Swarm Intell (2015) 9:229–265
DOI 10.1007/s11721-015-0112-z

Learning neural network structures with ant colony algorithms

Khalid M. Salama[1] · Ashraf M. Abdelbar[2]

Received: 18 October 2014 / Accepted: 15 September 2015 / Published online: 9 October 2015

Abstract Ant colony optimization (ACO) has been successfully applied to classification, where the aim is to build a model that captures the relationships between the input attributes and the target class in a given domain's dataset. The constructed classification model can then be used to predict the unknown class of a new pattern. While artificial neural networks are one of the most widely used models for pattern classification, their application is commonly restricted to fully connected three-layer topologies. In this paper, we present a new algorithm, ANN-Miner, which uses ACO to learn the structure of feed-forward neural networks. We report computational results on 40 benchmark datasets for several variations of the algorithm. Performance is compared to the standard three-layer structure trained with two different weight-learning algorithms (back propagation, and the $\mathrm{ACO}_{\mathbb{R}}$ algorithm), and also to a greedy algorithm for learning NN structures. A nonparametric Friedman test is used to determine statistical significance. In addition, we compare our proposed algorithm with NEAT, a prominent evolutionary algorithm for evolving neural networks, as well as three different well-known state-of-the-art classifiers, namely the C4.5 decision tree induction algorithm, the Ripper classification rule induction algorithm, and support vector machines.

Keywords Ant colony optimization (ACO) · Machine learning · Pattern classification · Neural networks

✉ Ashraf M. Abdelbar
abdelbara@brandonu.ca

Khalid M. Salama
kms39@kent.ac.uk

1 School of Computing, University of Kent, Canterbury, UK

2 Department of Mathematics and Computer Science, Brandon University, Brandon, MB, Canada

1 Introduction

Machine learning is an active research area involving the development of methods for automated data mining and analysis, which aims to uncover useful knowledge for decision making applications in real-world domains (Bishop 2006; Witten et al. 2010). Classification is a central data mining task concerned with predicting the class of a given pattern based on its input attributes, using a well-constructed model (classifier) (Han et al. 2011a; Tan et al. 2005). Bank credit scoring, financial fraud detection, churn analysis, and targeted advertising are examples of well-known classification problems, in addition to applications in numerous, diverse fields, including bioinformatics, healthcare, and engineering (Han et al. 2011a; Tan et al. 2005; Witten et al. 2010).

The process of building a classifier consists of two stages. The training stage utilizes a training set of labeled patterns—i.e., a set of patterns along with their correct class labels—that should be sufficiently representative of the domain of interest. A classification algorithm uses the training set to construct a model that captures the relationships between the attributes of the input patterns and their corresponding class labels. Then, during the subsequent operating stage, the model is used to predict the class of new unlabeled patterns that were not present during the training stage.

Inspired by biological systems, artificial neural networks (NN) (Haykin 2008) are one of the most widely studied and applied models for pattern discrimination (classification). NNs are generally presented as systems of inter-connected computational units (neurons)—each takes inputs and produces an output—and a set of real-valued weights associated with inter-neuronal connections. The connectivity structure of a NN, the activation function of the neurons, and the connection weights determine the decision boundaries that separate patterns with different classes in the data space, and are used to determine the class of a new pattern. The most commonly used pattern discrimination NNs are feed-forward neural networks (FFNN) with a three-layer topology (structure). The structure consists of an input layer, a single hidden layer, and an output layer, with full connectivity between the neurons in one layer and the neurons in the following layer. In a FFNN, the size of the input and output layers are determined by characteristics of the dataset, while the number of neurons in the hidden layer is often manually determined by practitioners based on various heuristics involving the number of input and output neurons, the size of the training set, the expected number of training iterations, and the estimated difficulty of the problem at hand.

In general, much of the work in the NN literature has focused on studying and developing techniques for training (i.e., learning the connection weights of) a NN with a given user-defined structure. Allowing arbitrary feed-forward topologies and automatically optimizing the structure of a NN—based on a dataset at hand—can lead to more effective pattern classification models. The present paper is an extended version of the ANTS 2014 conference paper (Salama and Abdelbar 2014), where ANN-Miner, an Ant-based Neural Network structure learning algorithm, was introduced.

In this paper, we build on the work described by Salama and Abdelbar (2014) in six ways. First, we use the $\mathrm{ACO}_{\mathbb{R}}$ algorithm (Socha and Dorigo 2008) for training the NN structures produced by our ANN-Miner algorithm. $\mathrm{ACO}_{\mathbb{R}}$ is a state-of-the-art ant colony algorithm for continuous optimization problems, which has been applied to training NNs with the standard three-layer structure (Blum and Socha 2005; Socha and Blum 2007). We also compare ANN-Miner to standard $\mathrm{ACO}_{\mathbb{R}}$ with the three-layer structure. Second, we use the quadratic loss function as a more effective function for evaluating the quality of the candidate constructed NNs, compared to the simple accuracy quality evaluation function that was used by Salama and Abdelbar (2014). Third, we compare our ACO-based algorithms with a baseline greedy hill-climber (GHC). Fourth, the number of datasets used in the experimental evaluation is increased from 20 to 40. Fifth, we compare our proposed approach to three different state-of-the-art classifiers, namely the C4.5 decision tree induction algorithm, the Ripper classification rule induction algorithm, and support vector machines (SVM), as well as to two well-known baseline classifiers, namely one-nearest-neighbor and Naïve Bayes. Sixth, we compare our proposed approach to NEAT (Stanley and Miikkulainen 2002; Stanley et al. 2005b, 2009), a prominent evolutionary algorithm for evolving neural networks.

The remainder of the paper is organized as follows. An overview of the ACO meta-heuristic is given in Sect. 2. We discuss the ACO related work—generally in classification and specifically in NNs—in Sect. 3. A background on feed-forward neural networks, along with different techniques for NN weight learning, is provided in Sect. 4. In Sects. 5 and 6, we describe our ANN-Miner algorithm and its related variations. Section 7 presents a review of related neuroevolutionary methods. Sections 8 and 9 report our experimental methodology and results, respectively. We conclude with some general remarks and directions for future research in Sect. 10.

2 Ant colony optimization

Ant colony optimization (ACO) (Dorigo et al. 1999; Dorigo and Stützle 2004, 2010) is a meta-heuristic for combinatorial optimization problems, inspired by the behavior of natural ant colonies. The basic principle of ACO is that a population of artificial ants cooperate with each other to find the best path in a graph, analogously to the way that natural ants cooperate to find the shortest path between two points such as their nest and a food source (Dorigo and Stützle 2004, 2010; Dorigo et al. 1996).

In ACO, each artificial ant constructs a candidate solution to the target problem, represented by a combination of solution components in the search space. Ants cooperate via indirect communication, by depositing pheromone on the selected solution components for a candidate solution. The amount of pheromone deposited is proportional to the quality of that solution, which influences the probability with which other ants will use that solution's components when constructing their solution. This contributes to the global search aspect of ACO algorithms. In addition, the probability of an ant choosing a solution component often also depends on the value of a heuristic function that measures the desirability of that component.

The global search aspect is also promoted by the fact that a population of ants will search for the best solution in parallel, thus exploring possibly different regions of the search space at each iteration of the algorithm. As a result of this global search, ACO is less likely to get trapped in local optima than conventional greedy algorithms, which increases the chances of finding a near-optimal solution in the search space. Note that in some widely used variations of the ACO procedure (e.g., $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System (Stützle and Hoos 2000)), multiple ant solutions are created in an iteration of the algorithm, and then the ant with the best created solution updates the pheromone trail. We use this behavior of the $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System in our ANN-Miner algorithm, as described in Sect. 5.2.

3 Related work on ACO for pattern classification

ACO has been successful in tackling the classification problem of data mining. A number of ACO-based algorithms have been introduced in the literature for learning different types of classification models, including classification rules, decision trees, Bayesian networks, and neural networks.

Ant-Miner (Parpinelli et al. 2002) is the first ant-based classification algorithm which discovers a classification model comprised of a list of IF–THEN classification rules. The algorithm has been followed by several extensions, such as Ant-Miner+ (Martens et al. 2007), FRANTIC-SRL (Galea and Shen 2006), *c*Ant-Miner (Otero et al. 2009), Multi-pheromone Ant-Miner (Salama et al. 2011, 2013), and recently *c*Ant-Miner$_{\mathrm{PB}}$ (Otero et al. 2013; Otero and Freitas 2013).

ACDT (Boryczka and Kozak 2010, 2011) and Ant-Tree-Miner (Otero et al. 2012; Salama and Otero 2014) are two different ACO-based algorithms for inducing decision trees for classification. Salama and Freitas have recently employed ACO to optimize the dependency relationships in various types of Bayesian network classifiers, such as Bayesian network augmented naïve-Bayes (Salama and Freitas 2013b), Bayesian multinets (Salama and Freitas 2014b, 2015), and class Markov blankets (Salama and Freitas 2013, 2014a).

In the context of pattern classification neural networks, the ACO meta-heuristic was utilized for learning NN weights in two previous works. Liu et al proposed ACO-PB, a hybrid of the ant colony and back propagation (BP) algorithms, to optimize NN weights (Liu et al. 2006). They use ACO to search a discretized set of weight values, and then use BP to fine-tune the discrete weights found by ACO. Blum and Socha applied ACO$_{\mathbb{R}}$, an ACO algorithm for continuous optimization (Socha and Dorigo 2008; Liao et al. 2014), to train feed-forward neural networks (Blum and Socha 2005; Socha and Blum 2007). We revisit ACO$_{\mathbb{R}}$ in more detail in Sect. 4.2 as we use it in our experiments with our ANN-Miner algorithm.

Note that, to the best of our knowledge, ACO has not been previously applied to learning the structure of neural networks prior to the introduction of our ANN-Miner algorithm. For a comprehensive review of ACO algorithms in data mining, the reader is referred to the survey by Martens et al. (2011).

4 Feed-forward neural networks

One of the most popular and well-established methods for pattern classification are feed-forward neural networks (FFNN), which are neural networks in which the pattern of connections between neurons is acyclic. The most common FFNN topology is a three-layer structure in which neurons are arranged in an input layer, a hidden layer, and an output layer, with full connectivity between layers—i.e., the output of every neuron in a layer feeds in as an input to every neuron in the succeeding layer. The external input to the network feeds into the input layer, and the network's external output is the output of the output layer.

Each neuron i is fairly simple and can be considered to be a simple circuit which receives r inputs $o_1, \ldots, o_r$ (these inputs may represent the outputs of neurons in the previous layer or may represent the network's external inputs) and produces a single output o_i:

$$net_i = \sum_{j=1}^{r} w_{ij} o_j + \theta_i \tag{1}$$

$$o_i = f(net_i) \tag{2}$$

where each input o_j is the output of a neuron in the previous layer, the weight w_{ij} represents a real-valued weight between neuron j and neuron i, θ_i represents a weight associated with neuron i itself called the neuron's self-bias, and f is a nonlinear *activation function*. Note that input neurons do not have self-biases.

After an input pattern x is presented to the network, the output of the network is observed and is referred to as the actual output vector y'. A discrepancy function E is used to compare the target output y to the actual output y' resulting in a scalar error value. A common discrepancy function is the simple sum of squared error:

$$E = \sum_{p \in P} E_p \tag{3}$$

where P is the set of training patterns and

$$E_p = \frac{1}{2} \sum_{i=1}^{m} (y_i - y_i')^2 \tag{4}$$

where m is the number of classes.

In pattern classification applications, the target vector y is m-dimensional where m is the number of classes. For a pattern with class label $\hat{c}$:

$$y_k = \begin{cases} 1 & \text{if } k = \hat{c} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

The weights and self-biases of a given FFNN are collectively referred to as the network's *weight vector* w. For example, a FFNN with four neurons in the input layer, five neurons in the hidden layer, and three neurons in the output layer would have a weight vector of 43 real numbers. If the weight vector for a given network is fixed, then the output of the network is a function of its input, and the total error E of the network is a mathematical function of the training set. If the training set is fixed, then the error E is a function of the weight vector w. Our objective is to find the value of the weight vector w which minimizes the error E.

4.1 Backward error propagation

One of the earliest, and still most popular, approaches to neural network learning is based on gradient descent. For each element w_i of the weight vector, the partial derivative $\frac{\partial E}{\partial w_i}$ represents w_i's contribution to the network error. Therefore, the gradient-descent principle is that each w_i should be changed by an amount Δw_i,

$$w_i = w_i + \Delta w_i \tag{6}$$

such that:

$$\Delta w_i \propto -\frac{\partial E}{\partial w_i} \tag{7}$$

This can be implemented as:

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \tag{8}$$

where η is referred to as the learning rate.

The back propagation (BP) algorithm (Werbos 1994) provides a mechanism for computing $\frac{\partial E}{\partial w_i}$ for each element w_i in the weight vector by first computing the contribution of each neuron in the output layer to the error, and then propagating backwards through the network to compute the contribution of each neuron and weight in previous layers to the error.

