ECAI 2010
H. Coelho et al. (Eds.)
IOS Press, 2010

doi:10.3233/978-1-60750-606-5-909

A NEAT Way for Evolving Echo State Networks

Kyriakos C. Chatzidimitriou[1] and **Pericles A. Mitkas**[2]

Abstract. The Reinforcement Learning (RL) paradigm is an appropriate formulation for agent, goal-directed, sequential decision making. In order though for RL methods to perform well in difficult, complex, real-world tasks, the choice and the architecture of an appropriate function approximator is of crucial importance. This work presents a method of automatically discovering such function approximators, based on a synergy of ideas and techniques that are proven to be working on their own. Using Echo State Networks (ESNs) as our function approximators of choice, we try to adapt them, by combining evolution and learning, for developing the appropriate ad-hoc architectures to solve the problem at hand. The choice of ESNs was made for their ability to handle both non-linear and non-Markovian tasks, while also being capable of learning online, through simple gradient descent temporal difference learning. For creating networks that enable efficient learning, a neuroevolution procedure was applied. Appropriate topologies and weights were acquired by applying the NeuroEvolution of Augmented Topologies (NEAT) method as a meta-search algorithm and by adapting ideas like historical markings, complexification and speciation, to the specifics of ESNs. Our methodology is tested on both supervised and reinforcement learning testbeds with promising results.

1 INTRODUCTION

Modeling the mechanisms of learning and decision making of autonomous agents as Reinforcement Learning (RL) problems is an appropriate match [13]. An autonomous agent following the RL paradigm will work towards maximizing the total amount of reward it receives over time, by making changes in its policy (i.e. the mapping of state to actions) based on feedback returned by interacting with its environment. For complex, real-world tasks though, with large, continuous state and action spaces, there is a need for a Function Approximator (FA) to take the role of modeling the policy and provide generalization capabilities to the RL algorithm [14].

In order to automate the task of finding efficient FAs per problem, the area of adaptive function approximation was developed [13]. The goal of the area is to adapt, with little or no human input, the parameters of the FA to the problem at hand, through the synthesis of learning and evolution. Evolution (global search) is applied in order to find FAs that are better able to learn (local search), through Temporal Difference (TD) learning in the RL case [17].

Towards this direction, we present a methodology that synthesizes ideas, proved to work on their own, into a single bundle and in a novel fashion. Our methodology comprises three components: i) a FA, ii) the coupling of learning and evolution and iii) a NeuroEvolution (NE) method. For the first component, our FA of choice is Echo State Networks (ESNs) [5, 6], from the realm of Reservoir Computing (RC). Since we require the FA to be able to a) capture any existing non-linear dynamics of the environment and b) model any existing non-Markovian state signals, ESNs seem like an appropriate solution, due to their properties inherited from their Recurrent Neural Network (RNN) profile. Adding the fact that learning can be applied in an on-line fashion, using linear learning rules to the output layer of the network, ESNs make for a very powerful FA and an appropriate candidate for a dual adaptation through both learning and evolution, the second component of our methodology.

Finally, as the third component, we chose NeuroEvolution of Augmented Topologies (NEAT) [11, 12], a state-of-the-art NE method that is capable of Topology and Weight Evolving of Artificial Neural Networks (TWEANN). NEAT is used in the form of a meta-search evolutionary algorithm to evolve ESNs that will efficiently learn to solve the given task. In particular, we transformed the major ideas behind the NEAT method: a) use crossover with the help of historical markings, b) perform speciation to protect innovation and c) apply complexification by starting minimally and augmenting topologies in order to quickly develop parsimonious networks, to the specifics of ESN computing.

The paper continuous as follows: Section 2 summarizes the background behind the components involved. Section 3 provides a short survey of recent relevant work. Section 4 describes the methodology, while Section 5 presents the experiments, exhibits the obtained outcomes and comments on the results. Finally, Section 6 summarizes the paper with the conclusions and discussion on future improvements.

2 BACKGROUND

2.1 Echo State Networks

In this section we provide a brief overview of the ESN approach, focusing on certain aspects useful to the rest of the paper. More detailed analysis and examples can be found in [5, 6], while for an overview of the RC area the reader could refer to [8].

