Efficient Non-linear Control Through Neuroevolution

Faustino Gomez[1], Jürgen Schmidhuber[1,2], and Risto Miikkulainen[3]

[1] Dalle Molle Institute for Artificial Intelligence (IDSIA), Galleria 2, Lugano, CH
[2] Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany
[3] Department of Computer Sciences, University of Texas, Austin, TX 78712 USA

Abstract. Many complex control problems are not amenable to traditional controller design. Not only is it difficult to model real systems, but often it is unclear what kind of behavior is required. Reinforcement learning (RL) has made progress through direct interaction with the task environment, but it has been difficult to scale it up to large and partially observable state spaces. In recent years, neuroevolution, the artificial evolution of neural networks, has shown promise in tasks with these two properties. This paper introduces a novel neuroevolution method called CoSyNE that evolves networks at the level of weights. In the most extensive comparison of RL methods to date, it was tested in difficult versions of the pole-balancing problem that involve large state spaces and hidden state. CoSyNE was found to be significantly more efficient and powerful than the other methods on these tasks, forming a promising foundation for solving challenging real-world control tasks.

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, noisy, and unstable. The problem with designing or programming controllers for such systems by conventional 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 reinforcement learning (RL; [1]). Instead of trying to pre-program a response to every likely situation, an agent *learns* the task by interacting with the environment. In principle, RL methods can solve these problems: they do not require a mathematical model of the environment (i.e. the state transition probabilities), and can solve many problems where examples of correct behavior are not available. However, in practice, it has turned out difficult to scale them up to large state spaces and non-Markov tasks where the state of the environment is not fully observable. This is an important challenge because the real-world is continuous (i.e. infinite number of states) and artificial agents, like natural organisms, are necessarily constrained in their ability to fully perceive their environment.

Recently, methods for evolving artificial neural networks or *neuroevolution* [2], especially those that coevolve network functional units [3, 4, 5], have shown promising results on continuous, non-Markov tasks. The method introduced in this paper, Cooperative Synapse NeuroEvolution (CoSyNE), extends the idea of coevolving network components to the level of individual synaptic weights. The goal of this paper is to compare CoSyNE to a wide range of other learning systems in a setting that is challenging yet practical. To this end, a set of pole-balancing tasks is used ranging from the familiar simple versions to versions that are extremely difficult even for the most advanced methods.

2 Cooperative Synapse NeuroEvolution (CoSyNE)

Cooperative Synapse Neuroevolution (CoSyNE) uses cooperative coevolution to construct neural networks, but unlike other methods of this type (e.g. SANE [3] and ESP [5]) it searches at the level of individual network weights rather than neurons.