4.2 Using ACO$_{\mathbb{R}}$ to learn NN weights

The ACO$_{\mathbb{R}}$ algorithm has recently been applied to learning the weights of a fixed-topology FFNN (Blum and Socha 2005; Socha and Blum 2007). In this application of ACO$_{\mathbb{R}}$, an archive of L previous solutions is maintained, where a solution in this context refers to an instantiation of the weight vector w. In ACO$_{\mathbb{R}}$, the solution archive plays the role that pheromone plays in other ACO algorithms.

In each iteration, each ant in the colony generates a candidate solution, again where each candidate solution is an instantiation of the weight vector. If there are m ants in the colony, then m solutions (weight vectors) are generated per iteration, and these m solutions are added to the archive, which is now temporarily of size $(L+m)$. The worst m solutions in the archive are then identified and discarded, thereby returning the archive to size L.

To evaluate a candidate solution (weight vector) w, the weight vector w is used to initialize the weights of a neural network. The training set is then applied to the network, and the total error over the training set is taken as a measure of the quality of the candidate solution—the lower the error, the better the solution.

At the start of each iteration, the solutions in the archive are sorted by quality, with the best solution being given a rank of 1 and the worst a rank of L. Each solution s_i of rank i is given a coefficient ω_i computed as:

$$\omega_i = g(i; 1, qL) \tag{9}$$

where g denotes the Gaussian function:

$$g(x; \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{10}$$

This means that the coefficient ω_i is assigned to be the value of the Gaussian function with argument i, mean 1.0, and standard deviation equal to qL, where q is a user-supplied parameter. Note that smaller values of q cause the better ranked solutions to have higher coefficients ω. Further, note that:

$$\omega_1 > \omega_2 > \cdots > \omega_L \tag{11}$$

Next, let us consider how an ant constructs a solution. Let u denote the solution being constructed. Recall that u is a weight vector whose dimensionality depends on the topology of the network. The first step to constructing u is to select one of the L solutions in the archive by which to be influenced in the construction process. If the r-th solution in the (sorted) archive, of rank r, is denoted s_r, then a solution is selected based on:

$$\Pr(\text{select } s_a) = \frac{\omega_a}{\sum_{r=1}^{L} \omega_r} \tag{12}$$

Let s_a be the archive solution that is selected according to Eq. (12). Each element of u is then generated by sampling the Gaussian probability density function (PDF):

$$u_j \sim N(s_{aj}, \sigma_{aj}) \tag{13}$$

where $N(\mu, \sigma)$ denotes the Gaussian PDF with mean μ and variance σ^2, s_{aj} represents the value of the j-the element of the solution s_a selected using Eq. (12), and σ_{aj} is computed as:

$$\sigma_{aj} = \xi \sum_{r=1}^{L} \frac{|s_{aj} - s_{rj}|}{L-1} \tag{14}$$

where ξ is a user-supplied parameter of the algorithm which plays a role similar to evaporation rate in other ACO algorithms. The higher the value of ξ, the slower the speed of convergence of the algorithm.

Once each ant has constructed its solution, the archive is updated as described above. The process repeats until the desired termination criteria are met.

5 A novel ACO algorithm for learning neural network structures

Unlike many neural network applications that use a simple three-layer network topology with full connectivity between layers (as discussed in Sect. 4), we allow our ACO-based technique to deviate from this, as follows. ANN-Miner allows connections to be generated between hidden neurons and other hidden neurons—under the restriction that the topology remain acyclic—as well as direct connections between input neurons and output neurons. This permits the production of networks with a variable number of layers, as well as arbitrary connections that skip over layers. The ACO elements of the ANN-Miner algorithm are defined in the following subsections.

5.1 ACO construction graph

In general, the core element of an ACO-based algorithm is the construction graph, which contains the solution components in the search space, and with which an ant constructs a candidate solution. In the case of the problem at hand, a candidate solution is a network structure, and the solution components are the selected connections between the neurons. The number of input neurons and output neurons depends of course on the dataset and the representation that is used for the attributes of the dataset, while the total number of hidden neurons is a user-supplied parameter. Suppose the total number of neurons is N, with N_i input neurons, N_o output neurons, and N_h hidden neurons. The set of available potential connections, denoted C, will then be comprised of four types of potential connections:

1. Connections between input and hidden neurons (specifically $N_i \times N_h$ connections),
2. Connections between hidden and output neurons (specifically $N_h \times N_o$ connections),
3. Connections between input and output neurons (specifically $N_i \times N_o$ connections),
4. Connections between different hidden neurons.

The available connections between the N_h hidden neurons are defined as follows. In order to ensure that the network structure is acyclic, we impose the restriction that the connection $(n_i \rightarrow n_j)$ is not available if $n_i \geq n_j$. In other words, each hidden neuron has a numeric index, and we only allow connections from a given hidden neuron n_i to a higher-numbered neuron n_j. It is well known that any directed acyclic graph is isomorphic to a graph where the nodes are lexicographically ordered, and for all arcs (u, v) in the graph, u precedes v in the lexicographic order. Hence, the number of available connections between the N_h hidden neurons is

$$(N_h - 1) + (N_h - 2) + \cdots + 1 + 0 = N_h(N_h - 1)/2$$

As previously mentioned, the number of input and output neurons, N_i and N_o, are determined by the characteristics of the dataset. In the work described in this paper, we set the number of available hidden neurons to $N_h = N_i + N_o$.

It is interesting that our ACO procedure can be viewed as *pruning* the maximally connected network structure that contains all $|C|$ possible connections by selecting which connections to include in the network structure and which connections to exclude. Note that using the maximally connected NN structure may harm the generalization ability of the produced NN. That is, a NN model with a large number of connections can potentially overfit the training (in-sample) data during the training phase, by capturing noisy relationships related only to the training set. Consequently, this model might not perform well on new (out-of-sample) data.

Hence, each potential connection $c = n_i \rightarrow n_j$, connecting neuron i to neuron j, has two solution components in the construction graph: D_c^{true}, representing the decision to include connection $n_i \rightarrow n_j$ in the current candidate NN structure being constructed by the ant, and D_c^{false}, representing the decision not to include the connection. Each solution component D_c^a is associated with a pheromone amount (indirectly representing an estimate of the quality of this component in constructing effective candidate NN models). Therefore, the construction graph can be represented as a two-dimensional $2 \times |C|$ array, consisting of an element D_c^a for every $c = 1, \ldots, |C|$, and $a \in \{false, true\}$.

5.2 A high-level view of the ANN-miner algorithm

The pseudo-code of the ANN-Miner algorithm is given in Algorithm 1. In the initialization step of ANN-Miner (line 4), the amount of pheromone assigned to each solution component D_c^a—where a can be true or false—in the construction graph is initialized with the value 0.5. Hence, for each connection c, the probability of including $i \rightarrow j$ (i.e., selecting D_c^{true}) in the topology equals the probability of not including $i \rightarrow j$ (i.e., selecting D_c^{false}).

Algorithm 1 Pseudo-code of ANN-Miner.

```
1: **Begin**
2: $NN_{bsf} = \phi$;
3: $t = 1$;
4: $InitializePheromone()$;
5: **repeat**
6:     $NN_{tbest} = \phi$;
7:     $Q_{tbest} = 0$;
8:     **for** $i = 1 \rightarrow$ colony_size **do**
9:         $NN_i = ant_i.CreateSolution()$;
10:        $Q_i = EvaluateQuality(NN_i)$;
11:        **if** $Q_i > Q_{tbest}$ **then**
12:            $NN_{tbest} = NN_i$;
13:            $Q_{tbest} = Q_i$;
14:        **end if**
15:    **end for**
16:    $UpdatePheromone(NN_{tbest}, Q_{tbest})$;
17:    **if** $Q_{tbest} > Q_{bsf}$ **then**
18:        $NN_{bsf} = NN_{tbest}$;
19:        $Q_{bsf} = Q_{tbest}$;
20:    **end if**
21:    $t = t + 1$;
22: **until** $t =$ max_iterations **or** $Convergence($conv_iterations$)$;
23: $NN_{final} = PostProcessing(NN_{bsf})$;
24: **return** $NN_{final}$;
25: **End**
```

The pseudo-code of the ANN-Miner algorithm follows the approach of $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System (Stützle and Hoos 2000), where in each iteration, each ant in the colony constructs a solution, and the ant with the best constructed solution updates the pheromone trail. In the inner **for**-loop (lines 8–15), each ant_i in the colony creates a candidate solution NN_i, i.e., a complete neural network (line 9). Then the quality of the constructed solution is evaluated (line 10). The best solution NN_{tbest} produced in the colony is selected to update the pheromone trail by an amount that is proportional to the quality of its solution Q_{tbest}. After that, the algorithm compares the iteration-best solution NN_{tbest} with the best-so-far solution NN_{bsf} (the **if** statement in lines 17–20) to keep track of the best solution found so far during the algorithm's execution.

This set of steps is considered an iteration of the outer **repeat-until** loop (lines 5–22) and is repeated until the same solution is generated for a number of consecutive iterations specified by the `conv_iterations` parameter (indicating convergence) or until `max_iterations` is reached. The values of `conv_iterations`, `max_iterations` and `colony_size` are user-specified. The parameter settings used in our experiments are shown in Sect. 8.2.

The best-so-far neural network undergoes an (optional) post-processing step (line 23) to produce the final neural network NN_{final} to be returned by the algorithm. Basically, the algorithm learns the final weights of the connections in the neural network NN_{bsf}—which uses the best NN structure found during the search process of the ACO algorithm. This is discussed in Sect. 6.2.

5.3 Creating a candidate solution

Algorithm 2 Pseudo-code of solution creation procedure.

```
1: **Begin** CreateSolution()
2: $NN \leftarrow \phi$;
3: **for** $c \in C$ **do**
4:     $D_c^a = SelectDecisionComponent()$;
5:     **if** $D_c^a == D_c^{true}$ **then**
6:         $NN = NN \cup (i \rightarrow j)_c$;
7:     **end if**
8: **end for**
9: $TrainNeuralNetwork(NN, \mathcal{T}_l)$;
10: **return** $NN$;
11: **End**
```

Algorithm 2 describes the process of creating a new candidate solution (neural network), which is called as a subroutine in line 9 of Algorithm 1. The procedure starts with an empty (edge-less) neural network (line 2). For each connection c in the available set of connections C, the ant selects D_c^a to decide whether to include this connection in the candidate network NN or not (line 3)—by either selecting solution component D_c^{true} or D_c^{false}. The selection of the solution component at each step is based on the following probabilistic state transition formula:

$$p(D_c^a) = \frac{\tau\left(D_c^a\right)}{\tau\left(D_c^{true}\right) + \tau\left(D_c^{false}\right)} \tag{15}$$

where $p(D_c^a)$ is the probability of selecting decision D^a for connection c, and $\tau(D_c^a)$ is the current amount of pheromone associated with D_c^a. Note that, in this ACO algorithm, we do not use any heuristic information, that is, the probability of selecting a solution component is solely dependent on the current pheromone amounts associated with each component.

If the selected component is D_c^{true}, that is, the ant selected the decision to include connection c in the NN structure, the corresponding connection $(n_i \rightarrow n_j)_c$ is appended to the candidate network NN (the **if** statement in lines 5–7). After the ant visits all the available connections in the construction graph and performs the include-or-not decision, the network structure of NN is now complete, and the weights of the neural network are ready to be learned. If a given neuron i either does not have an incoming path from any of the network inputs or does not have an outgoing path to any of the network outputs, then that neuron i is not included in the constructed network. If it happens that one of the network outputs does not have an incoming path from any of the network inputs, then the constructed network is assigned a poor quality evaluation without applying the BP weight-training process.

We train the neural network NN (line 9) using the back propagation (BP) procedure (described in Sect. 4.1), with some optimized parameter values (discussed in the following section), as a "quick and dirty" method to obtain a complete neural network and evaluate its pattern classification quality. We use BP for training the candidate neural network, not because it is the best weight optimization method, but because it is a fast procedure that is going to be repeated many times over the course of the computation. In addition, we are only interested in the relative quality difference between different NN structures trained by the same (even if not very efficient) BP procedure.

5.4 Evaluating the quality of a candidate solution

A key objective of a pattern classification algorithm is to learn models with good generalization capabilities, i.e., models that are able to accurately predict the class labels of *new* unknown data patterns. Overfitting occurs when the induced model has good classification performance (fit) on the training (in-sample) data used in the learning process, yet shows bad predictive performance (generalization) on new/testing data. Therefore, we split the training set $\mathcal{T}$ at the beginning of the algorithm into two mutually exclusive parts: 1) the learning set $\mathcal{T}_l$, which contains 80 % of the training set and is used to learn the candidate NN structure and weights (line 9, Algorithm 2); and 2) the validation set $\mathcal{T}_v$, which contains 20 % of the training set and is used to evaluate the quality of the model (line 10, Algorithm 1).

