Neurocomputing 103 (2013) 198–209

Contents lists available at SciVerse ScienceDirect

Neurocomputing

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

Adaptive reservoir computing through evolution and learning

Kyriakos C. Chatzidimitriou*, Pericles A. Mitkas

Department of Electrical and Computer Engineering, Aristotle University of Thessaloniki, Thessaloniki 54124, Greece

ARTICLE INFO

Article history:
Received 22 March 2012
Received in revised form
6 August 2012
Accepted 18 September 2012
Communicated by A. Belatreche
Available online 23 October 2012

Keywords:
Reservoir computing
Echo state networks
Neuroevolution
Evolutionary computation
Reinforcement learning

ABSTRACT

The development of real-world, fully autonomous agents would require mechanisms that would offer generalization capabilities from experience, suitable for a large range of machine learning tasks, like those from the areas of supervised and reinforcement learning. Such capacities could be offered by parametric function approximators that could either model the environment or the agent's policy. To promote autonomy, these structures should be adapted to the problem at hand with no or little human expert input. Towards this goal, we propose an adaptive function approximator method for developing appropriate neural networks in the form of reservoir computing systems through evolution and learning. Our *neuro-evolution of augmenting reservoirs* approach comprises of several ideas, successful on their own, in an effort to develop an algorithm that could handle a large range of problems, more efficiently. In particular, we use the neuro-evolution of augmented topologies algorithm as a meta-search method for the adaptation of echo state networks for handling problems to be encountered by autonomous entities. We test our approach on several test-beds from the realms of time series prediction and reinforcement learning. We compare our methodology against similar state-of-the-art algorithms with promising results.

© 2012 Elsevier B.V. All rights reserved.

1. Introduction

The "holy grail" of artificial intelligence (AI) is to create fully functional autonomous agents, embodied or not, that would coexist with their counterparts and humans in the real world [1]. Expressed in another way, the goal is to build better intelligent systems [2]. In line with the target above, Stone [1] advocated that one of the most appropriate schemes for creating autonomous artificial entities is through the *reinforcement learning* (RL) paradigm [3]. In addition, experimental analysis suggests that mammalian brains employ learning mechanisms developing behaviors similar to those produced by RL [4]. An autonomous agent developed under the RL framework is oriented towards maximizing the total amount of reward it receives over time, in a trial-and-error fashion, by constantly adapting its policy, i.e. the mapping of state to actions. Adaptation, based on immediate feedback returned by interacting with the environment, will favor actions that maximize long-term expected reward. Moreover, the mechanism should work without ever being taught with examples of correct behavior, as in supervised learning.

For simple RL problems, the policy can be represented in an array form. For complex, real-world tasks though, with large, continuous state and action spaces, there is the need for a *function approximator* (FA) to represent the policy. The FA will offer generalization capabilities by providing the expected return (value) to state–action pairs the agent has never encountered before. FAs are usually borrowed from the supervised learning domain and may take the form of cerebellar model articulator controllers, radial basis functions, neural networks and decision trees among others [3]. High performance of the FA necessitates appropriate choices with respect to the architecture and the parameters of the FA. Against this background and having in mind that the purpose of Computer Science is to automate tasks and provide general problem solving algorithms, the area of *adaptive function approximation* was developed [1]. The goal of the area is to adapt the architecture and the parameters of the FA to the problem at hand, with little or no human input. This is performed through the synergy of learning and evolution. Evolution (the global, stochastic search part) is applied in order to find FAs that are better able to learn (the local, gradient descent search part) [5].

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 main goal is to create an efficient and robust adaptive function approximator that works well across several domains. Our methodology comprises three components: (i) a FA, (ii) a neuro-evolution (NE) method, and (iii) the coupling of learning and evolution. The paper continuous as follows: Section 2 introduces the components involved. Section 3 provides an account of the closest matches of related work. Section 4

* Corresponding author. Tel.: +30 2310996349; fax: +30 2310996398.
E-mail addresses: kyrcha@issel.ee.auth.gr (K.C. Chatzidimitriou), mitkas@auth.gr (P.A. Mitkas).

0925-2312/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.neucom.2012.09.022

describes the developed methodology. Section 5 presents the domains, the experimental setup, the results of the experiments and comments on them. Finally, Section 6 summarizes the paper, presents our conclusions and discusses future improvements.

2. Background

In this section we provide a brief overview of the components involved in the construction of our adaptive function approximation methodology, which are the *echo state network* (ESN), *neuroevolution*, and more specifically, the *neuroevolution of augmented topologies* (NEAT) methodology and the coupling of *learning and evolution*. The goal of this section is to introduce the reader to the algorithms, concepts and notation used in the rest of the paper.

2.1. Echo state networks

For the first component, our FA of choice is the *echo state network* (ESN), from the area of reservoir computing (RC). The ESN [6,7] seems like an appropriate solution due to its properties inherited from its recurrent neural network (RNN) profile. RNNs are able to (a) capture any existing non-linear dynamics of the environment and (b) model any existing non-Markovian state signals. In addition, difficulties with bifurcations, training, long-range memory and implementation, often cited as shortcomings of classic RNNs, are replaced with ESN advantages such as modeling accuracy and capacity, biological plausibility, extensibility and parsimony [8]. Moreover, even though ESNs can handle non-linear environments, learning can be applied in an on-line fashion using linear learning rules to the output layer of the network. On the other hand, creating reservoirs at random, as initially proposed, seems unsatisfactory and the need for a specific reservoir design, adapted to the task, is conjectured to yield better results [9,8]. Against this background, the ESNs make for a powerful FA and an appropriate candidate for a dual adaptation through both learning (output layer) and evolution (reservoir topology).

A basic form of an ESN is presented in Fig. 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 backprojection 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 incorporate $f(x) = \tanh(x)$ as their activation function, while the output units use either the sigmoid functions, $g_1(x) = \tanh(x)$ and $g_2(x) = 1/(1+e^{-x})$ or the identity function, $g_3(x) = x$, according to the problem at hand. In this work, we consider discrete time models and ESNs without backprojection connections. The absence of matrix W^{back} does not inhibit the capability of the networks to form recurrent connections, since these are formed in the reservoir W, as seen also in Fig. 1.

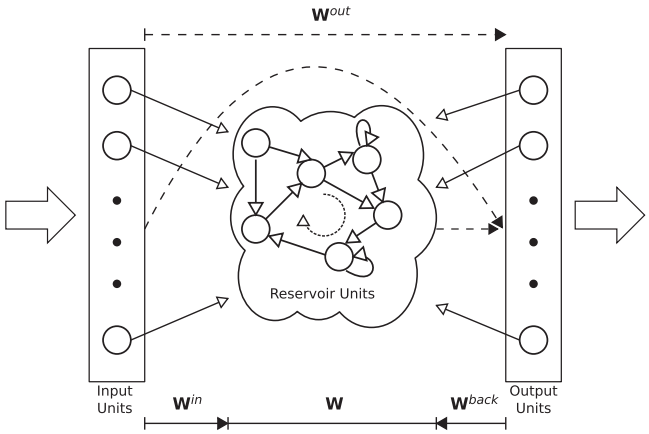

Fig. 1. A basic form of an ESN. Solid arrows represent fixed weights and dashed arrows adaptable weights. Connections can also form cycles in the reservoir for performing temporal calculations.

Best practices for generating ESNs, i.e. procedures for generating the random connection matrices W^{in},W and W^{back}, can be found in [7,8]. 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.

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]. As for RL tasks, linear gradient descent (GD) SARSA temporal difference learning [3,10] or least squares temporal difference learning (LSTD) [11,12] can be used in order to adapt the output weights. Last but not least, policy search could also be applied in order to find the appropriate $\mathbf{W}^{out}$ matrix, through for example evolutionary computation.