The idea behind RC and ESNs, is that a random RNN, created under certain algebraic constraints, could be driven by an input signal to create a rich set of dynamics in its reservoir of neurons, forming non-linear response signals. These signals, along with the input signals, could be combined to form the so-called *read-out function*, a linear combination of features, $y = \mathbf{w}^T \cdot \phi(\mathbf{x})$, which constitutes the prediction of the desired output signal, given that the weights, w, are trained accordingly. The recurrences, present in the reservoir in the form of cycles in the connection topology, enable the ESN to maintain a dynamic memory and process temporal information. Even though other read-out functions, like feedforward neural networks, can be connected to the reservoir, a linear function is enough

[1] Aristotle University of Thessaloniki / Centre for Research and Technology Hellas, Greece, email: kyrcha@issel.ee.auth.gr
[2] Aristotle University of Thessaloniki / Centre for Research and Technology Hellas, Greece, email: mitkas@eng.auth.gr

to achieve excellent performance in practical applications.

A basic form of an ESN is presented in Figure 1. The reservoir consists of a layer of K input units, connected to N reservoir units through a $N \times K$ weighted connection matrix W^{in}. The connection matrix of the reservoir, W, is a $N \times N$ matrix. Optionally a back-projection matrix W^{back} could be present, with dimensions $N \times L$, where L is the number of output units, connecting the outputs back to the reservoir neurons. The weights from input units (linear features) and reservoir units (non-linear features) to the output are collected into a $L \times (K + N)$ matrix, W^{out}. For this work, the reservoir units use $f(x) = tanh(x)$ as an activation function, while the output units use either $g(x) = tanh(x)$ or the identity function, $g(x) = x$.

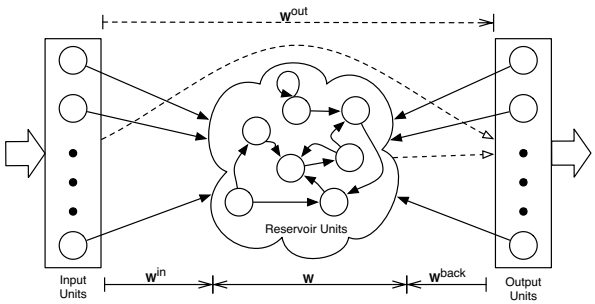

Figure 1. A basic form of an ESN. Solid arrows represent fixed weights and dashed arrows adaptable weights.

Best practices for generating ESNs, i.e. procedures for generating the random connection matrices W^{in}, W and W^{back}, can be found in [6, 8]. Briefly, these are: (i) W should be sparse, (ii) the mean value of weights should be around zero, (iii) N should be large enough to introduce more features for better prediction performance, (iv) the spectral radius, ρ, of W should be less than 1 to practically (and not theoretically) ensure that the network will be able to function as an ESN. Finally, a weak uniform white noise term can be added to the features for stability reasons.

In this work, we consider discrete time models and ESNs without backprojection connections. As a first step, we scale and shift the input signal, $\mathbf{u} \in \mathbb{R}^K$, depending on whether we want the network to work in the linear on the non-linear part of the sigmoid function. The reservoir feature vector, $\mathbf{x} \in \mathbb{R}^N$, is given by Equation 1:

$$\mathbf{x}(t+1) = \mathbf{f}(\mathbf{W}^{in}\mathbf{u}(t+1) + \mathbf{W}\mathbf{x}(t) + \mathbf{v}(t+1)) \quad (1)$$

where $\mathbf{f}$ is the element-wise application of the reservoir activation function and $\mathbf{v}$ is a uniform white noise vector. The output, $\mathbf{y} \in \mathbb{R}^L$, is then given by Equation 2:

$$\mathbf{y}(t+1) = \mathbf{g}(\mathbf{W}^{out}[\mathbf{u}(t+1)|\mathbf{x}(t+1)]) \quad (2)$$

with $\mathbf{g}$, the element-wise application of the output activation function.

For supervised learning tasks, the problem can be formulated as a linear regression problem and the output weights can be determined using a variety of numerical linear algebra algorithms [8]. For RL tasks with K continuous states and L discrete actions, we can use an ESN to model a Q-value function, where each network output unit l, can be mapped to an action $a_l, l = 1 \ldots L$, with the network output value y_l denoting the long-term discounted value, $Q(\mathbf{s}, a_l)$ of performing action a_l, when the agent is at state $\mathbf{s}$. Given $g(x) = x$, this Q-value can be represented by an ESN as:

$$y_l = Q(\mathbf{s}, a_l) = \sum_{i=1}^{K} w_{li}^{out} s_i + \sum_{i=K+1}^{K+N} w_{li}^{out} x_{i-K}, l = 1, \ldots, L \quad (3)$$

while actions can be chosen under the ϵ-greedy policy [14].