Let m denote the number of classes, $\hat{c}$ denote the true (correct) class for a given pattern x, and c denote the class that is predicted by the neural network NN (i.e., $c = \arg\max_i\{o_i\}$). Recall from Sect. 4 that we use y to denote the m-dimensional target output vector of the network, and use y' to denote the actual output vector, where $y' = (o_1, o_2, \ldots, o_m)$. The output vector can be transformed into a vector p of class probability scores through a simple normalization:

$$p_k = \frac{o_k}{\sum_{j=1}^{m} o_j} \tag{16}$$

A simple, and perhaps the most widely used, classification measure is accuracy. This is the quality measure that was used in previous work on ANN-Miner (Salama and Abdelbar 2014). For a given pattern x,

$$Acc(NN|x) = \begin{cases} 1 \text{ if } \hat{c} = c \\ 0 \text{ if } \hat{c} \neq c \end{cases} \tag{17}$$

For an entire validation set $\mathcal{T}_v$,

$$Q_{Acc}(NN|\mathcal{T}_v) = \frac{1}{|\mathcal{T}_v|} \sum_{x \in \mathcal{T}_v} Acc(NN|x) \tag{18}$$

where Q_{Acc} is the pheromone amount to be deposited in the pheromone update step (line 16, Algorithm 1); the higher the value of $Q_{Acc}(NN|\mathcal{T}_v)$, the better the quality of NN.

The accuracy measure has a deficiency, which can be illustrated using the following example. Suppose we have a pattern x where $m = 3$ and $\hat{c} = 1$. Consider three candidate NNs with the following probability score vectors given pattern x: $NN_1(x) = (0.9, 0.1, 0)$, $NN_2(x) = (0.6, 0.4, 0)$ and $NN_3(x) = (0.6, 0.2, 0.2)$. In the three NNs of the example, $Acc(x)$ will be equal to 1 for all three probability vectors. However, it is obvious that NN_1 should receive a better quality preference than NN_2 and NN_3, since it produces a higher probability for the true class.

In this work, we use the quadratic loss function (QLF), which is a widely used error measure, to evaluate the quality of constructed candidate NN models. For a given pattern x:

$$QLF(NN|x) = \sum_{k=1}^{m} \left[y_k - y'_k\right]^2 \tag{19}$$

Because components of the y vector are equal to 1 for the correct class and to 0 for all other classes, this equation can be rewritten as:

$$QLF(NN|x) = (1 - p_{\hat{c}})^2 + \sum_{k\in[1,m]:k\neq c} (p_k)^2 \tag{20}$$

In the same aforementioned example, the three probability vectors would have QLF values of: 0.02, 0.32, and 0.24. Thus, the QLF error measure would prefer NN_1, followed by NN_3, followed by NN_2. Thus, not only does QLF favor the models that produce a higher probability for the true class, but it also favors the models that produce the lowest probabilities for the other classes.

For an entire validation set $\mathcal{T}_v$,

$$Q_{QLF}(NN|\mathcal{T}_v) = 1 - \frac{1}{|\mathcal{T}_v|} \sum_{x\in\mathcal{T}_v} QLF(NN|x) \tag{21}$$

where Q_{QLF} is the pheromone amount to be deposited in the pheromone update step.

5.5 Updating pheromone trails

After the quality Q_i is computed for each candidate solution NN_i constructed by all the ants in the colony at iteration t, the iteration-best solution is identified and used to update the pheromone amounts on the construction graph. The pheromone amounts are increased on all the components D_c^a of the solution constructed by the iteration-best ant during its trail, where D_c^a represents the decision to include ($a = true$) or not to include ($a = false$) connection c in the NN structure. This influences the probability for the subsequent ants to include, or not to include connection c. The amount of pheromone deposited is based on Q_{tbest}, the quality of the iteration-best solution NN_{tbest}, as follows:

$$\tau(D_c^a) = \tau(D_c^a) + [\tau(D_c^a) \times Q_{tbest}] \forall c \in C, D_c^{true} \in NN_{tbest} \tag{22}$$

To simulate pheromone evaporation, normalization is then applied on each pair of solution components associated with each connection c in the construction graph. This keeps the total pheromone amount on each pair $\tau(D_c^{true})$ and $\tau(D_c^{false})$ equal to 1, as follows:

$$\tau(D_c^a) = \frac{\tau(D_c^a)}{\tau(D_c^{true}) + \tau(D_c^{false})} \quad \forall c \in C \tag{23}$$

6 Variations of the ANN-miner algorithm

6.1 Accumulated wisdom

We present two variations of the ANN-Miner algorithm: the first variation is the standard ANN-Miner algorithm described in Sect. 5, and the second is a variation called wANN-Miner that makes use of the optimized weights of the best-so-far network. In both variations, after each ant_i constructs a candidate network structure NN_i, the neural network is trained using the BP procedure to learn the weights of its connections.

In the standard ANN-Miner algorithm, after ant_i constructs a candidate network structure NN_i (where NN_i consists of a set of inter-neuronal connections without associated weights), the weights of NN_i are randomly initialized. This means that the algorithm does not make use of the optimized weights of previously constructed neural networks. Such an approach performs a fair comparison between different candidate NN structures, since they all start weight optimization from the same point: random initialization of the weights. In ANN-Miner, we perform BP for each candidate NN_i, for 20 epochs and with a learning rate of 0.1. Moreover, the weight-learning procedure in the post-processing step (described in the following subsection) also starts with randomly initialized weights.

In contrast, the second variation, called wANN-Miner, makes use of the optimized weights of the best-so-far neural network NN_{bsf} constructed in previous iterations. In other words, in wANN-Miner, the colony retains the weight optimization "wisdom," and accumulates on it throughout the algorithm's execution. Specifically, in this variation, NN_{bsf} consists of a set of inter-neuronal connections along with their associated weights. After each ant constructs a candidate network structure NN_i, the weights of its connections are not randomly initialized but rather are initialized with the weights present in NN_{bsf}. However, some connections in NN_i may not be present in NN_{bsf}, in which case their weights will be randomly initialized. Furthermore, there may be some connections in NN_{bsf} that are not present at all in NN_i. Such differences in the NN structures maintain the exploration aspect of the weight-learning process, in addition to the exploitation aspect that is realized by building on the best weights learned in previous iterations.

The back propagation procedure is then applied to NN_i; if NN_i produces a better classification quality than NN_{bsf}, NN_i will replace NN_{bsf}, and its connection weights will be used as initial values in constructing subsequent candidate neural networks.

In wANN-Miner, we perform BP, for each candidate NN_i, for only 10 epochs and with a lower learning rate of 0.05, making use of the accumulated weight optimization wisdom. Moreover, the BP weight-learning procedure in the post-processing step also starts with the weights of NN_{bsf}.

6.2 Post-processing procedure

The ANN-Miner algorithm performs a final step to learn the connection weights of the NN_{bsf} optimized structure produced by the ACO procedure. We use the two NN weight-learning algorithms discussed in Sect. 4:

1. the standard gradient-descent-based back propagation algorithm,
2. the ant-based $\text{ACO}_{\mathbb{R}}$ algorithm for continuous optimization.

Furthermore, we also evaluate baseline variations in which no post-processing is applied, and the network NN_{bsf} is returned without any further weight optimization. The idea behind that is to test the hypothesis that wANN-Miner may not benefit from the weight-learning post-processing step to the same extent that the first variation, ANN-Miner, may benefit. This is demonstrated in the results in Sect. 9.

7 Review of related neuroevolutionary methods

Evolving neural network topologies and weights has been of substantial interest to the evolutionary computation community since the 1990s. There are several useful surveys of this area: (Floreano et al. 2008) is a broad survey; (Schliebs and Kasabov 2013) focuses on evolving spiking neural networks; and (Risi and Togelius 2014) focuses on neuroevolution in games.

Some neuroevolutionary methods, including our own ANN-Miner, use a direct topology representation, in which the candidate solution representation includes decision variables to control the existence of every potential connection, and possibly every potential node. Other approaches use an indirect representation, for example an evolvable set of rules or a grammar, to indirectly specify the network topology. Numerous works have employed indirect encodings (Stanley et al. 2009; Hornby and Pollack 2002; Cangelosi et al. 1994; Kodjabachian and Meyer 1998; Stanley 2007; Clune et al. 2009; Valsalam and Miikkulainen 2011; Valsalam et al. 2012). We focus on direct encoding-based methods in this review, since they are more similar to our work.

We suggest a four-category classification of neuroevolutionary methods. The first two categories (which we will call Type I and II) are methods that evolve both the network topology and the weights, without the use of gradient descent. Type I methods are ones that employ some type of crossover operator, and Type II methods are ones that do not. Type III methods evolve the network topology, but use some type of gradient-descent approach to optimize the weights, in order to evaluate the fitness of each constructed topology. Type IV methods evolve the weights for a fixed-topology network. We will consider each of these four types in turn in the following subsections.

7.1 Type I methods

Because Type I methods employ some type of crossover operator, they face challenges not faced by Type II methods. One such challenge is what has been called the “competing conventions” problem (Whitley et al. 1993; Stanley and Miikkulainen 2002). This problem refers to there being multiple ways to express what is intuitively the same network. For example, the two networks in Fig. 1 are logically equivalent, but appear to be different. Although both have the same fitness, crossover between them could result in a very poor fitness solution.

Another problem that is shared by Type I and II methods is that when the network structure is changed through crossover or mutation, by adding or removing an edge or a node, the new network often has initially low fitness. After the weights have had some time to adapt to the new structure, the “true” fitness of the new structure can reveal itself. However, evolutionary pressure will often force the newly created structure to be removed from the population before its weights have had a chance to “catch-up.”

This problem is not faced by Type III methods, because they use gradient descent to quickly adapt the weights while computing the fitness of a structure. For Type I and II methods, there is a need to protect a newly created or newly modified structure until its weights have had a chance to adapt.

Prominent in the Type I family of methods is Stanley et al.’s NeuroEvolution through Augmenting Topologies (NEAT) algorithm (Stanley and Miikkulainen 2002; Stanley et al.

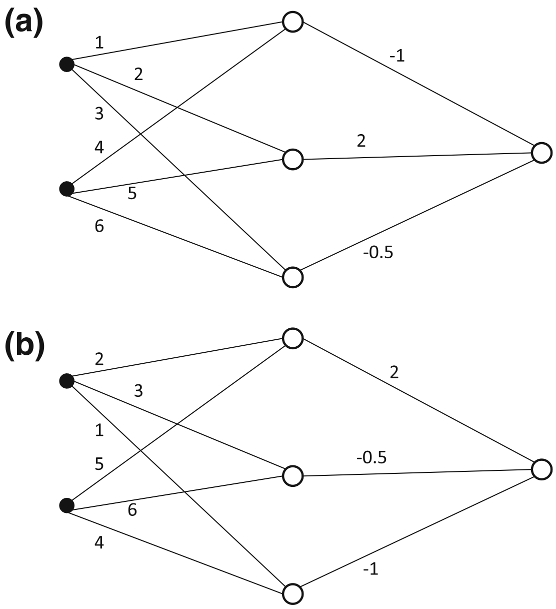

Fig. 1 A practical illustration of the competing conventions problems. The two networks shown are logically the same, but appear to be different

2005b), and its variations (Stanley 2007; Stanley et al. 2009; Whiteson et al. 2005). NEAT protects newly created structures through an elaborate mechanism for speciation or niching (Potter and De Jong 1995). NEAT also includes an interesting approach to the competing conventions problem (see below).

NEAT's solution representation, or *genome*, includes two types of *genes*: node genes and connection genes. Each node or potential node in the network has a corresponding node gene. NEAT starts off with a small network consisting only of the input neurons and output neurons, with no hidden neurons. Hidden neurons are then added gradually over the course of the computation. A connection gene specifies a single connection; the gene representation includes: the source node, the destination node, the weight, an "enable bit" that indicates whether or not that connection exists in the network, and an *innovation number* that we describe further below. NEAT's genome includes a list of node genes, and a list of connection genes.

Connection weights are adapted by mutation as in most evolutionary algorithms, with each connection being perturbed with some probability in each generation. Structural mutations, which affect the network topology, expand the genome by adding genes. There are two types of structural mutation. In one type, a single new connection gene with a random weight is added connecting two previously unconnected nodes. In another type, an existing connection between two nodes a and b is split into two connections, with a new node c being inserted in between. More specifically, the single connection from a to b with weight w is replaced with a connection from a to the new node c with weight 1, and a second connection from c to b with weight w.

Whenever a new gene is created through structural mutation, it is assigned a unique serial ID called the innovation number, which provides a mechanism for tracking the historical origin of a gene. The innovation number is immutable: it is not changed by mutation, and in the case of crossover, a gene crosses over with its innovation number intact.