2.2. NeuroEvolution

In general, evolutionary computation (EC) algorithms [13] are stochastic search, population-based algorithms that are developed from ideas and principles of natural evolution, designed to handle efficiently multiple local minima, since they are less likely to trap than gradient descent based algorithms, plus are capable of dealing with problems where no explicit and/or exact objective function is available [2]. EC algorithms can – in principle – handle optimization problems that are vast, complex, non-differentiable, non-continuous, multimodal, and noisy [2,13,14]. *Neuro-evolution* (NE) is an area of EC, focusing on algorithms that try to optimize one or more characteristics of NNs like the weights, the topology, the meta-parameters and so on. Detailed accounts of NE methods can be found in [2,4,15].

2.2.1. NeuroEvolution of augmented topologies

Despite their successes and recognition, NE methods have shortcomings. The more eminent problems are those of *competing conventions* [16] and *premature convergence* [4]. In the competing conventions problem, the knowledge in a NN is distributed among all the weights. Thus combining one part of a NN with another part of another NN is likely to destroy both NNs [2]. Due to the competing conventions problem, crossover is usually not applied in NE, although it has been shown that it may be useful and important in increasing the efficiency of evolution for some problems [2]. In premature convergence, difficult fitness landscapes with local optima can cause a rapid decrease of the population diversity and thus render the search inefficient [4]. Against this background, we are using neuro-evolution of augmented topologies (NEAT) as our meta-search evolutionary framework for the NE part of our algorithm of evolving ESNs. NEAT [15,17], a state-of-the-art NE method that is capable of topology and weight evolution of artificial neural networks (TWEANN). The reason is that NEAT has suggested procedures that help alleviate the problems mentioned above. For this reason, it makes for a very good candidate for adapting it and using it to evolve ESNs. In particular, we have transformed the major ideas behind the NEAT approach: (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.

Algorithm 1 displays the procedure used by NEAT. After the initialization procedure, the evolutionary phase begins. Each organism in the population passes the evaluation phase. The performance of each individual is recorded as fitness until the evolution step takes place. First of all, stagnated species, i.e. species that did not improve their fitness during over the past $Stagnation_{max}$ generations, are removed. Population is clustered into old or newly created species based on their distance from the species representative and a certain threshold. The species representative is the first individual to form a new species. Species with no attained organisms are removed. Fitness is calculated for both genomes and species and according to that, a proportional number of offspring is assigned to each of the species for reproduction. During the reproduction phase, for each species, the champion gene is kept unaltered. 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.

Algorithm 1. The NEAT algorithm as seen from a meta-search point of view. No specific implementation details are included for the evolutionary operators.

```
p → initialize-population
for all generations do
  for all genomes do
    network → phenotype(genome)
    fitness → evaluate(network)
    genome → fitness
  end for
  remove-stagnated-species(Stagnation_max)
  cluster-population
  remove-empty-species
  calculate-adjusted-gene-fitness
  calculate-species-fitness
  calculate-offspring-per-species
  remove-non-assigned-species
  for all species do
    keep-champion-gene
    while offspring-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
end for
```

2.2.2. Learning and evolution

Despite the advantages of NE over classical learning schemes, one can couple NE and learning (supervised or reinforcement) in order to have a complete adaptation as the search is conducted both in global (evolution, population based stochastic search) and local (learning through gradient descent, hill-climbing or other local search algorithms) terms. This coalition is often referred to as memetic computing [13] or hybrid training [2]. Through this combination, we try to evolve networks that are better able to learn making the fitness landscape smoother and discover nearby good networks that could not be easily identifiable otherwise [18]. In addition, it would be desirable to include the *Baldwin effect* [19] in the evolutionary process through this interaction [20], so that desirable properties developed through learning can be incorporated to the evolution process. In general, hybrid algorithms tend to perform better than others for a large number of problems [2], while according to the no-free-lunch theorem [21], the best training algorithm is always problem dependent. Thus, our goal is not to create a state-of-the-art algorithm for certain problems but rather a robust algorithm that is able to handle a large range of problems efficiently. We test the benefits of such a synergy experimentally, and investigate the difference in opinions on the value of hybrid algorithms, along with the debate over which EC model, Lamarckian or Darwinian, suits us the best.

The basic argument for choosing to use both learning and evolution for the adaptation of the FA is the complementarity of the two methods. Structure plays a crucial role for the learning behavior of either biological or artificial neural systems [14]. Hence, it appears consequential to apply evolutionary methods for the structural adaptation of neural systems, especially in the absence of efficient "classical" methods. Furthermore, evolutionary methods are easier to apply with respect to optimizing meta-parameters (for example learning rates, network densities, etc.) or FAs that are non-differentiable [22]. On the other hand, this direct search through evolutionary optimization in the space of policies can be like searching for a needle in a haystack [18]. The ESN provides for a perfect candidate for adaptation using both evolution and learning. If EC identifies a basin of attraction where the global minimum is, then local search can locate the global minimum [2]. As far as ESNs are concerned, the existence of multiple local minima has been previously established [23]. In addition, evolutionary search methods could handle the optimization of meta-parameters, such as the density or the spectral radius of the reservoir. On the other hand, in favor of learning, is the differentiability of the ESN output layer, which allows for the use of "classic" linear methods in order to make local changes towards the local minima, keeping evolutionary search away from wandering around. In other words, evolution will search for structures that enable a better learning towards the local minima, and altogether towards the global minimum. The concept is illustrated in Fig. 2.

3. Related work

ESNs have been used before as FAs for RL tasks by [10], but in their basic, non-optimized form. In the same paper, the authors show that ESNs using the SARSA(λ) algorithm exhibit the same convergence behavior as a linear FA with SARSA(λ), with the extra capability of holding for k-order Markov decision processes (MDPs) as well. This result makes ESNs more theoretically desirable as FAs for RL problems.

When it became evident that better results could be obtained by optimizing the reservoir to exhibit rich dynamics, instead of randomly generating it, many methods came up to do exactly that [24–28], mainly evolutionary, but with the network size pre-specified or optimized separately from the topology and weights. From the beginning of the development of our methodology, the

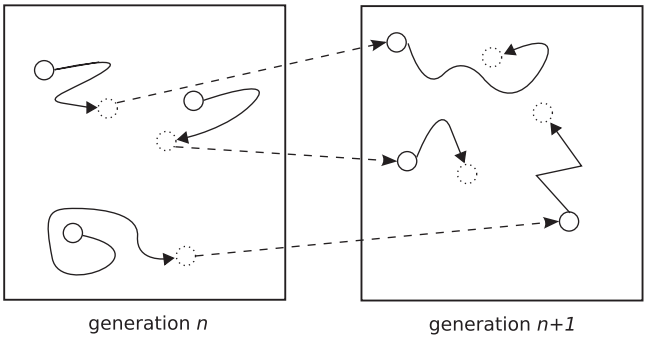

Fig. 2. Solid line circles represent the population of networks of the NE method spread around the search space. Through learning, the networks are adapted towards an area of lower error under, for example, gradient descent (dashed circles). Through evolution (Lamarckian in this particular case), they are moved to another location in the search space, where, possibly, the neighborhood will contain a lower local minimum for learning to find.

goal was to have open ended evolution and to create networks that will adapt to the problem at hand, autonomously, without knowing the capacity, in terms of neurons, required to handle the complexity of the problem. This is why we have selected NEAT as our search algorithm of choice. The closest matches to our method can be found in [5,29]. In [5], NEAT is used, as is, to develop ad hoc neural networks without recurrent connections so that TD-learning could be applied using standard error backpropagation updates to further adapt weights towards a solution. In [29] 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, together, using NE and TD-learning. This differentiates our focus from any work that optimizes only macroscopic reservoir parameters. The differences from [5], apart from the obvious use of ESNs versus ad hoc neural networks, are that (a) ESNs maintain a memory and thus solve tasks with non-Markovian state signals, and (b) the network is trained using simple linear least squares or TD-learning, instead of error backpropagation, providing theoretical and algorithmic advantages derived by the simplicity of the linear combination of reservoir features. Finally, this work differs from [29] 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 [29]), 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 examined before.

