space was quantized into 26 possible actions: $\pm 0.1, 0.25, 0.5, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10$.

SARSA-CABA

Parameter	Task	
	1a	1b
τ_d	0.03	0.03
τ_k^x	0.05	0.05
τ_k^u	0.1	0.1
ε	0.05	0.05
α	0.4	0.1
γ	0.99	0.99
λ	0.4	0.4

τ_d is the density threshold, τ_k^x and τ_k^u are the smoothing parameters for the input and output spaces, respectively. See Santamaria et al. (1998) for a more detailed description of the Case-Based Memory architecture.

SARSA-CMAC

Parameter	Task	
	1a	1b
ε	0.05	0.05
α	0.4	0.1
γ	0.9	0.9
λ	0.5	0.3
No. of tilings	45: 10 based on $x, \dot{x}, \theta_1$ 5 based on x, θ 5 based on $x, \dot{\theta}$ 5 based on $\dot{x}, \dot{\theta}$ 5 based on x 5 based on $\dot{x}$ 5 based on θ 5 based on $\dot{\theta}$	50 : 10 based on x_t, x_{t-1}, θ_t 10 based on $x, \theta_t, \theta_{t-1}$ 5 based on x_t, θ_t 5 based on x_{t-1}, θ_{t-1} 5 based on x_t 5 based on x_{t-1} 5 based on θ_t 5 based on θ_{t-1}

where x_t and θ_t are the cart position and pole angle at time t. Each variable was divided in to 10 intervals in each tiling. For a more complete explanation of the CMAC architecture see Santamaria et al. (1998).

SANE

Parameter	Task	
	1(a,b)	2(a,b)
no. of neurons	100	200
no. of blueprints	50	100
evals per generation	200	400
size of networks	5	7

The mutation rate for all runs was set to 10%.

CNE

Parameter	Task			
	1a	1b	2a	2b
no. of networks	200	200	400	1000
size of networks	5	5	5	rand [1..9]
burst threshold	10	10	10	15

The mutation rate for all runs was set to 40%. Burst threshold is the number of generations after which burst mutation is activated if the best network found so far is not improved upon. CNE evaluates each of the networks in its population once per generation.

ESP

Parameter	Task			
	1a	1b	2a	2b
network type	FF	FR	FF	FR
initial no. of subpops	5	5	5	5
size of subpopulations	20	20	40	100
evals per generation	200	200	400	1000
burst threshold	10	10	10	5

The mutation rate for all runs was set to 40%. Burst threshold is the number of generations after which burst mutation is activated if the best network found so far is not improved upon. FF denotes a feed-forward network, whereas FR denotes a fully recurrent network.

Acknowledgments

This research was partially funded by the following grants: NSF EIA-0303609, NSF IIS-0083776, THECB (Texas Higher Education Coordinating Board) ARP-003658-476-2001, and CSEM Alpnach and the EU MindRaces project: FP6 511931.

References

J. S. Albus. A new approach to manipulator control: The cerebellar model articulation controller (CMAC). *Journal of Dynamic Systems, Measurement, and Control*, 97(3):220–227, 1975.

C. W. Anderson. Learning to control an inverted pendulum using neural networks. *IEEE Control Systems Magazine*, 9:31–37, 1989.

C. W. Anderson. Strategy learning with multilayer connectionist representations. Technical Report TR87-509.3, GTE Labs, Waltham, MA, 1987.

L. C. Baird and Andrew W. Moore. Gradient descent reinforcement learning. In *Advances in Neural Information Processing Systems 12*, 1999.

R. K. Belew, J. McInerney, and N. N. Schraudolph. Evolving networks: Using the genetic algorithm with connectionist learning. In C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, editors, *Proceedings of the Workshop on Artificial Life (ALIFE '90)*. Reading, MA: Addison-Wesley, 1991. ISBN 0-201-52570-4.

J. A. Boyan and A. W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In *Advances in Neural Information Processing Systems 7*, 1995.

B. Bryant and R. Miikkulainen. Neuroevolution of adaptive teams: Learning heterogeneous behavior in homogeneous multi-agent systems. In *Congress in Evolutionary Computation, Canberra, Australia*, 2003.

P. J. Darwen. *Co-Evolutionary Learning by Automatic Modularization with Speciation*. PhD thesis, University College, University of New South Wales, November 1996.

R. Eriksson and B. Olsson. Cooperative coevolution in inventory control optimization. In *Proceedings of 3rd International Conference on Artificial Neural Networks and Genetic Algorithms*, 1997.

F. Gomez and R. Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5:317–342, 1997.

F. Gomez, D. Burger, and R. Miikkulainen. A neuroevolution method for dynamic resource allocation on a chip multiprocessor. In *Proceedings of the INNS-IEEE International Joint Conference on Neural Networks*, pages 2355–2361, Piscataway, NJ, 2001. IEEE.