The historical information captured by the innovation numbers provides a way to implement crossover in a way that minimizes the impact of the competing conventions problem. When crossover is to be performed on two parent genomes P and Q, the genes in both genomes with the same innovation numbers are identified and are called matching genes. In constructing the offspring, genes are randomly chosen from either parent at matching genes, while nonmatching genes are always taken from the more fit parent.

The percentage of matching genes between two genomes P and Q in the population can be used to produce a distance measure δ that measures the degree of similarity between P and Q. Genomes whose distance from one another is less than some compatibility threshold δ_t are taken to be members of the same *species*. To prevent overlap between species, in each generation, a genome is placed in the first species that it is found to be compatible with. In this way, the population can be partitioned in each generation into species, with crossover taking place almost exclusively within a species, although interspecies crossover is allowed with a very low probability.

To prevent a high-fitness species from dominating the entire population, a fitness sharing (Goldberg and Richardson 1987) mechanism is applied: If a genome P has an actual fitness of x, then its adjusted fitness is set to x/n where n is the number of members of P's species in the current generation. The number of offspring each species S is allocated in generation $(t+1)$ is determined in proportion to the sum of the adjusted fitness of the members of S in generation t. Each species S then first eliminates the least-fit members of S, then the surviving members reproduce to create S's assigned-share of the population of generation $(t+1)$.

NEAT has been applied primarily to reinforcement learning problems, including its use in games (Stanley and Miikkulainen 2004; Stanley et al. 2005, b)—although recent work (Sohangir et al. 2014) has applied it to classification. Extensions to NEAT include a real-time version (Stanley et al. 2005b), a variant called HyperNEAT which uses an indirect hypercube representation to specify the topology (Stanley et al. 2009), and a variation for evolving gene regulatory networks (Cussat-Blanc et al. 2015).

Also within the Type I Family is a hybrid approach (Yu et al. 2007) which used a PSO variation that includes crossover to optimize the weights and topology of single-hidden-layer feed-forward networks, and the work of others who used conventional genetic algorithms (Castillo et al. 2000; Whitley et al. 1993).

7.2 Type II methods

Type II methods avoid the competing conventions problem entirely by not employing crossover altogether, relying instead on mutation/perturbation operators. Evolutionary computation approaches within this family include evolutionary programming approaches (which use mutation alone without crossover) (Palmes et al. 2005; McDonnel and Waagen 1993; Gutiérrez et al. 2011; Fogel 1993; Ang et al. 2008; Fang and Xi 1997). Some used evolutionary programming approaches where the mutation parameters are controlled by an annealing temperature (Angeline et al. 1994; Leung et al. 2003), or by other heuristic measures (Oong and Isa 2011).

Several researchers (Chan et al. 2013; Yu et al. 2008) have used PSO approaches, without crossover, to optimize the weights and structure of single-hidden-layer feed-forward networks.

7.3 Type III methods

Type III methods use evolutionary computation (broadly defined) to evolve the network topology, but use some type of gradient descent to optimize the weights. This is the approach that we follow in the ANN-Miner algorithm. In such methods, fitness evaluation requires running a gradient-descent algorithm (such as back propagation) within the fitness function, typically starting from randomly initialized weights. Examples of this approach include several works (Yao and Liu 1997; Whitley et al. 1990; Martínez-Estudillo et al. 2005). Note that we include in this category methods which optimize weights using a combination of a gradient-based method and another method. For example, one approach (Yao and Liu 1997) used a combination of gradient descent and simulated annealing. Another approach (Martínez-Estudillo et al. 2005) used an elaborate scheme where evolutionary computation operators are employed to obtain an initial set of weights, which are then clustered, and the Levenberg–Marquardt (LM) gradient-based method is applied to the best of each cluster.

7.4 Type IV methods

Type IV methods evolve the weights of a fixed-topology neural network. The ACO$_{\mathbb{R}}$ (Socha and Dorigo 2008) algorithm can be considered to fall in this category. Other examples include numerous PSO-based approaches (Yeh 2013; Cai et al. 2010; Salerno 1997; Lu et al. 2003; Juang 2004; Settles et al. 2003; Dutta et al. 2013; Dehuri et al. 2012; Okada 2014; Song et al. 2007; Yeh et al. 2011; Han et al. 2011b; Lin et al. 2009), as well as methods that combine PSO with simulated annealing (Da and Xiurun 2005), in addition to approaches based on differential evolution (Ilonen et al. 2003; Garro et al. 2011) and cuckoo search (Valian et al. 2011; Nawi et al. 2013). Numerous conventional genetic algorithm approaches for evolving the weights of a fixed-topology network have also been explored (Gomez and Miikkulainen 1999; Saravanan and Fogel 1995; Yang and Kao 2001; Coshall 2009; Kang et al. 2010).

8 Experimental methodology

8.1 Comparative evaluation

In our experiments, we compare the predictive performance of several NN learning algorithms. First, as a baseline, we use the standard three-layer topology with back propagation (BP) for weight learning. This is referred to as 3L-BP. In addition, we also use the standard three-layer structure, but with ACO$_{\mathbb{R}}$ for weight optimization, and refer to it as 3L-ACO$_{\mathbb{R}}$. Furthermore, we use five variations of our proposed ant-based algorithm for optimizing NN structure (ANN-Miner, ANN-Miner-BP, wANN-Miner, wANN-Miner-BP, and wANN-Miner-ACO$_{\mathbb{R}}$). Each of our ANN-Miner variations is defined by: 1) whether it initializes the connection weights after each iteration or it memorizes the optimized weights throughout the algorithm, and 2) whether it uses a post-processing step of weight learning (and the algorithm utilized in this step) or not. Table 1 summarizes the NN learning algorithms used in the experiments.

Moreover, we implemented a Greedy Hill-Climbing (GHC) approach to learn NN structures, using back propagation (BP) as a subroutine to learn NN weights. This is referred to as GHC-BP. The algorithm starts with a maximally connected multilayer NN structure containing all the possible connections between the network neurons. Then the algorithm attempts to *prune* the NN structure using the first-improvement approach, as follows. Iteratively, GHC-

Table 1 Neural network learning algorithms used in the experiments

Algorithm	Abbreviation	Description
3L-BP	3L-BP	The standard three-layer structure
		Uses BP for learning weights
3L-ACO$_{\mathbb{R}}$	3L-ACO$_{\mathbb{R}}$	The standard three-layer structure
		Uses ACO$_{\mathbb{R}}$ for learning weights
ANN-Miner	ANN	No post-processing step
		Randomly initializes weights each iteration
ANN-Miner-BP	ANN-BP	Uses BP as a post-processing step
		Randomly initializes weights each iteration
wANN-Miner	wANN	No post-processing step
		Uses the weights of the best-so-far NN as the initial weights
wANN-Miner-BP	wANN-BP	Uses BP as a post-processing step
		Uses the weights of the best-so-far NN as the initial weights
wANN-Miner-ACO$_{\mathbb{R}}$	wANN-ACO$_{\mathbb{R}}$	Uses ACO$_{\mathbb{R}}$ as a post-processing step
		Uses the weights of the best-so-far NN as the initial weights
Greedy search	GHC-BP	Uses BP for learning weights
		Prunes on the multilayer fully connected NN structure

Algorithm 3 Pseudo-code of GHC-BP.

```
1: Begin
2: $NN_{best} = input$;
3: $size = GetConnectionCount(NN_{best})$;
4: $Q_{best} = EvaluateQuality(NN_{best})$;
5: for $i = 1 \rightarrow size$ do
6:     if $i >$ max_evaluations then
7:         break;
8:     end if
9:     $NN_{current} = Remove(NN_{best}, Connection_i)$;
10:    $Q_{current} = EvaluateQuality(NN_{current})$;
11:    if $Q_{current} >= Q_{best}$ then
12:        $NN_{best} = NN_{current}$;
13:        $Q_{best} = Q_{current}$;
14:    end if
15: end for
16: $NN_{final} = PostProcessing(NN_{best})$;
17: return $NN_{final}$;
18: End
```

BP temporarily removes one connection from the NN structure, learns the weights of the new NN using BP, and evaluates its quality. If the quality improves (or does not change), this connection is removed permanently from the NN structure; otherwise the connection is returned back to the structure. Such an algorithm allows us to examine the effect of using the ACO meta-heuristic as a global search for optimizing the NN structures in comparison with using a greedy local search. The pseudo-code of GHC-BP is presented in Algorithm 3.

Table 2 Parameter settings used in experiments

Algorithm	Parameter	Role	Value
ACO	`max_iterations`	Max # of iterations	500
	`colony_size`	# of solutions created per iteration	10
	`conv_iterations`	Max # of nonimproving iterations	10
Back propagation	Learning rate η	Post-processing	0.01
		ANN candidate NN training	0.1
		wANN candidate NN training	0.05
	Epochs	Post-processing	1000
		ANN candidate NN training	20
		wANN candidate NN training	10
$\text{ACO}_{\mathbb{R}}$	m	# of ants per iteration	1
	ξ	Controls speed of convergence	0.85
	q	Controls locality of search	10^{-4}
	L	# of solutions in the archive	50
Greedy hill-climbing	`max_evaluations`	Max # of solution evaluations	(`max_iterations` × `colony_size`)

8.2 Experimental setup

The experiments were carried out using the *stratified* 10-times 10-fold cross-validation procedure. In essence, a dataset is divided into 10 mutually exclusive partitions (folds), with approximately the same number of patterns in each partition. Then each classification algorithm is run 10 times, where each time a different partition is used as the test set and the other nine partitions are used as the training set. The results are then averaged and reported as the accuracy rate of the NN classifier. Since we are evaluating stochastic algorithms, we run each 10 times—using a different random seed to initialize the search each time—for each of the 10 iterations of the cross-validation procedure.

The parameter configuration used in our experiments is shown in Table 2. For the sake of fairness of comparison, we limit each algorithm to the same fixed number of solution evaluations to construct a final NN classifier. In GHC-BP (Algorithm 3, line 6), the external parameter `max_evaluations` represents the maximum number of solution evaluations that the algorithm performs during the hill-climbing search. As can be seen in Table 2, it is set equal to `max_iterations` multiplied by `colony_size`, which is the maximum number of evaluations for ANN-Miner. However, note that the maximum number of evaluations might not be utilized completely. The ACO-based algorithms might use a smaller number of iterations if they converge earlier and the greedy algorithm might also stop earlier if the total number of connections in the fully connected NN structure (with all the connections) is less than `max_evaluations`.

The performance of ANN-Miner was evaluated using 40 public-domain datasets from the well-known UCI (University of California at Irvine) dataset repository (Asuncion and Newman 2007). The main characteristics of the datasets are shown in Table 3.

In follow-up experiments, described in Sects. 9.4 and 9.5, we compare our approach to a number of well-known state-of-the-art and baseline classifiers, and to NEAT, a prominent neuroevolutionary technique.

9 Computational results

9.1 Predictive accuracy

Predictive accuracy results are reported in Table 4 for each of the algorithms under evaluation. These results represent the average predictive accuracy over 100 runs of the 10-times 10-fold cross-validation procedure described in Sect. 8, for each of the 40 datasets. For each dataset, the highest accuracy is shown in boldface. In addition, the last row of the table reports the average rank of each algorithm. For each algorithm g, the rank of g is first obtained for each dataset individually, and then the individual dataset ranks are averaged across the 40 datasets for each algorithm. In case two or more algorithms are tied for a given dataset, then the tied algorithms are given the average of the ranks that they span.

From the table, we note that the best-ranking algorithm is wANN-ACO$_{\mathbb{R}}$ with a rank of 1.88, followed closely by wANN-BP with a rank of 2.28. These are followed by ANN-BP in third place with a rank of 3.54, then by wANN in fourth place with a rank of 4.15. In fifth place is 3L-BP with a rank of 5.63. Finally, in the last three places, respectively, are ANN with a rank of 6.06, 3L-BP with a rank of 6.23, and the greedy GHC-BP with a rank of 6.25.

wANN-ACO$_{\mathbb{R}}$ had the highest predictive accuracy in 22 of the 40 datasets, and wANN-BP had the highest accuracy in 20 datasets. wANN and ANN-BP had the highest accuracy in 6 and 5 datasets, respectively. 3L-BP and 3L-ACO$_{\mathbb{R}}$ each had the highest accuracy in a single dataset, while ANN and GHC-GP did not have the highest accuracy in any datasets.