4. NeuroEvolution of augmented reservoirs

In this section we present the *neuro-evolution of augmented reservoirs* (NEAR), an algorithm that uses NEAT as a meta-search algorithm, modified according to the particularities of ESNs, which are adapted through evolution and learning.

4.1. Genome

Each genome, G, in our methodology consists of the following genes: (1) $W^{in} \in \mathbb{R}^{NK}$, the input weights from input nodes to the reservoir nodes, (2) $W \in \mathbb{R}^{N^2}$, the matrix defining the reservoir topology and its connection weights, (3) $W^{out} \in \mathbb{R}^{N(K+L)}$, the output weights from input and reservoir units to the output units, (4) $D \in [0,1]$, the density of the reservoir, and (5) $\rho \in \mathbb{R}$, the spectral radius by which the reservoir weights will be normalized.

As implied from the above list, direct encoding was used. There is a one-to-one mapping between the parameters of the network and the genetic encoding of each gene. We should note though that the genome maintains the matrix W before it is scaled by using its largest eigenvalue and ρ. This is performed in the procedure of converting the genotype to its phenotype. Direct encoding is straightforward to implement [2], so we adopted this approach, since the domains under consideration will not allow for the genome to explode in space requirements ($O(N^2)$, since $N \gg K,L$). A linear genome (like in NEAT) could be examined if we used a sparse representation of the reservoir W,[1] based on the good practice of creating sparse reservoirs. On the other hand, we wanted to test whether dense reservoirs are also capable of driving high performance networks as discussed in [29]. For these two reasons a direct encoding scheme with $O(N^2)$ space requirements was adopted.

The initial reservoir has at least one neuron, $N = N^{initial} = 1$, in order to capture the minimum non-linearity (that is solve the XOR problem). The weights in W^{in} and W^{out} are randomly initialized in $[-1,1]$. D is randomly initialized in [0.05, 1.0] and defines the percentage of the active, non-zero, connections in the reservoir, while ρ is randomly initialized in [0.55, 0.99]. 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 maintain 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.

4.2. TD learning and policy search

Following the initialization procedure, each of the organisms in the population is converted to an ESN and passes to the evaluation phase (Algorithm 1). The network is evaluated for a number of episodes and is adapted using either TD learning or a policy search algorithm. In this work, policy search was implemented as standard weight mutation in the W^{out} matrix. The performance of the individual is recorded as fitness, and upon the completion of the evaluation of all the population the evolution step takes place (Algorithm 1).

Following [5], we too apply and test both Lamarckian and Darwinian evolution philosophies, when the learning component is enabled. 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. Even though Lamarckian evolution seems like the method of choice, learning could just guide evolution to select the genome for reproduction that is better (for example learns better and faster), even though the learned weights are not transferred (Darwinian evolution). This is also a case of the Baldwin effect.

4.3. 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,

[1] In this case the power iteration algorithm could be used to find the largest absolute eigenvalue when converting from the genotype to the phenotype.

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 $w' = w - \text{sgn}(r)\cdot p \cdot w_{pow}$, or by restarting it completely with equal probability, $w' = -\text{sgn}(r)\cdot p \cdot w_{pow}$, 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 probability p_n defines the rate of increase of the genomes sizes (networks) in the population. 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 $w^{in'}_{ij} = (2\cdot p - 1)\cdot w_{pow}, i > N$ and $w'_{ij} = 0, i,j > N$, $w^{out'}_{ij} = 0, j > N$. 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: $w'_{ij} = w_{ij} - \text{sgn}(r)\cdot p \cdot w_{pow}$.

4.4. Mating

Mating in the form of crossover is a major feature of both NEAT and NEAR. Even though there are arguments against its use in NE, [30] have calculated that the problem is not as severe as initially conjectured. In addition, [31] have provided extensive experiments on the robustness of solutions through mating to the behavior of evolutionary algorithms. 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. An example for matrix W is given in Fig. 3.

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) to choose the size of the fittest parent. We implemented both approaches by adding a parameter to choose between the two.

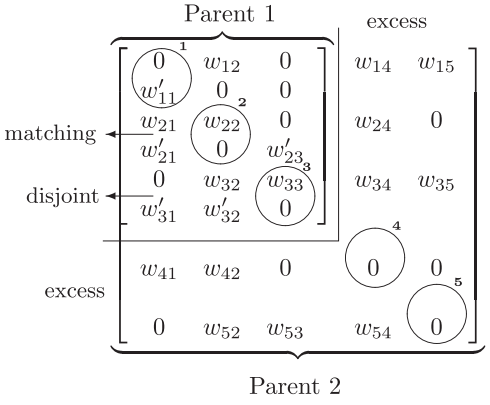

Fig. 3. The alignment of two reservoirs of different sizes. The circles virtually represent the nodes, annotated by their innovation numbers at the top right part of each circle.

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 Eq. (1):

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

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 chosen parent.

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 $D \approx \tilde{D}$.

4.4.1. Crossover adaptation

When crossover takes place in the NEAT methodology, the fittest of the two individuals is chosen with respect to retaining or throwing away excess neurons. During the development of the NEAR methodology, we have observed that always retaining the number of nodes of the fittest individual could lead to stagnation in the improvement of the fitness function. In order to resolve this problem, we have added a probability that chooses the strategy to select, i.e. select as a placeholder the fittest or the largest parent. By choosing the largest parent, the procedure could escape local minima and plateaux, by moving the individual further away from that area of the search space.

The efficiency of an EC methodology, and in our case NEAR, can be improved by the on-line adaptation of the evolutionary strategy parameters [5,29]. Thus, in order to optimize the strategy of selection, we adapt the probability of choosing, a strategy also used in [29], where the mutation operator probabilities are adjusted on-line based on the operator's performance in the specific problem.

Following the notation of [29], we have the set of possible crossover operators $\Omega = \{fittest, largest\}$ and $p^{(k)}_1, p^{(k)}_2 = 1 - p^{(k)}_1$ the probabilities of choosing either the *fittest* or the *largest* crossover operator, respectively, in generation k. Let $O_i(k)$ with $i = 1,2$ denote the offspring created by the application of the corresponding operator. The mean quality of each operator is given by

$$q^{(k)}_i = \frac{1}{|O_i(k)|} \sum_{g \in O_i(k)} \max[0, \Phi(g) - \max[\Phi(par_1(g)), \Phi(par_2(g))]] \quad (2)$$

where $par(g)$ denotes the parent of individual g and Φ the fitness function. The operator probabilities are found by making the following calculations, first:

$$s^{k+1}_i = \begin{cases} \frac{c_\Omega \cdot q^{(k)}_i}{q^{(k)}_{all}} + (1 - c_\Omega)\cdot s^k_i & \text{if } q^{(k)}_{all} > 0 \\ \frac{c_\Omega}{|\Omega|} + (1 - c_\Omega)\cdot s^k_i & \text{else} \end{cases} \quad (3)$$

and later:

$$p^{(k+1)}_i = p_{min} + \frac{(1 - |\Omega|\cdot p_{min}) s^{k+1}_i}{\sum_{i' \in \Omega} s^{k+1}_{i'}} \quad (4)$$

with $q^{(k)}_{all} = \sum_{i' \in \Omega} q^{k+1}_{i'}$, $c_\Omega \in (0,1]$ a parameter controlling the decrease and p_{min} a bound on the probability. We selected $c_\Omega = 0.5$ and $p_{min} = 0.1$, while $s^{(0)}_1 = s^{(0)}_2 = 1$.

Table 1
Similarities and differences between NEAT and NEAR.