F. J. Gomez. *Robust Nonlinear Control through Neuroevolution*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, August 2003. Technical Report AI-TR-03-303.

F. J. Gomez and R. Miikkulainen. Active guidance for a finless rocket using neuroevolution. In E. Cant-Paz et al., editor, *Proceedings of the Genetic Evolutionary Computation Conference (GECCO-03)*. Springer-VerlagBerlin; New York, 2003.

D. Grady. The vision thing: Mainly in the brain. *Discover*, 14:57–66, June 1993.

U. Grasemann and R. Miikkulainen. Effective image compression using evolved wavelets. In *Proceedings of the Genetic Evolutionary Computation Conference (GECCO-05)*, New York, 2005. ACM. ISBN 1-59593-010-8.

B. Greer, H. Hakonen, R. Lahdelma, and R. Miikkulainen. Numerical optimization with neuroevolution. In *Proceedings of the 2002 Congress on Evolutionary Computation (CEC2002)*, 2002.

F. Gruau, D. Whitley, and L. Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 81–89, Cambridge, MA, 1996a. MIT Press.

F. Gruau, D. Whitley, and L. Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. Technical Report NC-TR-96-048, NeuroCOLT, 1996b.

G. Grudic. Simulation code for policy gradient reinforcement learning. http://www.cis.upenn.edu/ grudic/PGRLSim/, 2000.

N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001.

S. A. Harp, T. Samad, and A. Guha. Towards the genetic synthesis of neural networks. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 360–369, 1989.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In S. C. Kremer and J. F. Kolen, editors, *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE Press, 2001.

J. H. Holland and J. S. Reitman. Cognitive systems based on adaptive algorithms. In D. A. Waterman and F. Hayes-Roth, editors, *Pattern-Directed Inference Systems*. Academic Press, New York, 1978.

J. Horn, D. E. Goldberg, and K. Deb. Implicit niching in a learning classifier system: Nature's way. *Evolutionary Computation*, 2(1):37–66, 1994.

P. Husbands and F. Mill. Simulated co-evolution as the mechanism for emergent planning and scheduling. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 264–270. San Francisco, CA: Morgan Kaufmann, 1991. ISBN 1-55860-208-9.

C. Igel. Neuroevolution for reinforcement learning using evolution strategies. In R. Reynolds, H. Abbass, K. C. Tan, B. McKay, D. Essam, and T. Gedeon, editors, *Congress on Evolutionary Computation (CEC 2003)*, volume 4, pages 2588–2595. IEEE, 2003.

J.-S. R. Jang. Self-learning fuzzy controllers based on temporal backpropagation. *IEEE Transactions on Neural Networks*, 3(5):714–723, September 1992.

T. Jansen and R. P. Wiegand. The cooperative coevolutionary (1+1) ea. *Evolutionary Computation*, 12(4), 2004.

T. Jansen and R. P. Wiegand. Exploring the explorative advantage of the CC (1+1) ea. In E. Cant-Paz et al., editor, *Proceedings of the Genetic Evolutionary Computation Conference (GECCO-03)*. Springer-VerlagBerlin; New York, 2003.

D. Jefferson, R. Collins, C. Cooper, M. Dyer, M. Flowers, R. Korf, C. Taylor, and A. Wang. Evolution as a theme in artificial life: The Genesys/Tracker system. In C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, editors, *Proceedings of the Workshop on Artificial Life (ALIFE '90)*. Reading, MA: Addison-Wesley, 1991. ISBN 0-201-52570-4.

H. Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4:461–476, 1990.

J. R. Koza. *Genetic Programming*. MIT Press, Cambridge, MA, 1991.

L.-J. Lin. Self-improving reactive agents based on reinforcement learning, planning, and teaching. *Machine Learning*, 8(3):293–321, 1992.

L.-J. Lin and T. M. Mitchell. Memory approaches to reinforcement learning in non-Markovian domains. Technical Report CMU-CS-92-138, Carnegie Mellon University, School of Computer Science, May 1992.

A. Lubberts and R. Miikkulainen. Co-evolving a go-playing neural network. In *Coevolution: Turning Adaptive Algorithms Upon Themselves, Birds-of-a-Feather Workshop, Genetic and Evolutionary Computation Conference (GECCO-2001)*, 2001.

M. Mandischer. Representation and evolution of neural networks. In R.F. Albrecht, C.R. Reeves, and N.C. Steele, editors, *Proceedings of the Conference on Artificial Neural Nets and Genetic Algorithms at Innsbruck, Austria*, pages 643–649. Springer-Verlag, 1993.

N. Meuleau, L. Peshkin, K.-E. Kim, and L. P. Kaelbling. Learning finite state controllers for partially observable environments. In *15th International Conference of Uncertainty in AI*, 1999.