Linear Gradient Descent (GD) SARSA TD-learning can be used to adapt weights [14, 15], where the update equations take the form of:

$$\delta = r + \gamma Q(\mathbf{s}, a') - Q(\mathbf{s}, a_l) \quad (4)$$

$$\mathbf{w}_l^{out\prime} = \mathbf{w}_l^{out} + \alpha\delta[\mathbf{s}|\mathbf{x}] \quad (5)$$

with a' the next action to be selected and α the learning rate.

2.2 NEAT

NEAT [11, 12] is a TWEANN algorithm, constructed on four principles that made it a reference algorithm in the area of NE. First of all, the network, i.e. the phenotype, is encoded as a linear genome (genotype), making it memory efficient with respect to algorithms that work with full weight connection matrices. Secondly, using the notion of *historical markings*, newly created connections are annotated with innovation numbers. NEAT during crossover aligns parent genomes by matching the innovation numbers and performs crossover on these matching genes (connections). Due to the competing conventions problem [9], crossover is usually not applied in NE, but historical markings have made crossover practically applicable. Under the NEAT method, it has been proved experimentally that crossover could do more good than harm. The third principle is to protect innovation through *speciation*, by clustering organisms into species in order for them to have time to optimize by competing only in their own niche. Last but not least, NEAT starts with minimal networks, that is networks with no hidden units, in order (a) to initially start with a minimal search space and (b) to justify every complexification made in terms of fitness. NEAT complexifies networks through the application of structural mutations, by adding nodes and connections, and further adapts the networks through weight mutation by perturbing or restarting weight values. The above successful ideas could be used in other NE settings in the form of a meta-search evolutionary procedure. In our case, we follow these ideas to achieve an efficient search in the space of ESNs. A brief description of the NEAT procedure can be found in Section 4.

3 RELATED WORK

ESNs have been used before as FAs for RL tasks in [15], but in their basic, non-optimized form. In the same paper the authors show that ESNs using the SARSA(λ) algorithm have the same convergence behavior as a linear FA with SARSA(λ), with the extra value of holding for k-order Markov Decision Processes (MDPs) as well, making ESNs more theoretically desirable.

When it became evident that better results could be obtained by optimizing the reservoir, instead of randomly generating it, many methods came up that do exactly that. In [1], next ascent local search is used to find better connection topologies of W. In [4], the authors move away from performing a TWEANN method for evolving the network, but rather focus on evolving the parameters, N, ρ and D, the connection density of W, to find a good set of values. They then refine the fittest ESN produced by applying a second evolutionary method directly on its weight genome, constructed by vectorizing

the weight matrices of the ESN. In two more recent approaches [7] applies the CMA-ES algorithm to optimize the macroscopic parameter triplet: N, ρ and D, while [2] evolves output weights (matrix W^{out}) using evolutionary strategies.

The closest matches to our method can be found in [17, 10]. In [17], NEAT is used, as is, to develop ad-hoc Neural Networks (NNs) without recurrent connections so that TD-learning could be applied using standard error backpropagation updates to further adapt weights towards a solution. In [10] the authors use their own evolutionary TWEANN method to develop ESNs with competitive results in time-series prediction problems.

Our method differs from the aforementioned approaches in several aspects. First of all we are trying to optimize all parameters of the reservoir, i.e. the connection topology, the weights, N, ρ, D, using NE and TD-learning. This differentiates our focus from any work that optimizes only macroscopic reservoir parameters. The difference from [17], apart from the obvious use of ESNs versus ad-hoc NNs, is that a) we have the additional property of ESNs to maintain a memory and thus solve tasks with non-Markovian state signals, and b) the network is trained using simple linear TD-learning instead of error backpropagation. Finally, this work differs from [10] in that a) it uses an established NE method as a meta-search algorithm, b) it applies different operators and techniques (no crossover operator exists and no speciation is performed in [10]), and c) our focus is mainly on RL tasks rather than time-series prediction tasks. So, at least to the best of our knowledge, such a synergy has not been tried before.

Following [17], we too apply and test both Lamarckian and Darwinian evolution philosophies. In the first case, the adaptable part, W^{out}, is transferred from generation to generation after evolutionary operators are applied to it, while in the second case, it is re-initialized in every generation.

4 METHODOLOGY

Each genome, G, in our methodology consists of the following genes: $W^{in} \in \mathbb{R}^{NK}$, $W \in \mathbb{R}^{N^2}$, $W^{out} \in \mathbb{R}^{N(K+L)}$, $D \in \mathbb{R}$ and $\rho \in \mathbb{R}$.