Table 5 reports the results of applying a nonparametric Friedman test with the Holm post-hoc test (Derrac et al. 2011), at the conventional 0.05 threshold, to compare all pairings of the eight algorithms under evaluation. The Friedman statistic χ_F^2 is found to be 150.9 with seven degrees of freedom, corresponding to a p value of 9E$-$11. Thus, we can reject the null hypothesis and proceed with the post-hoc tests. For each pairing, we report the computed p value, and the corresponding Holm critical value. The difference between the two algorithms is statistically significant if the p value is less than or equal to the corresponding Holm threshold. Statistically significant p values are shown in boldface. We observe the following:

- wANN-ACO$_{\mathbb{R}}$ is significantly better than all the other algorithms, except for wANN-BP—which differs from it only in the type of post-processing that is employed.
- The use of post-processing always results in a statistically significant improvement: wANN-ACO$_{\mathbb{R}}$ and wANN-BP are both significantly better than wANN; ANN-BP is significantly better than ANN.
- The use of “wisdom” results in a statistically significant improvement in accuracy when combined with ACO$_{\mathbb{R}}$ post-processing, but not when combined with BP post-processing: wANN-ACO$_{\mathbb{R}}$ is significantly better than ANN-ACO$_{\mathbb{R}}$, but no statistically significant

Table 3 Characteristics of the datasets used in the experiments

Dataset	Instances	Classes	Attributes		
			Total	Numeric	Categorical
annealing	896	6	38	9	29
automobile	205	7	25	15	10
balance	625	3	4	0	4
breast-l	283	2	9	0	9
breast-p	198	2	32	32	0
breast tissue	106	6	9	9	0
breast-w	569	2	30	30	0
car	1728	4	6	0	6
chess	3196	2	36	0	36
credit-a	690	2	14	6	8
credit-g	1000	2	20	7	13
cylinder	540	2	35	19	16
dermatology	366	6	34	1	33
ecoli	336	8	7	7	0
glass	214	7	9	9	0
hay	132	3	4	0	4
heart-c	303	5	13	7	6
heart-h	293	5	13	7	6
hepatitis	155	2	19	6	13
horse	366	2	22	7	15
ionosphere	351	2	34	34	0
iris	150	3	4	4	0
liver disorders	345	2	6	6	0
lymphography	148	4	18	3	15
monks	556	2	6	0	6
nursery	12,960	5	8	0	8
parkinsons	195	2	22	22	0
pima	768	2	8	8	0
s-heart	270	2	13	6	7
segmentation	2273	7	19	19	0
soybean	307	19	35	0	35
thyroid	215	3	5	5	0
transfusion	722	2	4	4	0
ttt	958	2	9	0	9
vehicle	846	4	18	18	0
vertebral-column-2c	310	2	6	6	0
vertebral-column-3c	310	3	6	6	0
voting	425	2	16	0	16
wine	178	3	13	13	0
zoo	101	7	16	0	16

Table 4 Predictive accuracy (%) results for the algorithms under evaluation

Dataset	3L-BP	3L-ACO$_{\mathbb{R}}$	ANN	ANN-BP	wANN	wANN-BP	GHC-BP	wANN-ACO$_{\mathbb{R}}$
`annealing`	67.41	70.83	60.97	79.14	78.08	**83.21**	69.02	81.39
`automobile`	34.73	48.83	45.07	48.95	58.11	**60.49**	43.42	58.31
`balance`	**96.16**	95.50	90.50	94.50	91.33	94.00	90.33	95.83
`breast-l`	66.93	69.98	54.25	69.89	71.40	72.55	60.15	**72.95**
`breast-p`	67.79	70.21	65.76	72.68	73.29	73.29	73.28	**75.76**
`breast tissue`	33.81	39.36	41.63	56.46	54.10	**64.27**	48.18	59.27
`breast-w`	93.86	83.45	94.38	94.73	**95.43**	**95.43**	91.57	**95.43**
`car`	90.70	91.20	89.47	93.94	**98.19**	**98.19**	95.26	95.59
`chess`	92.61	88.70	81.82	88.64	95.12	98.58	95.75	**98.98**
`credit-a`	84.35	**84.50**	83.48	84.35	82.75	83.19	79.27	**84.50**
`credit-g`	72.20	70.60	71.90	72.00	71.90	**72.40**	70.40	72.20
`cylinder`	65.05	69.11	69.45	69.78	71.07	73.17	68.73	**73.74**
`dermatology`	80.68	81.75	86.80	**93.16**	93.05	**93.16**	85.60	**93.16**
`ecoli`	79.53	75.04	81.25	84.86	84.86	**85.76**	82.76	85.22
`glass`	46.36	48.98	48.87	**57.88**	48.46	50.84	42.36	52.57
`hay`	60.01	63.45	63.02	69.31	71.45	**71.59**	61.75	71.06
`heart-c`	55.46	55.93	51.44	60.04	53.46	58.72	49.62	**61.77**
`heart-h`	57.43	62.66	62.63	62.66	60.22	**63.29**	59.03	**63.29**
`hepatitis`	79.42	80.10	79.46	80.75	**81.92**	**81.92**	80.04	**81.92**
`horse`	76.04	78.00	77.71	80.50	80.32	81.81	74.63	**82.00**
`ionosphere`	88.23	86.65	89.67	92.52	92.81	**93.38**	91.62	93.30
`iris`	85.33	88.49	92.52	92.81	93.38	**93.66**	91.95	92.93

Table 4 continued

Dataset	3L-BP	3L-ACO$_{\mathbb{R}}$	ANN	ANN-BP	wANN	wANN-BP	GHC-BP	wANN-ACO$_{\mathbb{R}}$
`liver disorders`	57.40	58.32	64.64	**65.26**	64.08	64.08	63.52	64.62
`lymphography`	78.42	77.54	74.28	77.81	74.94	**79.88**	73.81	**79.88**
`monks`	61.77	63.25	41.56	46.75	42.28	60.18	57.03	**63.43**
`nursery`	91.15	93.19	87.72	91.76	96.62	97.93	92.52	**98.78**
`parkinsons`	79.94	79.15	80.55	82.05	81.50	83.05	82.01	**83.89**
`pima`	73.82	73.02	74.86	76.04	75.13	**76.17**	74.73	75.18
`s-heart`	81.11	82.92	84.45	**85.56**	81.85	83.70	79.26	83.71
`segmentation`	93.01	88.40	88.81	92.77	92.74	**93.13**	92.50	92.92
`soybean`	57.58	57.58	54.48	58.79	68.28	**76.27**	54.66	65.17
`thyroid`	84.63	86.31	79.63	90.28	84.56	89.37	84.19	**91.08**
`transfusion`	70.56	72.45	73.19	73.23	71.77	72.60	72.04	**73.35**
`ttt`	76.63	78.05	50.32	90.94	97.89	98.00	86.00	**98.21**
`vehicle`	64.21	64.00	59.69	60.06	**72.18**	**72.18**	64.35	**72.18**
`vertebral-column-2c`	79.35	79.35	80.00	**83.55**	80.97	81.61	82.58	82.51
`vertebral-column-3c`	68.06	66.45	69.32	74.84	66.29	74.19	64.19	**74.90**
`voting`	93.89	93.75	93.24	94.65	**94.82**	**94.82**	92.87	**94.82**
`wine`	90.41	93.53	94.93	96.04	93.27	94.38	89.41	**96.31**
`zoo`	81.25	88.57	89.11	90.54	**95.50**	**95.50**	85.35	94.89
rank (avg)	6.23	5.63	6.06	3.54	4.15	2.28	6.25	**1.88**

Table 5 Results of the Friedman test with the Holm post-hoc test, at the 0.05 significance threshold, for the predictive accuracy results reported in Table 4

#	Comparison	p	Holm
1.	wANN-ACO$_{\mathbb{R}}$ versus GHC-BP	**1E−15**	0.00179
2.	wANN-ACO$_{\mathbb{R}}$ versus 3L-BP	**2E−15**	0.00185
3.	wANN-ACO$_{\mathbb{R}}$ versus ANN	**2E−14**	0.00192
4.	wANN-BP versus GHC-BP	**4E−13**	0.002
5.	wANN-BP versus 3L-BP	**6E−13**	0.00208
6.	wANN-BP versus ANN	**5E−12**	0.00217
7.	wANN-ACO$_{\mathbb{R}}$ versus 3L-ACO$_{\mathbb{R}}$	**8E−12**	0.00227
8.	wANN-BP versus 3L-ACO$_{\mathbb{R}}$	**1E−09**	0.00238
9.	ANN-BP versus GHC-BP	**7E−07**	0.0025
10.	ANN-BP versus 3L-BP	**9E−07**	0.00263
11.	ANN-BP versus ANN	**4E−06**	0.00278
12.	wANN-ACO$_{\mathbb{R}}$ versus wANN	**3E−05**	0.00294
13.	wANN versus GHC-BP	**1E−04**	0.00313
14.	ANN-BP versus 3L-ACO$_{\mathbb{R}}$	**1E−04**	0.00333
15.	wANN versus 3L-BP	**2E−04**	0.00357
16.	wANN versus ANN	**5E−04**	0.00385
17.	wANN-BP versus wANN	**6E−04**	0.00417
18.	wANN-ACO$_{\mathbb{R}}$ versus ANN-BP	**0.002**	0.00455
19.	wANN versus 3L-ACO$_{\mathbb{R}}$	0.007	0.005
20.	wANN-BP versus ANN-BP	0.021	0.00556
21.	3L-ACO$_{\mathbb{R}}$ versus GHC-BP	0.254	0.00625
22.	ANN-BP versus wANN	0.263	0.00714
23.	3L-ACO$_{\mathbb{R}}$ versus 3L-BP	0.273	0.00833
24.	3L-ACO$_{\mathbb{R}}$ versus ANN	0.424	0.01
25.	wANN-ACO$_{\mathbb{R}}$ versus wANN-BP	0.465	0.0125
26.	ANN versus GHC-BP	0.732	0.01667
27.	ANN versus 3L-BP	0.767	0.025
28.	3L-BP versus GHC-BP	0.964	0.05

improvement was detected for wANN-BP over ANN-BP. Without post-processing: wANN is significantly better than ANN.

9.2 Model size

It is also interesting to consider the model size, expressed as the number of connections in the neural network, produced by each of the algorithms under evaluation. The model size of the baseline 3L-BP and 3L-ACO$_{\mathbb{R}}$ algorithms is of course fixed, and is shown in the second column of Table 6 (under the heading 3L). For each of the other algorithms, the table reports the ratio of the average number of connections (averaged over the 100 runs of the 10-times 10-fold cross-validation procedure) to the number of connections reported for 3L. The final row reports the average ratio for each algorithm. The network size, of course, does not depend on the type of post-processing that is employed, if any. Therefore, we report the network size results for ANN-Miner, which will be the same for ANN-Miner-BP. Similarly,

Table 6 Model size (expressed as number of inter-neuronal connections) results for the algorithms under evaluation

Dataset	3L	ANN	wANN	GHC-BP
annealing	10,404	1.12	1.13	1.32
automobile	6724	1.17	1.19	1.34
balance	529	1.15	1.17	1.34
breast-l	2809	1.12	1.13	1.30
breast-p	1156	1.10	1.12	1.30
breast tissue	225	1.23	1.24	1.42
breast-w	1024	1.15	1.18	1.30
car	625	1.17	1.17	1.36
chess	5625	1.14	1.13	1.29
credit-a	2025	1.06	1.12	1.30
credit-g	4225	1.13	1.13	1.29
cylinder	6561	1.12	1.12	1.29
dermatology	18,769	1.01	1.12	1.31
ecoli	225	1.22	1.23	1.41
glass	256	1.23	1.23	1.42
hay	324	1.14	1.19	1.33
heart-c	784	1.19	1.19	1.37
heart-h	784	1.14	1.19	1.37
hepatitis	1156	1.02	1.12	1.30
horse	3969	1.02	1.10	1.29
ionosphere	1296	1.12	1.12	1.30
iris	49	1.12	1.15	1.31
liver disorders	64	1.18	1.22	1.28
lymphography	2601	1.11	1.14	1.33
monks	361	1.14	1.18	1.31
nursery	1024	1.21	1.22	1.36
parkinsons	576	1.13	1.13	1.30
pima	100	1.16	1.16	1.27
s-heart	729	1.12	1.14	1.30
segmentation	676	1.20	1.20	1.42
soybean	13,689	1.20	1.21	1.39
thyroid	64	1.17	1.21	1.32
transfusion	36	1.20	1.24	1.22
ttt	841	1.19	1.19	1.30
vehicle	484	1.13	1.20	1.36
vertebral-column-2c	64	1.22	1.19	1.26
vertebral-column-3c	81	1.23	1.23	1.33
voting	1156	0.94	1.08	1.30
wine	256	1.05	1.08	1.33
zoo	1849	1.09	1.14	1.38
avg	–	1.14	1.17	1.33

we report the size results for wANN-Miner, which will be the same for wANN-Miner-BP and wANN-Miner-ACO$_{\mathbb{R}}$.

From the table, we note that the average size ratio for wANN and ANN is only slightly larger than the baseline 3L—a ratio of 1.17 for wANN and 1.14 for ANN. The largest ratio (1.34) is obtained with the greedy GHC-BP.