D. Michie and R. A. Chambers. BOXES: An experiment in adaptive control. In E. Dale and D. Michie, editors, *Machine Intelligence*. Oliver and Boyd, Edinburgh, UK, 1968.

G. Miller and D. Cliff. Co-evolution of pursuit and evasion i: Biological and game-theoretic foundations. Technical Report CSRP311, School of Cognitive and Computing Sciences, University of Sussex, Brighton, UK, 1994.

D. E. Moriarty. *Symbiotic Evolution of Neural Networks in Sequential Decision Tasks*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 1997. Technical Report UT-AI97-257.

S. Nolfi and D. Parisi. Learning to adapt to changing environments in evolving neural networks. Technical Report 95-15, Institute of Psychology, National Research Council, Rome, Italy, 1995.

L. Panait, S. Luke, and J. F. Harrison. Archive-based cooperative coevolutionary algorithms. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 345–352, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-186-4. doi: http://doi.acm.org/10.1145/1143997.1144060.

J. Paredis. Steps towards co-evolutionary classification neural networks. In R. A. Brooks and P. Maes, editors, *Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems (Artificial Life IV)*, pages 102–108. Cambridge, MA: MIT Press, 1994. ISBN 0-262-52190-3.

J. Paredis. Coevolutionary computation. *Artificial Life*, 2:355–375, 1995.

A. S. Perez-Bergquist. Applying ESP and region specialists to neuro-evolution for Go. Technical Report CSTR01-24, Department of Computer Sciences, The University of Texas at Austin, 2001.

J. B. Pollack, A. D. Blair, and M. Land. Coevolution of a backgammon player. In C. G. Langton and K. Shimohara, editors, *Proceedings of the 5th International Workshop on Artificial Life: Synthesis and Simulation of Living Systems (ALIFE-96)*. Cambridge, MA: MIT Press, 1996. ISBN 0-262-62111-8.

M. A. Potter and K. A. De Jong. Evolving neural networks with collaborative species. In *Proceedings of the 1995 Summer Computer Simulation Conference*, 1995.

C. D. Rosin. *Coevolutionary Search Among Adversaries*. PhD thesis, University of California, San Diego, San Diego, CA, 1997.

J. C. Santamaria, R. S. Sutton, and A. Ram. Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavior*, 6(2):163–218, 1998.

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

K. O. Stanley. *Efficient Evolution of Neural Networks Through Complexification*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, August 2004. Technical Report AI-TR-04-314.

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

R. S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *Advances in Neural Information Processing Systems 8*, pages 1038–1044. Cambridge, MA: MIT Press, 1996.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998. ISBN 0-262-19398-1.

R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems 12*, volume 12, pages 1057–1063. MIT Press, 2000.

G. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8:257–277, 1992.

H. M. Voigt, J. Born, and I. Santibanez-Koref. Evolutionary structuring of artificial neural networks. Technical report, Technical University Berlin, Bio- and Neuroinformatics Research Group, 1993.

C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.

P. Werbos. Backpropagation through time: what does it do and how to do it. In *Proceedings of IEEE*, volume 78, pages 1550–1560, 1990.

B. A. Whitehead and T. D. Choate. Cooperative–competitive genetic evolution of radial basis function centers and widths for time series prediction. *IEEE Transactions on Neural Networks*, 1995.

S. Whiteson, N. Kohl, R. Miikkulainen, and P. Stone. Evolving keepaway soccer players through task decomposition. In E. Cant-Paz et al., editor, *Proceedings of the Genetic Evolutionary Computation Conference (GECCO-03)*. Springer-VerlagBerlin; New York, 2003.

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

R. P. Wiegand. *An Analysis of Cooperative Coevolutionary Algorithms*. PhD thesis, George Mason University, Fall 2003.

R. P. Wiegand, W. C. Liles, and K. A. De Jong. An empirical analysis of collaboration methods in cooperative coevolutionary algorithms. In L. Spector et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1235–1242. San Francisco, CA: Morgan Kaufmann, 2001. ISBN 1-55860-774-9. URL `citeseer.ist.psu.edu/481900.html`.

A. Wieland. Evolving neural network controllers for unstable systems. In *Proceedings of the International Joint Conference on Neural Networks* (Seattle, WA), pages 667–673. Piscataway, NJ: IEEE, 1991.

D. Wierstra, A. Foerster, J. Peters, and J. Schmidhuber. Solving deep memory pomdps with recurrent policy gradients. In *International Conference on Artificial Neural Networks*, 2007.

B. Yamauchi and R. D. Beer. Integrating reactive, sequential, and learning behavior using dynamical neural networks. In D. Cliff, P. Husbands, J.-A. Meyer, and S. W. Wilson, editors, *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, pages 382–391. Cambridge, MA: MIT Press, 1994. ISBN 0-262-53122-4.

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