The initial reservoir has one neuron, $N = 1$, in order to capture the minimum non-linearity (solve the XOR problem). One can also jumpstart the reservoir using more neurons, if necessary. The weights in W^{in} are randomly initialized in $[-1, 1]$, while all W^{out} elements are set to 0. D, is randomly initialized in $[D_{min}, D_{max}]$ and defines the percentage of the active, non-zero, connections in the reservoir, while ρ, is randomly initialized in $[\rho_{min}, \rho_{max}]$. Connections in W are randomly initialized in $[-1, 1]$ based on probability D. A register that keeps a running tally, $r = \sum_i^N \sum_j^N w_{ij}$, of the sum of the reservoir connection weights, helps keep the mean value of weights around 0, during mutation operations. We assume that all reservoir connections exist, but only the non-zero ones are enabled.

After the initialization procedure, each of the organisms in the population passes to the evaluation phase, where learning is enabled and the optimal policy is estimated using the update Equation 5. During this learning-evaluation phase, the performance of each individual is recorded as fitness until the NEAT evolution step takes place (Algorithm 1). First of all, stagnated species, i.e. species that did not improve their fitness on the recent past, are removed. Population is clustered into old or newly created species based on their distance from the species representative and a certain threshold. Species with no attained organisms are removed. Fitness is calculated for both genomes and species and according to that, a proportional number of offsprings is assigned to each of the species for reproduction. During the reproduction phase, for each species, the champion gene is kept as is. The remaining spots are filled by applying mutation, crossover or both, to selected parents that pass a predefined survival threshold, S_{thres}. Interspecies mating is also possible under certain probability, p_i. More details can be found in [11, 12].

Algorithm 1 NEAT evolution step

```
remove-stagnated-species
cluster-population
remove-empty-species
calculate-adjusted-gene-fitness
calculate-species-fitness
calculate-offsprings-per-species
remove-non-assigned-species
for all species do
  keep-champion-gene
  while offsprings-remain do
    x ← select-parent(S_thres)
    if mutate then
      offspring ← mutate(x)
    else
      y ← select-parent(S_thres)
      if random < p_i then
        y ← interspecies-selection(S_thres)
      end if
      if mate then
        offspring ← xover(x,y)
      else
        offspring ← mutate(xover(x,y))
      end if
    end if
    add-offspring(offspring)
  end while
end for
```

4.1 Mutation

The mutation operator pertains to both topology, (adding a node or a connection), and weights (restarting or perturbing weight values). More specifically:

- `mutate-D` and `mutate-`ρ: Given probabilities, p_D and p_ρ, the connection density D and spectral radius ρ are either perturbed, in bounds, by at most 5% at a time, or for a small probability (0.05), restarted.
- `mutate-weights`: Given a probability p_w, each weight in W^{in} and W is mutated, either by perturbing its value $w' = w - sgn(r) \cdot p \cdot w_{pow}$, or by restarting it completely with equal probability, $w' = -sgn(r) \cdot p \cdot w_{pow}$, with w_{pow} being the weight mutation power. For weights in W, r is the running tally and p a random number in $[0, 1]$, while for weights in W^{in}, $r = -1$ and p a random number in $[-1, 1]$. During Lamarckian evolution, output weights are also mutated like W^{in}. Only enabled connections of W are mutated.
- `add-node`: Given a probability p_n, a node is added into the reservoir and all its connections, in and out of this node, are initially disabled by setting them to 0. The weight connection matrices change as follows: $W^{in'} \in \mathbb{R}^{(N+1)K}$, $W' \in \mathbb{R}^{(N+1)^2}$ and $W^{out'} \in \mathbb{R}^{L(K+N+1)}$, with $w_{ij}^{in'} = (2 \cdot p - 1) \cdot w_{pow}, i > N$ and $w'_{ij} = 0, i, j > N$, $w_{ij}^{out'} = 0, j > N$. For the reservoir, the in

and out connection weights of the new node can be found in row $N+1$ and column $N+1$ of W.

- `add-connection`: Given a probability p_c, a connection is enabled and a weight is initialized in the reservoir with respect to the running tally kept by the genome: $w'_{ij} = w_{ij} - sgn(r) \cdot p \cdot w_{pow}$.

4.2 Mating

Following the historical markings technique, every time a new node is added, it is assigned an innovation number equal to its position in the diagonal of W, representing the relative time-step the neuron was added in the history of the specific genome. Then during crossover W, W^{in}, and if Lamarckian evolution is enabled, W^{out}, are aligned based on these numbers. For matrix W there is an example in Figure 2.