Figure 1 describes the CoSyNE procedure 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($n, m, \Psi$)
 1: Initialize $\mathcal{P} = \{P_1, \ldots, P_n\}$
 2: **repeat**
 3:     **for** $j = 1$ to $m$ **do**
 4:         $\mathbf{x}_j \Leftarrow (x_{1j}, \ldots, x_{nj})$
 5:         Evaluate($\mathbf{x}_j, \Psi$)
 6:     **end for**
 7:     $\mathcal{O} \Leftarrow$ Recombine($\mathcal{P}$)
 8:     **for** $k = 1$ to $l$ **do**
 9:         $x_{i,m-k} \Leftarrow o_{ik}$
10:     **end for**
11:     **for** $i = 1$ to $n$ **do**
12:         permute($P_i$)
13:     **end for**
14: **until** solution is found
```

Fig. 1. The CoSyNE Algorithm

CoSyNE then loops through a sequence of *generations* until a sufficiently good network is found (lines 2-14). 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.

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 $\mathcal{O}$ consisting of l new network chromosomes $\mathbf{o}_k$, where $o_{ik} = \mathcal{O}_{ik} \in O_i, k = 1..l$. The weights in each of the offspring chromosomes are then added to $\mathcal{P}$ by replacing the least fit weights in their corresponding subpopulation (lines 8-10).

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 (lines 11-13) so that each weight forms part of a potentially different network in the next generation.

Permuting the subpopulations increases diversity by allowing CoSyNE to sample networks that would not be generated through recombination alone. This means that which weights are retained in the population from one generation to the next is not determined only by which networks scored well in the previous generation, but rather by a broader sampling of the possible m^n networks that can be formed by selecting a weight from each subpopulation. More precisely, each generation the offspring lie within a subspace that is defined by all possible applications of the genetic operators to the set of parents (i.e. the subspace *spanned* by the parents). Each successive generation produces offspring from a subspace contained within the previous one (except for some sampling outside due to mutation), as the population converges to virtually a single search point. With permutation, points can be sampled outside of this subspace by forming networks from weights not found in the current set of parents. The overall effect is to make the algorithm less greedy because weights have a chance to reproduce even if they were not part of the parent chromosomes in previous generations.

Cooperative coevolution in general can delay convergence through this process, but because CoSyNE evolves at the lowest possible level of granularity (the individual parameter) the number of possible networks m^n is maximized. Therefore, there are a maximum number of ways to sample outside of the parent subspace and delay convergence, allowing CoSyNE more time to put the pieces together to form a good network.

The basic CoSyNE framework does not specify how the weights are grouped in the chromosomes (i.e. which entry in the chromosome corresponds to which synapse) or which 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 input, output, and recurrent weight segments. For the genetic operators we use multi-point crossover where 1-point crossover is applied to each neuron segment of the chromosome is used to generate the offspring, and mutation where each weight in $\mathcal{P}$ has a small probability of being changed to a new value chosen at random from the initial weight range $[-\alpha, \alpha]$.

3 Experiments in Pole-Balancing

CoSyNE was compared experimentally to a broad range of learning algorithms on a sequence of increasingly difficult versions of the pole-balancing task. The basic pole-balancing system consists of a pole hinged to a wheeled cart on a finite stretch of track that must be balance by applying a force to the cart at regular intervals. Although this task has been a popular artificial learning testbed for over 30 years, it turns out that the basic pole-balancing problem can be solved easily by random search. To make the problem more challenging, four task configurations of increasing difficulty (due to [6]) were used: one pole with complete state information (1a) and incomplete state information (1b), and two poles with complete state information (2a) and incomplete state information (2b). Task 1a is the classic one-pole configuration where the controller receives all four state variables: the position and velocity of the cart $(x, \dot{x})$, and the angle and angular velocity of the pole $(\theta_1,\dot{\theta}_1)$. In 1b, the controller only has access to x and θ_1; it does not receive the velocities $(\dot{x}, \dot{\theta}_1)$. In 2a, the system now has a second pole $(\theta_2, \dot{\theta}_2)$ next to the first, making the state-space six-dimensional, and non-linear. 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. The initial angle of the long pole was set to 4° from vertical for all trials. A task was considered solved if a network could balance the pole(s) 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 one-pole tasks, and 30 weights for the two-pole tasks. Mutation was set to 5% in all of the experiments. All simulations were run on a 1.50GHz Intel Xeon.

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 [6] for the equations of motion and parameters used). 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.