Property	NEAT	NEAR
TWEANN	√	√
Non-linear	√	√
Linear	×	√
Recursions	√	√
Speciation	√ (On micro parameters)	√ (On macro parameters)
Crossover	√ (On links—Fittest)	√ (On nodes—Adaptive)
Open-ended	√	√
Incremental	√ (On nodes)	√ (On nodes)
$N^{initial}$	0	1
Learning	√ (Hebbian)	√ (Any)
Structure	Ad hoc	ESN

A simpler alternative to consider is to randomly choose between the largest or the fittest individual with equal probability.

4.5. 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:

$$\delta = \frac{c_\alpha|\rho-\rho_r|}{\rho_r} + \frac{c_\beta|D-D_r|}{D_r} + \frac{c_\gamma|N-N_r|}{N_r} \quad (5)$$

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 pre-specified threshold, then the genome is added to the species.

4.6. NEAT and NEAR comparison

Table 1 presents the similarities and differences between NEAR and NEAT. Their main differences are (a) that NEAR can be coupled with any kind of learning algorithm, (b) the linear features present in NEAR and ESNs by having the inputs connecting directly to the outputs, and (c) the kind of networks they evolve. NEAR evolves strictly ESNs carrying all their background, theory, advantages and disadvantages.

The values of the parameters presented in this section can be found in Appendix A. The number of the parameters is approximately equal to that of NEAT and NEAT+Q. Even though the list of parameters might seem long, their default values, set intuitively, are generic enough to provide state-of-the-art behavior as we will see in Section 5.

5. Experiments and results

In this section, we present the domains, the experimental setups and the results in order to evaluate the NEAR methodology and study its behavior. Even though our main focus is on RL problems, we tested NEAR also in time series prediction problems, under a supervised learning scheme, in order for our evaluation to be more complete. NEAR was compared against related algorithms and versions of itself, either through direct or bibliographic comparison, both in time series and RL test-beds. The implementation of the ESN, NEAR and NEAT algorithms along with the test-beds and the scripts for running the experiments can be found in the *fiterr* project at http://kyrcha.info/projects/fiterr. RL test-beds and experiments were orchestrated using the RL-Glue platform [32].

5.1. Time series

Three time series prediction problems were chosen to test the NEAR methodology. For testing NEAR on time series, we have used the exact same experimental setup and notation as that found in [29], making the results reported directly comparable.

To evaluate the performance of the networks produced after the evolutionary process takes place, i.e. the champion networks, the normalized root mean square error ($NRMSE^{val}$) is calculated on a previously unseen part of the time series, the validation sequence. The train sequence is used as a wash-out sequence and after that the network is put in recursive mode, using the output it produces as input for the next time step in order to predict the validation series values. During the evolutionary phase, calculating fitness based on the error in the validation sequence would constitute reporting on that error an underestimation of the true generalization error, hence fitness is calculated on parts of the training sequence, the test sequences. The fitness of each individual is given by

$$\Phi(g) = \frac{1}{NRMSE_g^{test}} \quad (6)$$

where $NRMSE^{test}$ is the normalized root mean square error for the test sequences.

The three time series we are using as testbeds are: (a) the Mackey–Glass time series, (b) the multiple superimposed oscillator (MSO), and (c) the Lorentz attractor time series and in particular the x-coordinate of the system.

For fair comparison purposes, we jump-started the initial reservoir size of the NEAR population to include 25 hidden neurons. The method in [29] starts with 30 neurons but is capable of deleting nodes as well. Both methods use a population of 50 individuals and evolutionary steps were performed over 100 generations. Results are averaged over 30 trials. In addition, we examined the performance of a standard method for searching high quality reservoirs, that of randomly initializing a number of ESNs and then keeping the best one. Two such searches are presented, one with 50 reservoir neurons (ESN-50) and a larger one with 100 (ESN-100). Each random search tested 100 such ESNs. The best performing network is chosen according to the test error and not according to the final validation error, calculated on unseen data, in order to present a more accurate generalization error estimate in the end, that of the validation error.

5.1.1. Results

For all the time series problems, results for $NRMSE^{test}$ and $NRMSE^{val}$ are reported in Table 2. NEAR searches for an optimal topology, while least squares learning adjusts the output weights to their optimal values on the training data-set. For the Mackey–Glass problem, NEAR+LS method (LS for least squares) outperforms the ESN-100 method even though NEAR discovers networks with a reservoir of around 35 neurons. On the other hand, NEAR+MUT (using mutations to find the optimal output weights) was not able to learn a useful model of the time-series. For the MSO time series problem, the NEAR algorithm was not capable of adjusting the ESN to learn the dynamics of this time series. This is probably due to the shortcoming of ESNs to support prediction of superimposed generators, due to strong coupling between the reservoir neurons [8]. Special care must be taken for this case as we will discuss in Section 6. As for the Lorentz attractor time series problem, NEAR+LS outperforms every other approach with respect to the validation error. In addition it appears to generalize well, as indicated by the lower validation error with respect to $NRMSE^{test}$.

Table 2
$NRMSE^{test}$ and $NRMSE^{val}$ errors for the Mackey–Glass, Multiple-Sinewave Oscillator and Lorentz time series prediction problems. Algorithm 1 is NEAR+LS, 2 is NEAR+MUT, 3 is ESN-50 and finally 4 is ESN-100.

Alg.	Mackey–Glass		MSO		Lorentz	
	Test	Val	Test	Val	Test	Val
1	5.4×10^{-3}	$\mathbf{2.9\times10^{-3}}$	9.3×10^{-1}	9.4×10^{-1}	4.4×10^{-2}	$\mathbf{7.9}\times10^{-3}$
2	8.1×10^{-1}	9.6×10^{-1}	9.9×10^{-1}	9.8×10^{-1}	8.8×10^{-1}	4.4×10^{-1}
[29]	3.6×10^{-2}	6.5×10^{-2}	3.5×10^{-3}	$\mathbf{8.9\times10^{-3}}$	2.1×10^{-2}	1.2×10^{-2}
3	1.0×10^{-2}	1.0×10^{-2}	9.8×10^{-1}	9.8×10^{-1}	2.6×10^{-1}	4.0×10^{-2}
4	1.6×10^{-3}	3.8×10^{-3}	9.7×10^{-1}	9.8×10^{-1}	5.6×10^{-2}	4.0×10^{-2}

5.2. Reinforcement learning

Three RL tasks were considered in this work: (a) different scenarios of the mountain car task, (b) different versions of the pole balancing task, and (c) the server job scheduling task from the autonomic computing area. For this line of experiments, we evaluated the following algorithms: (a) NEAT, (b) NEAR with gradient descent TD learning and Lamarkian evolution (NEAR+TD-L), (c) with Darwinian evolution (NEAR+TD-D), (d) NEAR with policy search through standard weight mutation in the output matrix (NEAR+PS) and (e) multiple static ESNs with gradient descent temporal difference learning (ESN+TD). Also results reported in the literature [5,33] are compared against our approach.

5.2.1. Mountain car

For the mountain car (MC) task, we use the version by [34]. We call this version the 2D MC task. The 3D MC extends the 2D task by adding an extra spatial dimension. The 3D task was originally proposed in [35]. Along with the aforementioned versions, one more MC version was tested, which we will call the non-Markovian (NM) one, since the speed variable or variables in the 3D case, are hidden from the agent, and only its position or positions in the 3D case, are revealed, making the task more challenging. Each episode lasts 2500 time steps in all four MC versions, with a new episode beginning right after.