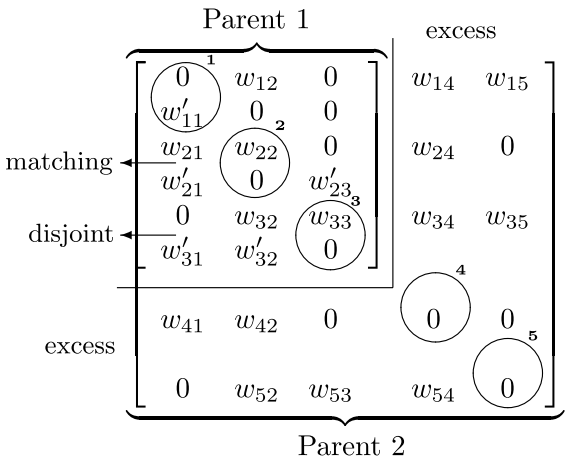

Figure 2. The alignment of two reservoirs of different sizes. The circles virtually represent the nodes, annotated by their innovation numbers at the top right part of each circle.

As a first step, the size of the offspring's reservoir must be determined. There are two alternatives: (i) to continue complexifying the network and choose the largest of the two and (ii) choose the size of the fittest parent. We adopted both approaches by adding a parameter to choose between the two.

Based on the size of the reservoir, excess weights are either completely adopted or completely discarded. For matching and disjoint connections, we follow the rules of Equation 6:

$$w'_{ij} = \begin{cases} \frac{w^1_{ij}+w^2_{ij}}{2} & \text{if } w^1_{ij}, w^2_{ij} \neq 0 \text{ and } p < 0.5 \\ w^1_{ij} & \text{if } w^1_{ij}, w^2_{ij} \neq 0 \text{ and } 0.5 \leq p < 0.75 \\ w^2_{ij} & \text{if } w^1_{ij}, w^2_{ij} \neq 0 \text{ and } 0.75 \leq p < 1.0 \\ w^1_{ij} & \text{if } w^2_{ij} = 0 \text{ and } w^1_{ij} \neq 0 \\ w^2_{ij} & \text{if } w^1_{ij} = 0 \text{ and } w^2_{ij} \neq 0 \end{cases} \quad (6)$$

where p is a random number between 0 and 1.

As for the input and output weights, the excess weights depend on the final value of N, while the rest are either averaged or chosen by either parent with equal probability. D and ρ are inherited from the fittest parent.

4.3 Speciation

For clustering genomes and since W could be quite sparse, it makes little sense to focus on the structural similarities like NEAT does (similarity measure based on matching, disjoint and excess genes), so we implemented a distance measure based on macroscopic features of reservoirs (Equation 7):

$$\delta = \frac{c_\alpha|\rho - \rho_r|}{\rho_r} + \frac{c_\beta|D - D_r|}{D_r} + \frac{c_\gamma|N - N_r|}{N_r} \quad (7)$$

where the r subscript corresponds to the representative of the species and the coefficients c_α, c_β and c_γ are predefined parameters. If $\delta < T$, a prespecified threshold, then the genome is added to the species.

4.4 Remarks and Parameters

Each offspring is verified, before being given for evaluation, so that D is the same as its actual connection density, $\tilde{D}$, after crossover and mutation operators are applied. Given the difference between D and $\tilde{D}$, a probability is calculated as to how many connections should be stochastically added or removed so that $D \approx \tilde{D}$.

We should also note that the genome maintains the matrix W before it is scaled to ρ. This is performed in the procedure of converting the genotype to its phenotype.

A final remark we would like to make with respect to our method and ESNs is that since for several problems $K, L << N$, the big memory expense is on storing W, consuming memory of $O(n^2)$, n being the number of nodes in the network graph. If one enforces reservoirs to be sparse, then matrix W could be stored as a sparse matrix rather than as a full matrix, forming a linear genome and saving computer memory. In that case, the largest eigenvalue used for scaling to ρ could be calculated by the power iteration method. For the present work, since we want to test our methodology in testbeds that do not require extreme memory needs, the $O(n^2)$ approach is implemented.

All the parameters of our method can be found in Table 1, along with their corresponding default values. If not stated explicitly, these are the values used in the experiments.

Table 1. Indicatory values of the parameters in our methodology.

Param.	Value	Param.	Value	Param.	Value
$noise$	10^{-7}	p_ρ	0.05	p_w	0.8
c_α	0.5	p_D	0.05	p_n	0.05
c_β	1.0	p_{mate}	0.33	p_c	0.1
c_γ	1.0	p_{mutate}	0.33	T	1.0
ρ_{max}	0.99	D_{max}	0.99	w_{pow}	0.25
ρ_{min}	0.66	D_{min}	0.01	S_{thres}	0.3
p_i	0.001	Generations	100	Population	100

5 EXPERIMENTAL RESULTS

5.1 Supervised Learning

To evaluate the performance of our methodology, we tested it on a standard time-series benchmark that of predicting the state of the Mackey-Glass (MG) system. From the values found in Table 1, we increased the p_n to 0.5, for the network to complexify faster and reach the appropriate number of features needed for predicting accurately the MG series. Additionally, during crossover, the N value of the largest individual was chosen, instead of the fittest one. We observed that the later approach stuck to initial local minima, not enabling the network to quickly complexify to appropriate sizes. Input and teacher signals were shifted by -1 with no scaling, while the

bias was scaled by 0.2 with no shifting. The $tanh$ sigmoid function was used for the output neurons.