9.3 Discussion

In this subsection, we perform a more detailed analysis of the results by comparing the effectiveness of different aspects of the proposed algorithm in improving the predictive quality of the produced NN classification models, as follows:

- *The "wisdom"-based variation:* We note that the "wisdom"-based versions of ANN-Miner produce better accuracy over the corresponding standard versions. wANN-BP has better accuracy than ANN-BP in 28 out of the 40 datasets, and wANN has better accuracy than ANN in 31 out of the 40 datasets. The two versions of wANN-Miner with post-processing (either with BP or ACO$_{\mathbb{R}}$) have better predictive accuracy average ranks than the other algorithms under comparison. wANN-ACO$_{\mathbb{R}}$ is significantly better, in predictive accuracy, than each of the other algorithms, except for wANN-BP.
- *Weight-learning post-processing:* It is interesting to consider whether wANN-Miner benefits from weight-learning post-processing to the same extent as ANN-Miner. wANN-BP has better accuracy than wANN in 32 datasets, worse in zero datasets, and the same in eight datasets. On the other hand, ANN-BP has better accuracy than ANN in all 40 datasets, without any ties. Thus, both variations benefit from post-processing (which is hardly surprising), but the accumulative wisdom variation benefits slightly less. Regarding the weight-learning algorithm used in the post-processing step: wANN-ACO$_{\mathbb{R}}$ has better accuracy than wANN-BP in 20 datasets, worse in 13 datasets, and the same in 7 datasets.
- *ACO versus greedy search:* Comparing wANN-BP to the greedy GHC-BP, we find that wANN-BP has better accuracy in 39 datasets, and worse in a single dataset. All the variations of ANN-Miner that include post-processing (i.e., wANN-ACO$_{\mathbb{R}}$, wANN-BP, and ANN-BP) had better accuracy than GHC-BP to a statistically significant extent. Even without post-processing, wANN had significantly better accuracy than GHC-BP. The only variation for which a statistically significant improvement over GHC-BP was not detected was ANN. This is in spite of GHC-BP producing larger networks than all of the ANN-Miner variations in 39 out of the 40 datasets.

Of course, all versions of ANN-Miner require much more time in the training phase than a simple fixed-topology neural network. However, this affects only the training phase, which usually takes place off-line before an application is deployed, and does not affect the operating phase. In many applications, the time consumed by the training phase is not important compared to the predictive accuracy of the operating phase.

The time consumed by a neural network in the operating phase is a function of the number of connections in the network. Table 6 indicates that the difference in network size between, for example, wANN and the fixed three-layer topology is somewhat modest—ranging from 8 % larger to 24 % larger, with an average of 17 % larger.

9.4 Comparison to state-of-the-art classifiers

As a follow-up experiment, we compare the best two approaches in Table 4, namely wANN-BP and wANN-ACO$_{\mathbb{R}}$, to several well-established strong classifiers:

– the Ripper classification rule induction algorithm using its Weka (Witten et al. 2010) implementation JRip;
– the C4.5 decision tree induction algorithm using its Weka implementation J48;
– two versions of the support vector machine (SVM) classifier: the quadratic-kernel-based Weka SVM implementation SMO, and the Gaussian kernel-based C-language LibSVM (Chang and Lin 2011) implementation;

as well as to two well-known baseline classifiers:

– the one-nearest-neighbor algorithm using its Weka implementation IB1;
– the Naïve-Bayes classifier using its Weka implementation NB.

A recent large-scale empirical study (Fernández-Delgado et al. 2014) used 121 datasets to compare 179 classifiers, representing 17 classifier families and concluded that one of the most effective families was support vector machines, and that the LibSVM implementation with Gaussian kernels in particular was one of the most effective classifiers in general.

We applied each of these six algorithms (LibSVM, SMO, JRip, J48, IB1, NB) to the 40 datasets used in our experiments, with stratified 10-fold cross-validation, as described in Sect. 8, using the same fold partitioning used in the other experiments in this paper. We used Weka's default parameters for the five Weka implementations and used LibSVM's default parameters for LibSVM.

The results are shown in Table 7; note that the results for wANN-BP and wANN-ACO$_{\mathbb{R}}$ are repeated for convenience from Table 4. The last row of the table indicates the average rank of each algorithm. We observe that the best average rank was obtained, not surprisingly, by LibSVM (the SVM implementation with Gaussian kernels), followed in second place by SMO (the SVM implementation with quadratic kernels). These were followed by wANN-ACO$_{\mathbb{R}}$ in third place, J48 in fourth place, and wANN-BP in fifth place.

Table 8 reports the results of applying a nonparametric Freidman test with the Holm post-hoc test, at the conventional 0.05 threshold, to compare wANN-ACO$_{\mathbb{R}}$ (which is treated as the control algorithm) to each of the other algorithms. The Freidman statistic χ_F^2 is determined to be 43.6 with seven degrees of freedom, corresponding to a p value of 2E−7. Thus, we can reject the null hypothesis and proceed with the post-hoc tests. For each comparison, we report the computed p value and the corresponding Holm critical value. Statistical significance is detected if the p value is less than or equal to the corresponding Holm threshold. Statistically significant p values are shown in boldface. We observe that:

– LibSVM is significantly better than ANN-ACO$_{\mathbb{R}}$;
– No statistically significant difference is detected between ANN-ACO$_{\mathbb{R}}$ and any of the other algorithms.

The reader should note that the motivation behind the experiment described in this subsection is only to show that the arbitrary-topology feed-forward neural networks evolved by ANN-Miner have strong predictive accuracy performance (i.e., performance that is similar to that of widely used classifiers). We recognize that what we are comparing is a feed-forward neural network whose topology has been specifically optimized for each dataset, to classifiers such as J48 and SMO with default parameter values that have not been specifically optimized for each dataset. (However, of course, note that the parameters of the ANN-Miner algorithm itself (and its variations) are also general default parameters, and have not been specifically optimized for each dataset.) Each run of ANN-Miner (or its variations) constructs and evaluates up to 5000 neural networks (see Table 2). In the case of the better-performing wANN-Miner variation, each constructed network is trained for 10 epochs, for a total of up to 50,000 epochs. Run-time will therefore generally be much greater than any of the classifiers

Table 7 Predictive accuracy (%) results for two of our proposed algorithms (repeated from Table 4) along with several well-known classifiers

Dataset	wANN-BP	wANN-ACO$_{\mathbb{R}}$	libSVM	SMO	JRip	J48	IB1	NB
`annealing`	83.21	81.39	88.13	85.97	94.27	92.46	**94.40**	76.93
`automobile`	60.49	58.31	70.74	68.74	68.69	**81.36**	73.64	58.53
`balance`	94.00	**95.83**	91.67	90.83	72.83	63.83	74.50	92.83
`breast-l`	72.55	72.95	**74.33**	71.68	69.44	72.86	65.99	71.25
`breast-p`	73.29	75.76	**77.79**	76.29	75.79	71.08	67.18	64.58
`breast tissue`	64.27	59.27	60.55	59.64	58.55	65.37	**70.09**	67.18
`breast-w`	95.43	95.43	97.02	**97.90**	94.20	94.91	95.61	93.50
`car`	**98.19**	95.59	96.90	93.51	87.66	92.98	61.81	85.91
`chess`	98.58	98.98	97.92	95.72	99.00	**99.47**	84.53	88.05
`credit-a`	83.19	84.50	84.49	84.93	85.51	**85.80**	81.02	77.10
`credit-g`	72.40	72.20	**77.00**	73.90	72.20	69.60	71.50	75.20
`cylinder`	73.17	73.74	**76.36**	73.22	64.29	74.50	68.75	66.70
`dermatology`	93.16	93.16	**97.54**	96.43	88.01	94.00	94.53	97.54
`ecoli`	85.76	85.22	**87.25**	83.35	81.86	83.66	81.31	85.47
`glass`	50.84	52.57	**71.19**	58.46	66.54	68.50	68.90	49.52
`hay`	71.59	71.06	66.15	76.16	**79.23**	65.39	63.08	73.08
`heart-c`	58.72	**61.77**	57.81	56.43	54.15	51.21	52.52	56.42
`heart-h`	63.29	63.29	67.10	**67.77**	63.72	66.73	46.55	65.37
`hepatitis`	81.92	81.92	85.04	**85.71**	78.13	80.16	83.12	83.17
`horse`	81.81	82.00	**84.44**	81.51	83.54	82.75	79.05	77.96
`ionosphere`	93.38	93.30	**93.96**	88.50	89.66	88.26	87.38	83.04
`iris`	93.66	92.93	**96.67**	96.67	92.00	94.67	95.33	95.33

Table 7 continued

Dataset	wANN-BP	wANN-ACO$_{\mathbb{R}}$	libSVM	SMO	JRip	J48	IB1	NB
`liver disorders`	64.08	64.62	**69.23**	58.56	66.34	64.56	60.87	55.89
`lymphography`	79.88	79.88	81.24	**85.19**	79.95	76.38	79.10	82.47
`monks`	60.18	63.43	**65.45**	63.64	60.36	61.46	56.73	62.73
`nursery`	97.93	**98.78**	98.40	93.12	96.93	97.18	86.75	90.37
`parkinsons`	83.05	83.89	88.21	87.16	88.76	88.16	**93.42**	70.13
`pima`	76.17	75.18	76.81	**76.81**	73.55	72.51	70.31	75.64
`s-heart`	83.70	83.71	84.07	84.45	78.52	75.56	73.33	**84.82**
`segmentation`	93.13	92.92	93.77	92.60	94.10	95.59	**95.60**	79.94
`soybean`	76.27	65.17	**89.66**	88.62	83.10	83.11	88.62	88.97
`thyroid`	89.37	91.08	95.80	88.88	92.53	91.15	96.23	**96.71**
`transfusion`	72.60	73.35	73.76	71.75	73.71	**73.78**	61.55	70.07
`ttt`	98.00	98.21	83.26	**98.42**	98.00	85.58	67.37	70.63
`vehicle`	72.18	72.18	**77.54**	75.06	68.08	72.93	69.26	45.39
`vertebral-column-2c`	81.61	82.51	**84.84**	79.03	82.58	81.29	80.97	78.39
`vertebral-column-3c`	74.19	74.90	82.90	76.78	80.32	78.71	78.71	**83.55**
`voting`	**94.82**	**94.82**	94.52	92.97	93.66	94.48	88.73	85.94
`wine`	94.38	96.31	98.27	**98.82**	92.19	93.30	94.94	97.19
`zoo`	95.50	94.89	95.00	**98.75**	95.00	97.50	**98.75**	93.75
rank (avg)	4.71	4.53	**2.46**	3.91	4.93	4.56	5.51	5.39

Table 8 Results of a Friedman test with the Holm post-hoc test applied to the results of Table 7, using wANN-ACO$_{\mathbb{R}}$ as the control method

Comparison	p	Holm
LibSVM versus wANN-ACO$_{\mathbb{R}}$	**2E−4**	0.0071
wANN-ACO$_{\mathbb{R}}$ versus IB1	0.0714	0.0083
SMO versus wANN-ACO$_{\mathbb{R}}$	0.2635	0.01
wANN-ACO$_{\mathbb{R}}$ versus JRip	0.4652	0.0125
wANN-ACO$_{\mathbb{R}}$ versus wANN-BP	0.7321	0.0167
wANN-ACO$_{\mathbb{R}}$ versus NB	0.8847	0.25
wANN-ACO$_{\mathbb{R}}$ versus J48	0.9454	0.05

that we consider in this experiment. Again, the purpose of this experiment is only to show that ANN-Miner is an effective method for evolving neural networks—i.e., that it is capable of evolving neural networks whose predictive accuracy is competitive with established techniques.

9.5 Comparison with NEAT

As a second follow-up experiment, we compare the results of our approach to the NEAT (Stanley and Miikkulainen 2002) neuroevolutionary algorithm. As described in Sect. 7, NEAT is a sophisticated algorithm that includes the idea of dividing the population into species, with most recombination (or “breeding”) occurring among members of the same species. NEAT is also one of the few neuroevolutionary methods whose source code is publicly available. In fact, NEAT has several publicly available implementations (which can be found at K. Stanley’s NEAT website (Stanley 2015)), including at least four in C++, at least two in Java, in addition to implementations in Matlab, C$^\sharp$, and Python. We used Ugo Vierucci’s Java implementation, which was based on K. Stanley’s original C++ implementation.

With the exception of population size and number of generations (discussed further below), we used the default parameter settings that were included in NEAT’s source code distribution, which are consistent with the parameter settings used by Stanley and Miikkulainen (2002). In order for the comparison between NEAT and ANN-Miner to be fair, we wanted both techniques to have the same computational budget. In other words, we wanted the total number of constructed networks to be the same for each method. For ANN-Miner and its variations, the total number of constructed networks is the colony size multiplied by the maximum number of iterations. As Table 2 indicates, we used a colony size of 10 and a maximum number of iterations of 500, which means that the total number of constructed networks is limited to no more than 5000. Stanley and Miikkulainen (2002) used a population size of 150 and a number of generations of 24, for a total of 3600 fitness evaluations. To equalize the computational budget of the two methods, we kept NEAT’s population size at 150 and increased the number of generations to 34, for a total of 5100 fitness evaluations. For each dataset, we also set NEAT’s maximum allowable number of hidden neurons to be the same as the maximum number of hidden neurons for ANN-Miner.