NEAR and NEAT were allowed to evolve over 100 generations. Their population consisted of 100 genomes. Each individual genome was evaluated over 100 episodes with *random restarts* in terms of initial position and car speed. Even though very good solutions can be found much earlier in the evolution procedure, the algorithms were allowed to run for that many episodes and generations in order to be in accordance with the experiments presented in [5] and by no means the experimental setup should be interpreted as an inability of the algorithm to find solutions promptly. For example, Fig. 4 displays the performance of the champion gene over 100 generations in the 3D mountain car with velocity information. In each generation, the champion gene was re-evaluated over 1000 episodes with learning and exploration disabled for performance validation purposes. Based on this metric, the champion of champions gene was again evaluated over another 1000 episodes, in order to report on a final generalization performance. The learning parameters of this task were $\gamma=1$, $\lambda=0$, $\alpha=10^{-5}$ and $\epsilon=0.01$ in the cases learning was applied. Table 3 presents the average and standard deviation of the champion of champions performance (CP) along with its final generalization performance (GP) for the four different setups of the mountain car problem. The results are average over 30 runs for all the algorithms evaluated. For the ESN+TD case 100 networks were initialized using the average values of the three properties, N, D and ρ (Table 4), and were allowed to learn for 1000 episodes.

From Table 3 it is evident that the NEAR algorithm is capable of discovering efficient networks whether the state signal includes velocity information or not. NEAR+PS with NEAR+TD-L following closely, derived the highest performing networks and in the classic 2D-MC-M case (with random restarts), equal generalization behavior was obtained to state-of-the-art performances like that of NEAT+Q and SARSA with tile coding [5]. On the other hand it appears that ESNs, even with the same number of neurons, cannot be competent enough without some kind of reservoir optimization. In fact ESN+TD in a couple of cases failed to find an acceptable solution (3D-MC test-bed), hence the high standard deviation in the performances. Table 4 has the calculated statistics of the reservoir properties of the highest performing ESNs discovered by the NEAR+PS methodology. As the agent starts missing state information (for example from 2D-MC-M to 2D-MC-NM), more reservoir features are needed for it to become efficient. Density and spectral radius are almost in the middle part of their ranges, which are [0.01,0.99] for D and [0.55,0.99] for ρ.

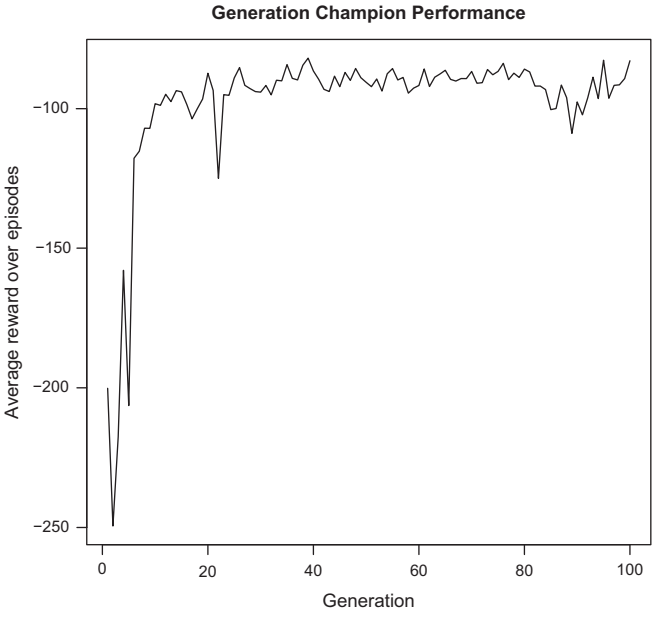

Fig. 4. The generation champion performance measured in average reward obtained over the episodes in the 3D mountain car domain with velocity information. One can observe that a good solution can be found rather early in the evolutionary process.

NEAR+PS was also tested against the Fourier basis expansion [36] methodology, in the 2D mountain car with velocity information testbed and under two settings. In the random restarts setup, Fourier basis expansion average performance over 30 runs, where it was allowed to learn for 1000 episodes, was -56.4 with a 2.01 standard deviation. In the second setup, where the initial position of the car is in the middle of the valley with zero velocity, the Fourier basis method finds a good solutions after 4 episodes. NEAR+PS discovers good solutions during the evaluation of the

Table 3
The average and standard deviation of the champion of champions performance (CP) along with its generalization performance (GP) of each algorithm over 30 runs for the four mountain car testbeds.

Algorithm	2D-MC-M				2D-MC-NM			
	CP	(SD)	GP	(SD)	CP	(SD)	GP	(SD)
NEAT	−49.5	(0.5)	−51.9	(2.2)	−154	(63)	−173	(75)
NEAR+TD-L	−48.9	(0.4)	**−51.3**	(1.4)	**−58**	(2)	**−62**	(6)
NEAR+TD-D	−49.7	(0.8)	−51.5	(1.18)	−60	(2)	−64	(5)
NEAR+PS	**−48.7**	(0.4)	−51.4	(1.8)	−62	(2)	−67	(6)
ESN+TD	−59.6	(11.5)	−59.2	(10.6)	−77	(8)	−77	(8)
Algorithm	3D-MC-M				3D-MC-NM			
	CP	(SD)	GP	(SD)	CP	(SD)	GP	(SD)
NEAT	−113	(10)	−124	(15)	−333	(39)	−368	(52)
NEAR+TD-L	−93	(2)	−95	(3)	−179	(20)	−180	(24)
NEAR+TD-D	−116	(7)	−121	(8)	−191	(16)	−196	(16)
NEAR+PS	**−90**	(3)	**−93**	(3)	**−172**	(25)	**−178**	(23)
ESN+TD	−371	(466)	−375	(473)	−568	(274)	−573	(273)

Table 4
Statistics (average and standard deviation in the parenthesis) over the properties of the best performing networks, i.e. those of NEAR+PS.

Parameter	2D-MC-M	2D-MC-NM	3D-MC-M	3D-MC-NM
N	4.50 (2.78)	6.13 (4.66)	3.80 (1.71)	4.23 (2.48)
D	0.42 (0.34)	0.44 (0.32)	0.54 (0.34)	0.43 (0.31)
ρ	0.73 (0.15)	0.80 (0.14)	0.81 (0.14)	0.78 (0.14)

first generation, which is rather quickly as well. After 1000 episodes of training the Fourier basis methodology allows the car to escape after 114 steps on average over 30 runs, while NEAR+PS establishes an escape route of 123 steps in the first generation and of 104 steps if allowed to optimize the networks further (50 generations).

5.2.2. Pole balancing

The standard pole balancing task concerns a controller that needs to balance a pole hinged on a wheeled cart, on a finite track, by applying force on it. We used five pole balancing tasks for testing the algorithms based on [37]. These tasks have also been used by other authors for analyzing RL algorithms [33,38]: (1) one pole with complete state information (SPB-M), (2) one pole with incomplete state information (SPB-NM), (3) two poles with complete state information (DPB-M), (4) two poles with incomplete state information (DPB-NM) and (5) two poles with incomplete state information but with a fitness function (damping fitness), which does not encourage finding controllers that would oscillate the poles back and forth (DPB-NM-D). This damping fitness function was used by [39] and can be found in many related works [15,33,38,28]. Cart and angular velocities are withheld from the agent when we have incomplete state information.

For the pole balancing environment, we adopted the setup in [33]. In the same reference an extensive evaluation of algorithms was made on four pole balancing variations, which we reference for comparison purposes. Since the highest performing methods in that study implemented policy search through neuroevolution, we tested the NEAR methodology without any learning variations, only NEAR+PS. The performance metric is the number of networks evaluated before a solution is found that will keep the pole or poles balanced. If learning was included then this performance metric would favor the non-learning methods since fewer evaluations would be needed, due to a repetition of episodes to enable learning.

Table 5
The average and standard deviation of the evaluations required over 50 runs. Results with asterisks are taken from [33].

Testbed	SPB-M	SPB-NM	DPB-M	DPB-NM	DPB-NM-D
AVG Eval.	**25**	**1948**	**35**	**420**	**680**
STD Eval.	25	1300	29	254	303
Solutions Found	50	50	50	50	50
NEAT AVG Eval.*	743	1523	3600	6929	24 543
Best AVG Eval.*	98	127	895	1249	3416
NEAR Ranking	1	6	1	1	1

Table 6
Statistics over the properties of the best performing networks for the pole balancing tasks.