Following the setup in [10], a sequence U of 3000 time-steps was supplied for training with the first 100 time-steps being a wash-out sequence, used to eliminate initial transient effects, and the rest 2900 time-steps used for off-line training. Test and validation Normalized Root Mean Square Errors (NRMSEs) are reported. For the NRMSE$^{\text{test}}$, after the network is trained, U is divided into 20 chunks of 150 time steps. The first 100 time-steps were used as wash-out, while for the second 50 time-steps, on-line prediction was made by connecting the output to the input. The 20 NRMSE$^{\text{test}}$ formed a measure of the fitness: $f = \frac{20}{\sum_{i=1}^{20} NRMSE_i^{test}}$. NRMSE$^{\text{valid}}$ is the error on the prediction of an additional 84 time-steps of the sequence, again by connecting the output to the input after feeding in U.

The median errors of the champion organisms of the 30 runs, were NRMSE$^{\text{test}}$ = $6.29 \cdot 10^{-3}$ and NRMSE$^{\text{valid}}$ = $1.969 \cdot 10^{-3}$. Macroscopic properties of these best performing individuals can be found in Table 3. For comparison, randomly instantiating, training and testing 100 plain ESNs, without the optimization procedure and with $N = 42$ (Table 3), for 30 runs, yielded median NRMSE$^{\text{test}}$ = $2.84 \cdot 10^{-2}$ and median NRMSE$^{\text{valid}}$ = $2.08 \cdot 10^{-2}$. Our test and validation errors reported are better by an order of magnitude than the ones in [10] and from the ones with non-optimized ESNs. Similar behavior to [10] is observed with respect to the macroscopic properties of the reservoirs. A N value of around 40 is enough to get tolerable errors, D is in the high limit, while ρ is close to 1. Also the generalization capabilities of the method are very good, since we get better validation error performance than test error performance.

5.2 Reinforcement Learning

All RL experiments were developed on the RL-Glue platform [16]. The reservoir units continued to have $tanh$ as their activation function, while the output units used the identity function. Moreover, the reservoir size, N, of the fittest parent was maintained in mating, unless for the non-Markovian tasks where the largest N was kept. There was no scaling and shifting of the input and output signals.

5.2.1 Mountain Car

For the Mountain Car (MC) task, we used the setup in [17]. Each of the state components, position x and velocity v, was divided into 10 regions (for a total of 21 inputs, adding a bias). An input of 1 was fed to the corresponding input unit, when the agent fell into a region, 0 otherwise. Each individual in the population was evaluated and improved its policy for 100 episodes (random initial states), with each episode lasting at most 2500 time step, before passing to the evolution stage. The learning rate, α, was set to 0.1 and left to decay to 0 at the end of the 100th episode, ϵ was set to 0.05 and γ to 1. The medians for each of the 10^6 episodes of the 25 runs, were averaged into bins of 1000 episodes, for a total of 1000 bins. Figure 3 depicts the performance of our method for both Darwinian (MC-D) and Lamarckian (MC-L) evolution. In addition, the champion of each run was evaluated for an additional 1000 episodes, starting again from random initial states, with a median average performance of -57.65 time steps for Darwinian and -57.44 for Lamarckian evolution. The statistics of the macroscopic properties of the champion networks are found in Table 3.

Similar performance was achieved to that of NEAT+Q in [17]. The fitness champion of each run gave an average generalization performance of around -57 versus -52 of NEAT+Q, while there was always a generation champion in each run that gave a performance of -48. The asymptotic behavior of Figure 3 was again similar as that of simple NEAT+Q, that is without the use of on-line evolutionary computation.

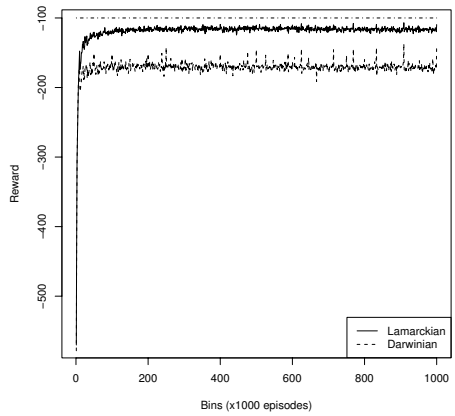

Figure 3. The performance of our methodology for both Darwinian and Lamarckian evolution.