Further, for the sake of fairness of comparison, we added a BP post-processing step to NEAT. This means that the final network produced by NEAT underwent 1000 epochs of BP using the BP post-processing parameter settings shown in Table 2. The BP post-processing step in NEAT-BP is identical to the post-processing that is applied in ANN-BP and wANN-BP.

NEAT-BP was applied to the 40 datasets used in our experiments, with stratified 10-times 10-fold cross-validation, as described in Sect. 8, using the same fold partitioning used in the other experiments in this paper. This means that NEAT-BP was run a total of 100 times for each dataset.

Table 9 reports the predictive accuracy results for NEAT-BP; for convenience, we also repeat the predictive accuracy results for two ANN variations (ANN-BP and ANN), and for two baseline classifiers (1NN, and NB). For NEAT-BP and ANN-BP, the table also reports for each dataset the number of inter-neuronal connections, expressed as a multiple of the number of connections in the baseline BP fixed-topology three-layer network (reported in Table 6 in the column labeled 3L). For example, for the `annealing` dataset, NEAT had an average size (number of inter-neuronal connections) that was 0.43 of the size (i.e., slightly less than half the size) of the baseline fixed-topology three-layer network, while ANN-BP had an average size that was 1.12 of the size of (i.e., slightly larger than) the baseline three-layer topology. The size for ANN is of course always the same as for ANN-BP, and is therefore not shown. The last row reports the average rank for the predictive accuracy columns, and the average size ratio for the model size columns.

We can make several observations regarding Table 9. We observe that the average number of connections is generally much smaller for NEAT than for ANN-Miner. For example, the number of connections for NEAT is less than a quarter of the connections for ANN for the `cylinder` dataset, but is almost equal for the `car` dataset; on average, over all datasets, the ratio of the number of connections for ANN to NEAT is 1.70.

However, NEAT's smaller model size came at the expense of predictive accuracy. ANN-BP had better accuracy than NEAT-BP on 37 datasets, and worse on three datasets. It is interesting to also compare ANN to NEAT-BP, although this is not a fair comparison since ANN does not employ BP post-processing while NEAT-BP does. We find that ANN had better accuracy than NEAT-BP on 31 datasets, and worse on nine datasets.

Comparing NEAT-BP to the two baseline classifiers, we find the following: compared to one-nearest-neighbor, NEAT-BP had better accuracy on 14 datasets and worse on 26 datasets; compared to NB, NEAT-BP had better accuracy on 13 datasets and worse on 27 datasets.

As Table 9 indicates, there are five datasets for which NEAT-BP's performance is particularly poor: `annealing`, `chess`, `ecoli`, `nursery`, and `soybean`. Several intersecting factors may explain this performance. For four of those datasets (the exception being `chess`), the number of class labels is large, ranging from five classes for `nursery` to six classes for `annealing` to eight for `ecoli` to a very large 19 classes for `soybean`. Those five datasets also stand out for their large number of attributes (particularly categorical attributes): 38 attributes (including 29 categorical) for `annealing`, 36 attributes (all categorical) for `chess`, seven attributes for `ecoli`, eight attributes (all categorical) for `nursery`, and 35 attributes (all categorical) for `soybean`. Two of those five datasets stand out for their large number of instances: 3,192 instances for `chess` and 12,960 instances for `nursery`. The `soybean` dataset stands out for its small number of instances (307 instances) relative to its large number of attributes (35 attributes) and large number of classes (19 classes). For three of those datasets, specifically `annealing`, `chess` and `soybean`, it is noteworthy that the average model size evolved by NEAT is particularly small (less than half the number of connections in the baseline 3L topology in all three cases).

10 Conclusions and future work directions

The results reported in this paper, using 40 UCI benchmark datasets, indicate that ANN-Miner is an effective algorithm for optimizing the structure of a feed-forward neural network

Table 9 Predictive accuracy (%) results for NEAT-BP, along with two ANN-Miner variations (repeated from Table 4) and two baseline classifiers (repeated from Table 7), as well as model size results for NEAT-BP and for ANN-BP (repeated from Table 6)

Dataset	Accuracy					Model size	
	1NN	NB	NEAT-BP	ANN	ANN-BP	NEAT-BP	ANN-BP
annealing	**94.40**	76.93	36.40	60.97	79.14	0.43	1.12
automobile	**73.64**	58.53	49.29	45.07	48.95	0.55	1.17
balance	74.50	92.83	83.45	90.50	**94.50**	0.65	1.15
breast-l	65.99	**71.25**	61.69	54.25	69.89	0.49	1.12
breast-p	67.18	64.58	68.06	65.76	**72.68**	0.28	1.10
breast tissue	**70.09**	67.18	59.37	41.63	56.46	0.85	1.23
breast-w	**95.61**	93.50	87.97	94.38	94.73	0.57	1.15
car	61.81	85.91	77.70	89.47	**93.94**	1.13	1.17
chess	84.53	88.05	56.00	81.82	**88.64**	0.42	1.14
credit-a	81.02	77.10	84.25	83.48	**84.35**	0.42	1.06
credit-g	71.50	**75.20**	67.73	71.90	72.00	0.53	1.13
cylinder	68.75	66.70	63.82	69.45	**69.78**	0.27	1.12
dermatology	94.53	**97.54**	70.25	86.80	93.16	0.31	1.01
ecoli	81.31	**85.47**	63.77	81.25	84.86	0.95	1.22
glass	**68.90**	49.52	45.80	48.87	57.88	0.88	1.23
hay	63.08	73.08	**78.15**	63.02	69.31	0.67	1.14
heart-c	52.52	56.42	48.15	51.44	**60.04**	0.88	1.19
heart-h	46.55	**65.37**	51.53	62.63	62.66	0.99	1.14
hepatitis	83.12	**83.17**	76.85	79.46	80.75	0.75	1.02
horse	79.05	77.96	73.75	77.71	**80.50**	0.52	1.02
ionosphere	87.38	83.04	88.15	89.67	**92.52**	0.72	1.12
iris	**95.33**	**95.33**	86.93	92.52	92.81	0.80	1.12
liver disorders	60.87	55.89	64.38	64.64	**65.26**	0.62	1.18
lymphography	79.10	**82.47**	73.27	74.28	77.81	0.90	1.11
monks	56.73	**62.73**	40.80	41.56	46.75	0.44	1.14
nursery	86.75	90.37	39.21	87.72	**91.76**	0.97	1.21
parkinsons	**93.42**	70.13	80.35	80.55	82.05	0.46	1.13
pima	70.31	75.64	75.65	74.86	**76.04**	0.72	1.16
s-heart	73.33	84.82	79.41	84.45	**85.56**	0.64	1.12
segmentation	**95.60**	79.94	86.25	88.81	92.77	0.85	1.20
soybean	88.62	**88.97**	37.21	54.48	58.79	0.33	1.20
thyroid	96.23	**96.71**	75.03	79.63	90.28	0.90	1.17
transfusion	61.55	70.07	73.04	73.19	**73.23**	0.72	1.20
ttt	67.37	70.63	77.25	50.32	**90.94**	0.61	1.19
vehicle	**69.26**	45.39	55.66	59.69	60.06	0.48	1.13
vertebral-column-2c	80.97	78.39	82.52	80.00	**83.55**	0.80	1.22
vertebral-column-3c	78.71	**83.55**	64.10	69.32	74.84	0.73	1.23
voting	88.73	85.94	91.18	93.24	**94.65**	0.68	0.94

Table 9 continued

Dataset	Accuracy					Model Size	
	1NN	NB	NEAT-BP	ANN	ANN-BP	NEAT-BP	ANN-BP
`wine`	94.94	**97.19**	90.59	94.93	96.04	0.73	1.05
`zoo`	**98.75**	93.75	77.00	89.11	90.54	0.92	1.09
avg rank/ratio	2.81	2.67	4.03	3.58	**1.92**	0.67	1.14

to a specific dataset, producing improved predictive accuracy compared to the standard three-layer topology and compared to a greedy algorithm for neural network structure optimization.

We have investigated several versions of ANN-Miner, and found the best performing version, in terms of predictive accuracy, to be wANN-Miner-ACO$_{\mathbb{R}}$. In this version, each newly created neural network is initialized with the weights of the best-encountered-so-far network, thus accumulating "wisdom" as the algorithm execution progresses. Furthermore, wANN-Miner-ACO$_{\mathbb{R}}$ uses BP to train each created neural network during the algorithm's execution, and then uses ACO$_{\mathbb{R}}$ as a post-processor to optimize the weights of the final topology. In terms of model size, wANN-Miner produced model sizes that were only slightly larger (specifically 17 % larger on average) than the baseline three-layer topology.

The wANN-Miner-ACO$_{\mathbb{R}}$ model was then compared to several state-of-the-art classifiers (SVM with Gaussian kernels and with quadratic kernels, Ripper, and C4.5) and to two widely-used baseline classifiers (Nearest Neighbor, and Naïve Bayes). In this comparison wANN-Miner-ACO$_{\mathbb{R}}$ ranked behind the two SVM variations, in terms of test set predictive accuracy, and ahead of the other four classifiers. Only the Gaussian-kernel SVM was significantly better than wANN-Miner-ACO$_{\mathbb{R}}$; no statistically significant difference was detected between wANN-Miner-ACO$_{\mathbb{R}}$ and any of the other five classifiers. In addition, wANN-Miner-BP was compared to the NEAT neuroevolutionary algorithm and found to have significantly better test set predictive accuracy, when both algorithms were given comparable CPU resources.

In future work, we would like to extend our ACO approach to optimize the structure of adaptive neurofuzzy inference systems (ANFIS) (Jang et al. 1997). Specifically, we would like to optimize the number of fuzzy rules, the number of fuzzy membership functions for each input, and the type of membership functions deployed in the fuzzification layer. This can later be further extended to optimizing the structure of Type-2 Fuzzy Systems (Karnik et al. 1999), for which manual tuning can be more challenging than in conventional fuzzy systems.

There are also several variations to the ANN-Miner algorithm that we would like to explore:

– We would like to apply the greedy network pruning approach of the GHC algorithm (Algorithm 4) as an additional post-processing step in ANN-Miner. This could potentially result in a reduction in network size without affecting accuracy.
– Other variations of BP can be used, in place of BP, inside the ANN-Miner algorithm. For example, resilient propagation (RP) often performs better than BP without consuming more CPU time.
– It is possible to adjust the number of epochs for which BP is allowed to run, inside ANN-Miner, based on the number of connections in the constructed network. The product of the number of epochs by the number of connections can be required to be constant. This means that constructed networks with a larger number of connections would be allowed

a smaller number of epochs, while networks with a smaller number of connections would be allowed a larger number of epochs.

– In the $ACO_{\mathbb{R}}$ algorithm, we would like to explore using the Cauchy probability distribution in place of the Gaussian probability distribution. The Cauchy has a much wider "tail" and has the potential to promote greater search diversity and help avoid local minima traps.

Acknowledgments The partial support of a grant from the Brandon University Research Council (BURC) is gratefully acknowledged. The authors would like to thank the anonymous reviewers and the associate editor for their insightful comments which have substantially improved the paper.

References

Ang, J., Tan, K., & Al-Mamun, A. (2008). Training neural networks for classification using growth probability-based evolution. *Neurocomputing*, *71*(16–18), 3493–3508.

Angeline, P., Saunders, G., & Pollack, J. (1994). An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, *5*(1), 54–65.

Asuncion, A., Newman, D. (2007). University of California Irvine machine learning repository. http://www.ics.uci.edu/~mlearn/MLRepository.html.

Bishop, C. M. (2006). *Pattern recognition and machine learning*. New York, NY: Springer.

Blum, C., & Socha, K. (2005). Training feed-forward neural networks with ant colony optimization: An application to pattern classification. In *Proceedings international conference on hybrid intelligent systems (HIS-2005)* (pp. 233–238). Piscataway, NJ: IEEE Press.

Boryczka, U., & Kozak, J. (2010). Ant colony decision trees: A new method for constructing decision trees based on ant colony optimization. In *Computational collective intelligence: Technologies and applications (ICCCI-2010), lecture notes in computer science* (Vol. 6421, pp. 373–382). Berlin:Springer.

Boryczka, U., & Kozak, J. (2011). An adaptive discretization in the ACDT algorithm for continuous attributes. In *Computational collective intelligence: Technology and applications (ICCCI-2011), lecture notes in computer science* (Vol. 6923, pp. 475–484). Berlin:Springer.

Cai, X., Venayagamoorthy, G., & Wunsch, D. (2010). Evolutionary swarm neural network game engine for Capture Go. *Neural Networks*, *23*(2), 295–305.