Param.	SPB	SPB-NM	DPB	DPB-NM	DPB-NM-D
N	1.00 (0.00)	4.08 (1.57)	1.00 (0.00)	2.22 (0.95)	2.18 (1.08)
D	0.46 (0.26)	0.47 (0.30)	0.55 (0.27)	0.52 (0.26)	0.65 (0.27)
ρ	0.77 (0.12)	0.77 (0.13)	0.72 (0.10)	0.80 (0.14)	0.76 (0.14)

Thus, we only tested the evolutionary pressure NEAR puts into topology and weight evolution of ESNs. The evaluations towards a solution are averaged over 50 runs. NEAR was able to find solutions in all 50 runs, in all four tasks. The results are displayed in Table 5.

In three out of four environments, NEAR exhibits the best performance. The SPB-NM task has proven somewhat more difficult. This is evident from Table 6, where one can observe that for this specific task a larger number of neurons are required to acquire a solution on average. This is probably due to the fact that with just two state variables (position and angle), more non-linear features are needed for developing the final policy. Another remark is that in the DPB-NM task the ρ parameter is higher than in the other test-beds, indicating that the state signals need to remain for a longer period of time in the repository in order for the appropriate computations to take place.

5.2.3. Server job scheduling

Server job scheduling (SJS) [5] is a task that belongs to the realm of autonomic computing. Certain types of jobs are waiting in the job queue of a server to be processed as new jobs arrive in the queue. Each job type has a utility function that changes over time and

represents the user anticipation over having his or her job scheduled (Fig. 5). Certain users want quicker response from the server, while others are not as eager. The goal of the scheduler is to pick a job (action) based on the status of the queue (state), receiving as utility, the value of the function of the scheduled job type at the time-step of execution (immediate reward). The performance is calculated as the sum of utilities (long term reward) when the queue empties. Each task begins with the scheduler finding 100 jobs in the queue, while at each time step a new job is added to the queue for the first 100 ticks. Each episode lasts 200 time-steps. For the state variables and action setup, we used the modeling found in [5].

As in the mountain car case, we follow the experimentation setup in [5]. Again we initialized a population of 100 genomes that was evolved over 100 generations with each genome learned and evaluated over 100 randomly initialized episodes. When learning was involved the same learning parameters were used as in the mountain car environment. CP and GP performances are reported on 1000 new randomly instantiated episodes. The results of Table 7 are averages over 30 runs.

Again we observe that NEAR+PS has dominated over its adversaries. The properties of the highest performing networks over the 30 runs were on average (standard deviation in parenthesis): $D=0.37(0.29)$, $\rho=0.80(0.14)$ and $N=3.29(1.98)$. Another observation is that Lamarckian evolution again outperformed Darwinian evolution as in the mountain car task, following also the conclusions of [5].

5.2.4. Ablation study

Our next goal in this experimental analysis was to discover whether certain major algorithmic components, like crossover and speciation, improve the performance of NEAR. The same kind of study was used in NEAT too [17] and we wanted to check whether the altered models of crossover and speciation we have derived for NEAR are also important components in the evolutionary process. The analysis was performed on the more difficult testbeds using the NEAR+PS alternative. In Table 8 we observe the results of the full algorithm, NEAR+PS, versus its performance when no speciation is present or when no mating is performed, but rather just weight, link and node mutations. Experimentally and following the results of [17] we can see that both components of NEAT and NEAR are productive in developing better networks through neuro-evolution.

Fig. 5. The utilities of the four different job types as time increases.

Table 7
The average and standard deviation of the generalization performance of each algorithm over 30 runs.

Algorithm	SJS			
	CP	(SD)	GP	(SD)
NEAT	−5675.18	(206.01)	−5726.88	(213.75)
NEAR+TD-L	−7856.34	(231.46)	−7856.46	(240.00)
NEAR+TD-D	−9220.41	(608.20)	−9219.92	(590.12)
NEAR+PS	**−5154.69**	(23.28)	**−5189.92**	(25.15)
ESN+TD	−12 160.58	(107.47)	−12 164.19	(112.23)

Table 8
Ablation study for the NEAR algorithm. The performance is presented using the task's corresponding metric and is the average over 30 runs for the mountain car task and 50 runs for the pole balancing task. The results corroborate that both crossover and speciation are important factors in the performance of NEAR.

Method	3D-MC-NM	DPB-NM
Policy search	−178.27	419.98
No crossover	−233.04	538.54
No speciation	−190.05	626.60

5.3. Discussion

In this section, we evaluated and compared several methodologies using multiple test-beds that came from different areas of machine learning. In general we found that NEAR is capable of discovering high performing ESNs through learning and evolution, something not always possible with the original, standard, random, ESN generation.

In the time series problems, the coalition of learning and evolution worked much better than in the RL domain, where NEAR+PS outperformed other alternatives with TD learning enabled, in almost all test-beds. The main reason behind this observation is the fact that through TD-learning the network is trying to calculate the true Q-value, i.e. the prediction of future expected reward, while during policy search, we are satisfied with correct action selection, regardless of the output value. Of course this is not adequate in the time series problems, where an exact value in the output is a necessity to minimize the error. The choice is based on the problem requirements.

Based on the guidelines in [40] we adopted the Friedman test [41,42] in order to test null hypothesis that NEAR and NEAT have on average the same performance. The mean ranking in the 10 testbeds between NEAR and NEAT was 1.1 and 1.9, respectively, producing a χ_F^2 value of 6.4, enough to drop the null hypothesis with $\alpha=0.025$. The dominance of NEAR over the standard NEAT methodology could be mainly attributed to two reasons. The absence of strictly linear features that are known to help other function approximators [43] and the development of ad hoc networks instead of theoretically and experimentally grounded approaches like the echo state networks. On the other hand, the mechanisms of the NEAT methodology have help NEAR in this case and could aid other neuro-evolution algorithms become even more efficient.

The series of experiments also supported conclusions found in similar threads of research like:

- Lamarckian evolution outperforms Darwinian evolution at least in the setups tested above [5].
- The reservoirs do not need to be sparse, especially if they pass through an optimization procedure [29].
- Policy search outperforms temporal difference learning in the absence of noise in the fitness function [44].

- Crossover actually could provide a boost in neuro-evolution methods, when performed in a right way [15,30], like for example using historical markings. The same conclusion could be stated for speciation as well.

Values of D and ρ of the best networks found are spread around the middle of the D and ρ ranges, respectively, while the number of neurons N is small in general. A density of around 45% and a spectral radius of around 0.75 can be considered good initial estimates. Even though these findings are in contrast with best practices of randomly instantiating ESNs, this is probably compensated by topology and weight evolution, structuring the reservoir according to the present D, ρ and N values.

Last but not least, standard deviation on the performance of static ESNs can vary a lot due to the initial randomness. On the other hand through NEAR, we observe a pressure towards consistently discovering networks that solve the task at hand efficiently, with small deviations from an acceptable generalization performance. Thus the neuro-evolution components of NEAT combined with the structure of an ESN are a viable solution towards building models, of the world or of decision making, in tasks that require temporal calculations, like those in time-series and reinforcement learning domains.

6. Conclusions and future work

In this work, we have presented NEAR, a novel methodology, which combines theories and algorithms, in order to develop, autonomously and through adaptation, function approximators that can handle a large variety of machine learning tasks in an efficient way. In particular, we have combined (a) the state-of-the-art neuro-evolution method, NEAT, which has solved problems that had hindered the use of neuro-evolution, (b) the reservoir computing approach of ESNs, and (c) the coupling of learning and evolution. The results of our experiments on 13 different test-beds, in both time-series and reinforcement learning domains, indicate that NEAR is at par, if not better, with state-of-the-art algorithms with respect to performance evaluation. We should also point out the algorithm's RNN structure can handle non-linear and non-Markovian state signals coupled with the ease of use of classic learning schemes like least squares and gradient descent.