5.2.2 Single and Double Pole Balancing

For this experiment we considered the four setups found in [3]. These are: (a) balancing one pole with complete (SPB-M) and (b) incomplete state information (SPB-NM) and (c) balancing two poles with complete (DPB-M) and (d) incomplete state information (DPB-NM). By incomplete state information, we mean that the state signal is missing the cart and angular velocities of the pole(s), thus the agent needs to maintain a memory of previous states in order to implicitly calculate them. The task is considered solved if the pole or poles are balanced for $100,000$ time-steps, equal to over 30 minutes of simulated time. The learning rate, α, started at 10^{-6} and the exploration, ϵ, started at 0.01 and decayed to 0 at the end of the 100th episode for each organism, while γ was set to 1.0. Inputs were normalized between -1 and 1. In Tables 2 and 3 we find the results of our experiments.

Table 2. The average (AVG) and standard deviation (SD) over the 50 runs of the number of generations, networks and episodes needed to find a solution.

Task	Generations		Networks		Episodes	
	AVG	SD	AVG	SD	AVG ($\times 10^3$)	SD ($\times 10^3$)
SPB-M	2.7	0.9	232.3	99.7	17.87	9.78
SPB-NM	46.2	20.5	4666.2	2041.2	456.59	204.15
DPB-M	6	2.1	628.9	211.7	52.89	21.17
DPB-NM	22.4	11.6	2287.9	1171.3	218.79	117.13

Our methodology was able to find solutions to all four tasks, on all runs. The results are not directly comparable to those in [3], since we did not perform direct policy search, but have included a learning step in between. So, as far as the number of networks evaluated (except the case of SPB-NM) is concerned, our approach is comparable to the highest performing ones in [3]. On the other hand, even though the runs ended successfully, the number of evaluation episodes is quite large. One could say that the number of episodes each individual

was evaluated was preset to a larger than required number. As an improvement, a more directed search could be applied, evaluating the highest performing individuals on more episodes and the lower level ones on fewer episodes [17]. For the SPB-NM case, the method found solutions in all runs, but failed to do so in a prompt manner. This is probably due to fact that no parameter optimization was made to the values of Table 1. Alternatively, one could initially set N to a higher value.

Table 3. Statistics of the macroscopic parameters of the best networks for all testbeds.

Task (runs)	N		D		ρ	
	AVG	SD	AVG	SD	AVG	SD
MG (30)	42.90	10.18	0.86	0.21	0.92	0.06
MC-L (25)	2.08	0.86	0.52	0.28	0.81	0.10
MC-D (25)	2.52	1.08	0.59	0.27	0.79	0.09
SPB-M (50)	1.22	0.41	0.55	0.30	0.80	0.09
SPB-NM (50)	4.24	1.94	0.58	0.26	0.8	0.09
DPB-M (50)	1.98	1.05	0.52	0.27	0.85	0.09
DPB-NM (50)	6.44	2.99	0.67	0.26	0.85	0.07

The number of nodes in the network for RL problems was relevant to the difficulty of the problem at hand. Reservoir densities had the tendency of being around values of 0.5 or larger. Based on these observations one could note that by using an evolutionary search algorithm solutions can be found that contain a small number of neurons and a large number of reservoir connections, despite the fact that it is a good practice to have large, sparse, networks.

Three variations of our methodology were also tested: (1) a population of non-evolved randomly instantiated ESNs, with N following the values of Table 3, trained using simple GD, (2) a simple linear network (no reservoir of neurons), matching a linear approximator trained using simple GD and weight mutation and (3) the complete methodology, using only weight mutation and not TD-learning, under Lamarckian evolution.

From the results in Table 4 it is evident an ESN performs better if its topology and weights are optimized through an evolutionary procedure. Non-optimized ESNs with simple GD were not capable of solving any of the pole balancing tasks. On the other hand, a linear FA with GD and weight mutation performed better than variant (1), but, as expected, was not able to solve the non-Markovian tasks. Finally, if GD is substituted with just weight mutation, in some cases even better performance could be obtained. This is an indication that our methodology could be further improved by substituting the simple GD with the more sophisticated evolution strategies or least squares TD-learning.

Table 4. The performance of three variants in terms of the median generalization performance of the champion of each run in the MC testbed and the number of networks evaluated for solving the task in the pole balancing testbeds. A dash is used when no solutions are found.