3.1 Other Methods

CoSyNE was compared to 14 other learning methods: seven *single-agent* and seven *evolutionary* methods. Due to space limitations, the reader is referred to the original papers and [5] for parameter settings and implementation details.
Single-Agent Methods

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

Policy Gradient RL (PGRL; [7]) where sampled Q-values are used to differentiate the performance of the policy with respect to its parameters. The policy was implemented by a feed-forward network (FNN) with one hidden layer.

Value and Policy Search (VAPS; [8]) extends the work of Baird et al. [9] to policies that can make use of memory. The algorithm uses stochastic gradient descent to search the space of finite policy graph parameters.

Q-learning with MLP (Q-MLP)**:** The basic Q-learning algorithm [10] using a an FNN trained with backpropagation to map state–action pairs to Q-values.

Sarsa(λ) with Case-Based function approximator (SARSA-CABA; [11])**:** This method uses on-policy Temporal Difference control with eligibility traces that uses a case-based memory to approximate the Q-function.
Sarsa(λ) with CMAC function approximator (SARSA-CMAC; [11])**:** This method is the same as SARSA-CABA except that it uses a Cerebellar Model Articulation Controller instead of a case-based memory.
Adaptive Heuristic Critic (AHC; [12])**:** uses a learning agent composed of two components: an *actor* (policy) and a *critic* (value-function), both implemented using an FNN trained with a variant of backpropagation.

Evolutionary Methods

Symbiotic, Adaptive Neuro-Evolution (SANE; [3]) is a cooperative coevolutionary method that evolves neurons in a single population.
Conventional Neuroevolution (CNE) is our implementation of single-population neuroevolution similar to the algorithm used in [6], where each chromosome in the population represents a complete neural network.
Evolutionary Programming (EP; [13]) is a general mutation-based evolutionary method that can be used to search the space of neural networks.
Cellular Encoding (CE; [14]) uses Genetic Programming to evolve graph-rewriting programs that control how neural networks are constructed.
Covariance Matrix Adaptation Evolutionary Strategies (CMA-ES; [15]) evolves the covariance matrix of the mutation operator in evolutionary strategies. The results in the pole-balancing domain were obtained by Igel [16].
NeuroEvolution of Augmenting Topologies (NEAT; [17]) is a neuroevolution method that evolves topology as well as synaptic weights.
Enforced SubPopulation (ESP; [5]) cooperatively coevolves neurons in a separate subpopulation for each network unit.

For Q-MLP, SANE, CNE, ESP, NEAT, and CoSyNE, experiments were run using our own code. For PGRL, AHC, SARSA, publicly available code from [18], [12], and [11], was used respectively, modified for the pole-balancing domain. For VAPS, EP, CMA-ES, and CE, the results were taken from the papers cited above.

3.2 Results

Balancing one pole is a relatively easy problem that gives us a base performance measurement 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. Table 1 shows the results for the this task for both complete and incomplete state information.

The results for task 1a 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. The other single-agent methods were all significantly slower than the evolutionary methods, especially in terms of CPU time. Depending on the kind of function approximator, the amount of computation required to evaluate and update the value-functions used by AHC, SARSA, and Q-learning can prove costly.

In contrast, the 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.

Task 1b is notably harder since in addition to controlling the system, the concomitant problem of velocity calculation must also be solved. Despite considerable effort, we were unable to solve this task with AHC and PGRL.

To make Q-MLP and the SARSA methods effective, their inputs were extended to include also the immediately previous cart position, pole angle, and action $(x_{t-1}, \theta_{t-1}, a_{t-1})$. This *delay window* of depth 1 is sufficient to disambiguate process states.

Table 1. Results for balancing one pole. Average of 50 simulations. All differences are statistically significant ($p < 0.01$).

Method	with velocities		w/out velocities	
	Evals	CPU	Evals	CPU
VAPS	—	—	(500k)	(5days)
AHC	189,500	95	failed	
PGRL	28,779	1,163	failed	
Q-MLP	2,056	53	11,311	340
SARSA-CABA	965	1,713	15,617	6,754
SARSA-CMAC	540	487	13,562	2,034
NEAT	743	7	1,523	6
CNE	352	5	724	5
SANE	302	5	1,212	6
ESP	289	4	589	5
CMA-ES	283	—	—	—
RWG	199	2	8,557	3
CoSyNE	98	1	127	2

VAPS is the slowest method in this comparison, with the single reported run (in parentheses) only balancing the pole for around 1 minute of simulated time after several days of computation [8]. Results for the SARSA methods are the average of successful runs only. Of the single-agent methods Q-MLP fared the best, reliably solving the task and doing so much more rapidly than SARSA.

The performance of the six 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.

Table 2. Two poles with velocities Average of 50 simulations. EP results taken from [13], CMA-ES from [16]. All differences are statistically significant ($p < 0.001$) except the number of evaluations for NEAT and ESP.