Several conclusions have been reached. First of all, crossover during neuro-evolution, if done justifiably, could help produce robust and efficient solutions. Second, policy search outperforms TD-learning, when a crude approximation is needed in order to develop a policy. However, it seems that when exact predictions are necessary, evolution must be coupled with learning. The only case where NEAR methodology lagged behind is with the multiple sinewave oscillator time-series prediction task, which can be attributed to the inherent inability of ESNs to accurately predict several superimposed generators. For this class of problems, NEAR must be augmented with the ability to evolve networks with loosely coupled clusters of closely coupled nodes. On the other hand, we have shown experimentally on numerous testbeds that ESNs can be good function approximators for RL problems if their topology is optimized through a method like neuro-evolution.

This work is the beginning of what seems to be a fruitful area for research and applications. We plan to investigate the improvement of the NEAR algorithm in three directions:

i. The neuroevolution part of NEAR can be further evaluated by applying the algorithm on increasingly more difficult problems [9] and assessing its effectiveness and scalability.
ii. The ESN model can be extended with features like leaky integrator neurons, feedback connectivity, intrinsic plasticity, multiple readout functions, hierarchical or modular architectures evolved through NEAR [7,8].
iii. We can delve deeper into which methods perform better with respect to adapting the output weights. For example, we plan to compare methods ranging from simple hill-climbing and Gaussian mutation to algorithms like PEGASUS [45], LSPI [46], iLSTD [47], GQ(λ) [48], CMA-ES [49], recursive least squares (RLS-TD) [50], among others.

We also plan to study the application of the NEAR methodology to concepts like *transfer learning*, for transferring reservoir features between tasks [51], and *co-evolution* [52], where policies could be co-evolved through NEAR in host and parasite populations, for example in games [53] or other multi-agent settings [54].

Further research is needed to reveal NEAR's behavior in non-stationary problems and to study its speed of adaptation to the changing problem distribution. Probably this is a case where one can evolve robust networks through Darwinian evolution and let learning adjust the output weights, on-line, to the distribution at hand.

Finally, the concepts of symbolic reasoning in neural systems can be explored with respect to ESNs, in general, and NEAR, in particular. Such examples can be found in the technical programs of the neural-symbolic learning and reasoning workshops.

Appendix A. Parameter values

In this appendix, parameter values are reported for all algorithms discussed in this paper in order to help the interested reader in the recreation of the experiments. One can find standard parameter values for NEAR in Table A1, while For the NEAT method and the MC and SJS domains we have used the parameter values reported in [5], except for the probability of adding a recurrent connection which was set to 0.2. A bias input node was used in the time series problems, while no such node was included in the RL experiments. The time series samples $x(i)$ were scaled by subtracting the mean and dividing by the range $(x(i)-mean)/(max-min)$ of the data-set, while normalization of the input signal was performed in the RL experiments between -1 and 1. The idea is so that the reservoir neurons to work in the linear part of their activation function.

Table A2 contains parameter values that have changed over the testbeds. In the time series problems, the rate of adding a node through probability p_n was increased in order to quickly scan through larger and larger networks and pass barriers imposed by the network size. As for the pole balancing non-Markovian tasks mainly the power of perturbing the weights was largely increased, so that the policy search has a greater reach in the range of its search.

The implementations of the sample generators for the Mackey–Glass, multiple-sinewave oscillator and the Lorentz attractor were programmed by the authors, along with the server job scheduling environment. The 2D and 3D mountain car tasks, the single pole balancing task and the Fourier basis expansion algorithm were downloaded from the rl-library[2] and adapted to our requirements. The core functionality of the double pole balancing environment was taken as is from the ESP-Java library,[3] in order to be in unison with the referenced literature, only altered to fit the rl–glue interfaces.

[2] http://library.rl-community.org/.
[3] http://nn.cs.utexas.edu/?ESP-Java.

Table A1
Default values of the parameters in our methodology for time series and RL problems.

Parameter	Value	Parameter	Value	Parameter	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
$N^{initial}$	1	p_i	0.001	w_{pow}	0.25
S_{thres}	0.3	$Stagnation_{max}$	5		

Table A2
Parameter values that are different between testbeds.

Parameter	SPB-NM	DPB-NM	DPB-NM-D	MG	MSO	Lorentz
w_{pow}	2.5	2.5	2.5	1	1	1
p_c	0.05	0.05	0.05	0.2	0.2	0.2
p_n	0.1	0.1	0.1	0.3	0.3	0.3
$N^{initial}$	1	1	1	25	25	25