Variant	MC	SPB-M	SPB-NM	DPB-M	DPB-NM
(1) Simple ESN	-174.51	-	-	-	-
(2) No reservoir	-57.08	339.66	-	561.5	-
(3) No learning	-50.84	368.78	7107.66	389.36	1809.84

6 CONCLUSIONS AND FUTURE WORK

In this paper, we presented our work of adapting ESNs through evolution and learning to the task at hand. For our evolutionary procedure, we applied the NEAT method as a meta-search evolutionary algorithm and created operators adapted to the specifics of ESNs. The results obtained are promising since the coalition proved highly robust and efficient in different benchmarks.

This work is our first step into developing a robust and automated way for adapting ESNs for RL problems in an ad-hoc manner. We look forward into adopting evolution strategies for evolving specific parameters like the D and ρ. Additionally, operator application probabilities could become adaptable and intrinsic plasticity could be introduced like in [10]. Finally, we plan to test our methodology into more exciting testbeds, like for trading and game playing agents.

REFERENCES

[1] Keith Bush and Batsukh Tsendjav, 'Improving the richness of echo state features using next ascent local search', in *Proceedings of the Artificial Neural Networks in Engineering Conference*, pp. 227–232, St. Louis, MO, (2005).

[2] Alexandre Devert, Nicolas Bredeche, and Marc Schoenauer, 'Unsupervised learning of echo state netowrks: a case study in artificial embryogeny', in *Proceedings of the 8th International Conference on Artificial Evolution*, volume 4926 of *LNCS*, pp. 278–290. Springer, (2008).

[3] Faustino Gomez, Jürgen Schmidhuber, and Risto Miikkulainen, 'Efficient non-linear control through neuroevolution', in *Proceedings of the European Conference on Machine Learning (ECML 2006)*, volume 4212/2006 of *Lecture Notes in Computer Science*, pp. 654–662. Springer Berlin / Heidelberg, (2006).

[4] Kazuo Ishii, Tijn van der Zant, Vlatko Bečanović, and Paul Plöger, 'Identification of motion with echo state network', in *Proceedings of the OCEANS 2004 MTS/IEEE - TECHNO-OCEAN 2004 Conference*, volume 3, pp. 1205–1210, (2004).

[5] Herbert Jaeger, 'The ''echo state'' approach to analysing and training recurrent neural networks - with an erratum note', Technical Report GMD Report 148, German National Research Center for Information Technology, (2001).

[6] Herbert Jaeger, 'Tutorial on training recurrent neural networks, covering BPTT, RTRL, EKF and the ''echo state network'' approach', Technical Report GMD Report 159, German National Research Center for Information Technology, (2002).

[7] Fei Jiang, Hugues Berry, and Marc Schoenauer, 'Supervised and evolutionary learning of echo state networks', in *Proceedings of 10th International Conference on Parallel Problem Solving from Nature, PPSN 2008*, volume 5199 of *LNCS*, pp. 215–224. Springer-Verlag, (2008).

[8] Mantas Lukosevicius and Herbert Jaeger, 'Reservoir computing approaches to recurrent neural network training', *Computer Science Review*, **3**, 127–149, (2009).

[9] N. J. Radcliffe, 'Genetic set recombination and its application to neural network topology optimization', *Neural computing and applications*, **1**(1), 67–90, (1993).

[10] Benjamin Roeschies and Christian Igel, 'Structure optimization of reservoir networks', *Logic Journal of IGPL*, (2009).

[11] Kenneth O. Stanley, *Efficient Evolution of Neural Networks*, Ph.D. dissertation, University of Texas at Austin, 2004.

[12] Kenneth O. Stanley and Risto Miikkulainen, 'Evolving neural networks through augmenting topologies', *Evolutionary Computation*, **10**(2), 99–127, (2002).

[13] Peter Stone, 'Learning and multiagent reasoning for autonomous agents', in *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pp. 13–30, (January 2007).

[14] Richard S. Sutton and Andrew G. Barto, *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA, 1998.

[15] István Szita, Viktor Gyenes, and András Lőrincz, 'Reinforcement learning with echo state networks', in *Artificial Neural Networks ICANN 2006*, volume 4131/2006 of *Lecture Notes in Computer Science*, pp. 830–839. Springer Berlin / Heidelberg, (2006).

[16] Brian Tanner and Adam White, 'Rl-glue: Language-independent software for reinforcement-learning experiments', *Journal of Machine Learning Research*, **10**, 2133–2136, (2009).

[17] Shimon Whiteson and Peter Stone, 'Evolutionary function approximation for reinforcement learning', *Journal of Machine Learning Research*, **7**, 877–917, (May 2006).