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

Balancing two poles represents a significant jump in difficulty. For task 2a, CoSyNE was compared with Q-MLP, CNE, SANE, NEAT, ESP, and the published results of 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 2 shows the results for this task. Q-MLP compares very well to the evolutionary methods with respect to evaluations, in fact, better than on task 1b, but again lags behind SANE, NEAT, and ESP 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. CMA-ES required the fewest number of evaluations, just edging out CoSyNE.

For task 2b, none of the single-agent methods made noticeable progress after 12 hours of computation. Therefore, only neuroevolution methods are compared. To allow for a comparison with CE, controllers were evolved using both the standard fitness function used in the previous tasks, and a the "damping" fitness function (used in [14]) that prevents controllers from solving the task by simply swinging the poles back and forth.

Table 3 compares the six neuroevolution 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 100,000 time steps. The results for CE are in parentheses in the table because only a single run was reported in [14].

Table 3. Two poles without velocities. Average of 50 simulations. All results are statistically significant except for the difference between ESP and NEAT using the standard fitness.

Method	Evaluations	
	Standard fit.	Damping fit.
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	24,543
CMA-ES	3,521	6,061
CoSyNE	1,249	3,416

Using the damping fitness, ESP, CNE, NEAT, and CoSyNE required an order of magnitude fewer evaluations than SANE and CE. ESP and NEAT were three times faster than CNE using either fitness function, with CNE failing to solve the task about 40% of the time. CoSyNE was the most efficient method for both fitness measures, outperforming ESP and NEAT by a factor of six on the standard fitness, and CMA-ES by a factor of two on the damping fitness.

4 Discussion and Conclusion

The comparison results show that the evolutionary methods are more efficient than the single-agent methods in this set of tasks. The best single-agent method in task 1a required an order of magnitude more CPU time than NE, and the transition from 1a to 1b represented a significant challenge, causing some of them to fail and others to take 30 times longer than NE. Only Q-MLP was able to solve task 2a and none of the single-agent methods could solve task 2b. In contrast, all of the evolutionary methods scaled up to the most difficult tasks, with CoSyNE beating the next fastest method by a wide margin.

The most challenging task exhibits 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 conventional 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.

Acknowledgments

This research was 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

1. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA (1998)
2. Yao, X.: Evolving artificial neural networks. Proceedings of the IEEE **87**(9) (1999)
3. Moriarty, D.E.: Symbiotic Evolution of Neural Networks in Sequential Decision Tasks. PhD thesis, University of Texas at Austin (1997) Tech. Rep. UT-AI97-257.
4. Potter, M.A., De Jong, K.A.: Evolving neural networks with collaborative species. In: Proceedings of the 1995 Summer Computer Simulation Conference. (1995)
5. Gomez, F.J.: Robust Nonlinear Control through Neuroevolution. PhD thesis, University of Texas at Austin (2003) Tech. Rep. AI-TR-03-303.
6. Wieland, A.: Evolving neural network controllers for unstable systems. In: Proceedings of the International Joint Conference on Neural Networks (Seattle, WA), Piscataway, NJ: IEEE (1991) 667–673
7. Sutton, R.S., McAllester, D., Singh, S., Mansour, Y.: Policy gradient methods for reinforcement learning with function approximation. In: Advances in Neural Information Processing Systems 12. Volume 12., MIT Press (2000) 1057–1063
8. Meuleau, N., Peshkin, L., Kim, K.E., Kaelbling, L.P.: Learning finite state controllers for partially observable environments. In: 15th International Conference of Uncertainty in AI. (1999)
9. Baird, L.C., Moore, A.W.: Gradient descent reinforcement learning. In: Advances in Neural Information Processing Systems 12. (1999)
10. Watkins, C.J.C.H., Dayan, P.: Q-learning. Machine Learning **8**(3) (1992) 279–292
11. Santamaria, J.C., Sutton, R.S., Ram, A.: Experiments with reinforcement learning in problems with continuous state and action spaces. Adaptive Behavior **6**(2) (1998)

12. Anderson, C.W.: Strategy learning with multilayer connectionist representations. Technical Report TR87-509.3, GTE Labs, Waltham, MA (1987)
13. Saravanan, N., Fogel, D.B.: Evolving neural control systems. IEEE Expert (1995)
14. Gruau, F., Whitley, D., Pyeatt, L.: A comparison between cellular encoding and direct encoding for genetic neural networks. NC-TR-96-048, NeuroCOLT (1996)
15. Hansen, N., Ostermeier, A.: Completely derandomized self-adaptation in evolution strategies. Evolutionary Computation **9**(2) (2001) 159–195
16. Igel, C.: Neuroevolution for reinforcement learning using evolution strategies. In Proceedings of the Congress on Evolutionary Computation. IEEE (2003)
17. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evolutionary Computation **10** (2002) 99–127
18. Grudic, G.: http://www.cis.upenn.edu/ grudic/PGRLSim/ (2000)