References

[1] P. Stone, Learning and multiagent reasoning for autonomous agents, in: Proceedings of the 20th International Joint Conference on Artificial Intelligence, 2007 pp. 13–30.
[2] X. Yao, Evolving artificial neural networks, Proc. IEEE 87 (1999) 1423–1447.
[3] R.S. Sutton, A.G. Barto, Reinforcement Learning: An Introduction, MIT Press, Cambridge, MA, 1998.
[4] D. Floreano, P. Dürr, C. Mattiussi, Neuroevolution: from architectures to learning, Evol. Intell. 1 (2008) 47–62.
[5] S. Whiteson, P. Stone, Evolutionary function approximation for reinforcement learning, J. Mach. Learn. Res. 7 (2006) 877–917.
[6] H. 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.
[7] H. 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.
[8] M. Lukoševičius, H. Jaeger, Reservoir computing approaches to recurrent neural network training, Comput. Sci. Rev. 3 (2009) 127–149.
[9] D. Prokhorov, Echo state networks: appeal and challenges, in: Proceedings of the International Joint Conference on Neural Networks (IJCNN), Montreal, Canada 2005.
[10] I. Szita, V. Gyenes, A. Lőrincz, Reinforcement learning with echo state networks, in: Artificial Neural Networks—ICANN 2006, Lecture Notes in Computer Science, vol. 4131/2006, Springer, Berlin, Heidelberg, 2006, pp. 830–839.
[11] S.J. Bratke, A.G. Barto, Linear least-squares algorithms for temporal difference learning, Mach. Learn. 22 (1996) 33–57.
[12] J.A. Boyan, Technical update: least-squares temporal difference learning, Mach. Learn. 49 (2002) 233–246.
[13] A.E. Eiben, J.E. Smith, Introduction to Evolutionary Computation, Natural Computing Series, Springer-Verlag, 2003.
[14] C. Igel, B. Sendhoff, Genesis of organic computing systems: coupling evolution and learning, in: R.P. Würtz (Ed.), Organic Computing. Understanding Complex Systems, Springer-Verlag, Berlin, Heidelberg, 2008.
[15] K.O. Stanley, Efficient Evolution of Neural Networks, Ph.D. Thesis, University of Texas at Austin, 2004.
[16] N.J. Radcliffe, Genetic set recombination and its application to neural network topology optimization, Neural Comput. Appl. 1 (1993) 67–90.
[17] K.O. Stanley, R. Miikkulainen, Evolving neural networks through augmenting topologies, Evol. Comput. 10 (2002) 99–127.
[18] G.E. Hinton, S.J. Nowlan, How learning can guide evolution, Complex Syst. 1 (1987) 495–502.
[19] J.M. Baldwin, A new factor in evolution, Am. Nat. 30 (1896) 441–451.
[20] D.H. Ackley, M.L. Littman, Interactions between learning and evolution, in: C.G. Langton, C. Taylor, J.D. Farmer, S. Rasmussen (Eds.), Artificial Life II, SFI Studies in the Sciences of Complexity, vol. X, Addison-Wesley, Reading, MA, 1991, pp. 487–509.
[21] D.H. Wolpert, W.G. Macready, No free lunch theorems for optimization, IEEE Trans. Evol. Comput. 1 (1997) 67–82.
[22] R. Miikkulainen, Neuroevolution, in: C. Sammut, G.I. Webb (Eds.), Encyclopedia of Machine Learning, Springer, 2011.
[23] H. Jaeger, M. Lukoševičius, D. Popovici, U. Siewert, Optimization and applications of echo state networks with leakyintegrator neurons, Neural Networks 20 (2007) 335–352.
[24] K. Ishii, T. van der Zant, V. Bečanović, P. Plöger, Identification of motion with echo state network, in: Proceedings of the OCEANS 2004 MTS/IEEE - TECHNO-OCEAN 2004 Conference, vol. 3, 2004, pp. 1205–1210.
[25] D. Xu, J. Lan, J.C. Principe, Direct adaptive control: an echo state network and genetic algorithm approach. in: Proceedings of the International Joint Conference on Neural Networks, Montreal, Canada, 2005, pp. 1483–1486.
[26] K. Bush, B. Tsendjav, Improving the richness of echo state features using next ascent local search. in: Proceedings of the Artificial Neural Networks in Engineering Conference, St. Louis, MO, 2005, pp. 227–232.
[27] Š. Babinec, J. Pospíchal, Two approaches to optimize echo state neural networks, in: Proceedings of Mendel 2005, 11th International Conference on Soft Computing, 2005, pp. 39–44.
[28] F. Jiang, H. Berry, M. Schoenauer, Supervised and evolutionary learning of echo state networks, in: Proceedings of 10th International Conference on Parallel Problem Solving from Nature, PPSN 2008, Lecture Notes in Computer Science, vol. 5199, Springer-Verlag, 2008, pp. 215–224.
[29] Benjamin Roeschies, Christian Igel, Structure optimization of reservoir networks, Logic Journal of the IGPL 18 (5) (2010) 635–669.
[30] S. Haflidason, R. Neville, Quantifying the severity of the permutation problem in neuroevolution, in: F. Peper, H. Umeo, N. Matsui, T. Isokawa (Eds.), Natural Computing, Proceedings in Information and Communications Technology, vol. 2, Springer, Japan, 2010, pp. 149–156.
[31] A. Livnat, C. Papadimitriou, J. Dushoff, M.W. Feldman, A mixability theory of the role of sex in evolution, in: Proceedings of the National Academy of Sciences of the United States of America. 2009.
[32] B. Tanner, A. White, Rl-glue: language-independent software for reinforcement-learning experiments, J. Mach. Learn. Res. 10 (2009) 2133–2136.
[33] F. Gomez, J. Schmidhuber, R. Miikkulainen, Efficient non-linear control through neuroevolution, in: Proceedings of the European Conference on Machine Learning (ECML 2006), vol. 4212/2006 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2006, pp. 654–662.
[34] S.P. Singh, R.S. Sutton, Reinforcement learning with replacing eligibility traces, Mach. Learn. 22 (1996) 123–158.
[35] M.E. Taylor, G. Kuhlmann, P. Stone, Autonomous transfer for reinforcement learning, in: AAMAS '08: Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems, 2008, pp. 283–290.
[36] G. Konidaris, S. Osentoski, P. Thomas, Value function approximation in reinforcement learning using the Fourier basis, in: Proceedings of the 25th Conference on Artificial Intelligence, 2011 pp. 380–385.
[37] A.P. Wieland, Evolving neural network controllers for unstable systems, in: Proceedings of the International Joint Conference on Neural Networks, IEEE, Seattle, WA, 1991, pp. 667–673.
[38] F. Gomez, J. Schmidhuber, R. Miikkulainen, Accelerated neural evolution through cooperatively coevolved synapses, J. Mach. Learn. Res. 9 (2008) 937–965.
[39] F. Gruau, D. Whitley, L. Pyeatt, A comparison between cellular encoding and direct encoding for genetic neural networks, in: J.R. Koza, D.E. Goldberg, D.B. Fogel, R.L. Riolo (Eds.), Genetic Programming 1996: Proceedings of the First Annual Conference, MIT Press, Cambridge, MA, 1996, pp. 81–89.
[40] J. Demšar, Statistical comparisons of classifiers over multiple data sets, J. Mach. Learn. Res. 7 (2006) 1–30.
[41] M. Friedman, The use of ranks to avoid the assumption of normality implicit in the analysis of variance, J. Am. Stat. Assoc. 32 (1937) 675–701.
[42] M. Friedman, A comparison of alternative tests of significance for the problem of m rankings, Ann. Math. Stat. 11 (1940) 86–92.
[43] J.H. Friedman, B.E. Popescu, Predictive learning via rule ensembles, Ann. Appl. Stat. 2 (2008) 916–954.
[44] S. Whiteson, M.E. Taylor, P. Stone, Critical factors in the empirical performance of temporal difference and evolutionary methods for reinforcement learning, J. Autonomous Agents Multi-Agent Syst. 21 (2009) 1–27.
[45] A.Y. Ng, M. Jordan, Pegasus: a policy search method for large MDPs and POMDPs. in: Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence, Stanford, California, 2000, pp. 406–415.
[46] M.G. Lagoudakis, R. Parr, Least-squares policy iteration, J. Mach. Learn. Res. 4 (2003) 1107–1149.
[47] A. Geramifard, M. Bowling, R.S. Sutton, Incremental least-squares temporal difference learning, in: Proceedings of the 21st National Conference on Artificial Intelligence and the 18th Innovative Applications of Artificial Intelligence Conference, 2006 pp. 356–361.
[48] H.R. Maei, R.S. Sutton, GQ(λ): a general gradient algorithm for temporal-difference prediction learning with eligibility traces. in: Proceedings of the Third Conference on Artificial General IntelligenceLugano, Switzerland, 2010.
[49] N. Hansen, A. Ostermeier, Completely derandomized self-adaptation in evolution strategies, Evol. Comput. 9 (2001) 159–195.
[50] X. Xu, H. He, D. Hu, Efficient reinforcement learning using recursive least-squares methods, J. Artif. Intell. Res. 16 (2002) 259–292.
[51] M.E. Taylor, P. Stone, Transfer learning for reinforcement learning domains: a survey, J. Mach. Learn. Res. 10 (2009) 1633–1685.
[52] C.D. Rosin, R.K. Belew, New methods for competitive coevolution, Evol. Comput. 5 (1997) 1–29.
[53] J.B. Pollack, A.D. Blair, Co-evolution in the successful learning of backgammon strategy, Mach. Learn. 32 (1998) 225–240.
[54] K.O. Stanley, R. Miikkulainen, Competitive coevolution through evolutionary complexification, J. Artif. Intell. Res. 21 (2004) 63–100.

Kyriakos C. Chatzidimitriou obtained his MSc degree from the Computer Science Department of Colorado State University, USA, in 2006 and his Diploma from the Electrical and Computer Engineering Department of the Aristotle University of Thessaloniki, Greece, in 2003. He is currently a PhD candidate with the later department and a research associate at the Informatics and Telematics Institute of CERTH. His research interests include the areas of reinforcement learning, evolutionary computation, neural networks and autonomous agents for real world domains. In 2012, with team Mertacor, he got the 1st place in the international Trading Agent Competition, Ad Auctions game.

Pericles A. Mitkas received his Diploma of Electrical Engineering from Aristotle University of Thessaloniki (AUTH) in 1985 and an MSc and PhD in Computer Engineering from Syracuse University, in 1987 and 1990. He was a faculty member with the Department of Electrical and Computer Engineering at Colorado State University. Currently he is a professor at the Department of Electrical and Computer Engineering of AUTH and a faculty affiliate of the Informatics and Telematics Institute of CERTH. Dr. Mitkas has served as Primary Investigator or Project Coordinator in several US and European research projects.