Cangelosi, A., Parisi, D., & Nolfi, S. (1994). Cell division and migration in a 'genotype' for neural networks. *Network: Computation in Neural Systems*, *5*, 497–515.

Castillo, P., Merelo, J., Prieto, A., Rivas, V., & Romero, G. (2000). G-Prop: Global optimization of multilayer perceptrons using GAs. *Neurocomputing*, *35*, 149–163.

Chan, K., Dillon, T., Chang, E., & Singh, J. (2013). Prediction of short-term traffic variables using intelligent swarm-based neural networks. *IEEE Transactions on Control Systems Technology*, *21*(1), 263–274.

Chang, C. C., & Lin, C. J. (2011). LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, *2*(3), 1–27.

Clune, J., Beckmann, B., Ofria, C., & Pennock, R. (2009). Evolving coordinated quadruped gaits with the HyperNEAT generative encoding. In *Proceedings IEEE congress on evolutionary computation (CEC-2009)* (pp. 2764–2771). Piscataway, NJ: IEEE Press.

Coshall, J. (2009). Combining volatility and smoothing forecasts of UK demand for international tourism. *Tourism Management*, *30*(4), 495–511.

Cussat-Blanc, S., Harrington, K. & Pollack, J. (2015). Gene regulatory network evolution through augmenting topologies. *IEEE Transactions on Evolutionary Computation*.

Da, Y., & Xiurun, G. (2005). An improved PSO-based ANN with simulated annealing technique. *Neurocomputing*, *63*, 527–533.

Dehuri, S., Roy, R., Cho, S. B., & Ghosh, A. (2012). An improved swarm optimized functional link artificial neural network (ISO-FLANN) for classification. *Journal of Systems and Software*, *85*(6), 1333–1345.

Derrac, J., García, S., Molina, D., & Herrera, F. (2011). A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. *Swarm and Evolutionary Computation*, *1*, 3–18.

Dorigo, M., & Stützle, T. (2004). *Ant colony optimization*. Cambridge, MA: MIT Press.

Dorigo, M., & Stützle, T. (2010). Ant colony optimization: Overview and recent advances. In *Handbook of Metaheuristics* (pp. 227–263). New York, NY: Springer.

Dorigo, M., Maniezzo, V., & Colorni, A. (1996). Ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, *26*(1), 29–41.

Dorigo, M., Di Caro, G., & Gambardella, L. (1999). Ant algorithms for discrete optimization. *Artificial Life*, *5*(2), 137–172.

Dutta, D., Roy, A., & Choudhury, K. (2013). Training artificial neural network using particle swarm optimization algorithm. *International Journal of Advanced Research in Computer Science and Software Engineering*, *3*(3), 430–434.

Fang, J., & Xi, Y. (1997). Neural network design based on evolutionary programming. *Artificial Intelligence in Engineering*, *11*(2), 155–161.

Fernández-Delgado, M., Cernadas, E., Barro, S., & Amorim, D. (2014). Do we need hundreds of classifiers to solve real world classification problems? *Journal of Machine Learning Research*, *15*(1), 3133–3181.

Floreano, D., Dürr, P., & Mattiussi, C. (2008). Neuroevolution: From architectures to learning. *Evolutionary Intelligence*, *1*(1), 47–62.

Fogel, D. (1993). Using evolutionary programming to create neural networks that are capable of playing Tic-Tac-Toe. In *Proceedings IEEE international conference on neural networks (ICNN-1993)* (Vol. 2, pp. 875–880). Piscataway, NJ: IEEE Press.

Galea, M., & Shen, Q. (2006). Simultaneous ant colony optimization algorithms for learning linguistic fuzzy rules. In *Swarm intelligence in data mining, studies in computational intelligence* (Vol. 34, pp. 75–99). Berlin, Heidelberg: Springer.

Garro, B., Sossa, H., & Vazquez, R. (2011). Evolving neural networks: A comparison between differential evolution and particle swarm optimization. In *Advances in swarm intelligence (ICSI-2011), lecture notes in computer science* (Vol. 6728, pp. 447–454). Berlin: Springer.

Goldberg, D., & Richardson, J. (1987). Genetic algorithms with sharing for multimodal function optimization. In *Proceedings international conference on genetic algorithms (ICGA-1987)* (pp. 41–49). Hillsdale, NJ: L. Erlbaum Associates.

Gomez, F., & Miikkulainen, R. (1999). Solving non-Markovian control tasks with neuroevolution. In *Proceedings international joint conference on artificial intelligence (IJCAI-1999)* (Vol. 2, pp. 1356–1361). San Francisco, CA: Morgan Kaufmann.

Gutiérrez, P., Hervás-Martínez, C., & Martínez-Estudillo, F. (2011). Logistic regression by means of evolutionary radial basis function neural networks. *IEEE Transactions on Neural Networks*, *22*(2), 246–263.

Han, J., Kamber, M., & Pei, J. (2011a). *Data mining: Concepts and techniques*. San Francisco, CA: Morgan Kaufmann.

Han, M., Fan, J., & Wang, J. (2011b). A dynamic feedforward neural network based on Gaussian particle swarm optimization and its application for predictive control. *IEEE Transactions on Neural Networks*, *22*(9), 1457–1468.

Haykin, S. (2008). *Neural networks and learning machines*. New York, NY: Prentice Hall.

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

Ilonen, J., Kamarainen, J. K., & Lampinen, J. (2003). Differential evolution training algorithm for feed-forward neural networks. *Neural Processing Letters*, *17*(1), 93–105.

Jang, J. S., Sun, C. T., & Mizutani, E. (1997). *Neuro-fuzzy and soft-computing: A computational approach to learning and machine intelligence*. Upper Saddle River, NJ: Prentice Hall.

Juang, C. F. (2004). A hybrid of genetic algorithm and particle swarm optimization for recurrent network design. *IEEE Transactions on Systems, Man and Cybernetics, Part B: Cybernetics*, *34*(2), 997–1006.

Kang, D., Mathur, R., & Rao, S. (2010). Real-time bias-adjusted O3 and PM2.5 air quality index forecasts and their performance evaluations over the continental United States. *Atmospheric Environment*, *44*, 2203–2212.

Karnik, N., Mendel, J., & Liang, Q. (1999). Type-2 fuzzy logic systems. *IEEE Transactions on Fuzzy Systems*, *7*(6), 643–658.

Kodjabachian, J., & Meyer, J. A. (1998). Evolution and development of modular control architectures for 1D locomotion in six-legged animats. *Connection Science*, *10*, 211–237.

Leung, F., Lam, H., Ling, S., & Tam, P. (2003). Tuning of the structure and parameters of a neural network using an improved genetic algorithm. *IEEE Transactions on Neural Networks*, *14*(1), 79–88.

Liao, T., Socha, K., Montes de Oca, M., Stützle, T., & Dorigo, M. (2014). Ant colony optimization for mixed-variable optimization problems. *IEEE Transactions on Evolutionary Computation*, *18*(4), 503–518.

Lin, C. J., Chen, C. H., & Lin, C. T. (2009). A hybrid of cooperative particle swarm optimization and cultural algorithm for neural fuzzy networks and its prediction applications. *IEEE Transactions on Systems, Man and Cybernetics, Part C: Applications and Reviews*, *39*(1), 55–68.

Liu, Y. P., Wu, M. G., & Qian, J. X. (2006). Evolving neural networks using the hybrid of ant colony optimization and BP algorithms. In *Advances in neural networks (ISNN-2006), lecture notes in computer science* (Vol. 3971, pp. 714–722). Berlin, Heidelberg: Springer.

Lu, W., Fan, H., & Lo, S. (2003). Application of evolutionary neural network method in predicting pollutant levels in downtown area of Hong Kong. *Neurocomputing*, *51*, 387–400.

Martens, D., De Backer, M., Haesen, R., Vanthienen, J., Snoeck, M., & Baesens, B. (2007). Classification with ant colony optimization. *IEEE Transactions on Evolutionary Computation*, *11*(5), 651–665.

Martens, D., Baesens, B., & Fawcett, T. (2011). Editorial survey: Swarm intelligence for data mining. *Machine Learning*, *82*(1), 1–42.

Martínez-Estudillo, A., Hervás-Martínez, C., Martínez-Estudillo, F., & García-Pedrajas, N. (2005). Hybridization of evolutionary algorithms and local search by means of a clustering method. *IEEE Transactions on Systems, Man and Cybernetics, Part B: Cybernetics*, *36*(3), 534–545.

McDonnel, J., & Waagen, D. (1993). Neural network structure design by evolutionary programming. In *Proceedings second annual conference on evolutionary programming* (pp. 79–89). La Jolla, CA: Evolutionary Programming Society.

Nawi, N., Khan, A., & Rehman, M. (2013). A new back-propagation neural network optimized with cuckoo search algorithm. In *Computational science and its applications (ICCSA-2013), lecture notes in computer science* (Vol. 7971, pp. 413–426). Berlin, Heidelberg: Springer.

Okada, H. (2014). Evolving fuzzy neural networks by particle swarm optimization with fuzzy genotype values. *International Journal of Computing and Digital Systems*, *3*(3), 181–187.

Oong, T., & Isa, N. (2011). Adaptive evolutionary artificial neural networks for pattern classification. *IEEE Transactions on Neural Networks*, *22*(11), 1823–1836.

Otero, F., & Freitas, A. (2013). Improving the interpretability of classification rules discovered by an ant colony algorithm. In *Proceedings genetic and evolutionary computation conference (GECCO-2013)* (pp. 73–80). New York, NY: ACM Press.

Otero, F., Freitas, A., & Johnson, C. (2009). Handling continuous attributes in ant colony classification algorithms. *Proceedings IEEE symposium on computational intelligence and data mining (CIDM-2009)* (pp. 225–231). Piscataway, NJ: IEEE Press.

Otero, F., Freitas, A., & Johnson, C. (2012). Inducing decision trees with an ant colony optimization algorithm. *Applied Soft Computing*, *12*(11), 3615–3626.

Otero, F., Freitas, A., & Johnson, C. (2013). A new sequential covering strategy for inducing classification rules with ant colony algorithms. *IEEE Transactions on Evolutionary Computation*, *17*(1), 64–76.

Palmes, P., Hayasaka, T., & Usui, S. (2005). Mutation-based genetic neural network. *IEEE Transactions on Neural Networks*, *16*(3), 587–600.

Parpinelli, R. S., Lopes, H. S., & Freitas, A. (2002). Data mining with an ant colony optimization algorithm. *IEEE Transactions on Evolutionary Computation*, *6*(4), 321–332.

Potter, M., & De Jong, K. (1995). Evolving neural networks with collaborative species. In *Proceedings summer computer simulation conference* (pp. 340–345). Ottawa, Canada: Society for Computer Simulation.

Risi, S. & Togelius, J. (2014). Neuroevolution in games: State of the art and open challenges. Tech. Rep. arXiv:1410.7326, Computing Research Repository (CoRR), http://arxiv.org/pdf/1410.7326.

Salama, K., & Abdelbar, A. (2014). A novel ant colony algorithm for building neural network topologies. In *Swarm intelligence (ANTS-2014), lecture notes in computer science* (Vol. 8667, pp. 1–12). Cham, Switzerland: Springer.

Salama, K., & Freitas, A. (2013). Extending the ABC-Miner Bayesian classification algorithm. In *Nature inspired cooperative strategies for optimization (NICSO-2013), studies in computational intelligence* (Vol. 512, pp. 1–12). Cham, Switzerland: Springer.

Salama, K., & Freitas, A. (2013b). Learning Bayesian network classifiers using ant colony optimization. *Swarm Intelligence*, *7*(2–3), 229–254.

Salama, K., & Freitas, A. (2014a). ABC-Miner+: Constructing Markov blanket classifiers with ant colony algorithms. *Memetic Computing*, *6*(3), 183–206.

Salama, K., & Freitas, A. (2014b). Classification with cluster-based Bayesian multi-nets using ant colony optimization. *Swarm and Evolutionary Computation*, *18*, 54–70.

Salama, K., & Freitas, A. (2015). Ant colony algorithms for constructing Bayesian multi-net classifiers. *Intelligent Data Analysis*, *19*(2), 233–257.

Salama, K., & Otero, F. (2014). Learning multi-tree classification models with ant colony optimization. In *Proceedings international conference on evolutionary computation theory and applications (ECTA-14)* (pp. 38–48). Rome, Italy: Science and Technology Publications.

Salama, K., Abdelbar, A., & Freitas, A. (2011). Multiple pheromone types and other extensions to the ant-miner classification rule discovery algorithm. *Swarm Intelligence*, *5*(3–4), 149–182.

Salama, K., Abdelbar, A., Otero, F., & Freitas, A. (2013). Utilizing multiple pheromones in an ant-based algorithm for continuous-attribute classification rule discovery. *Applied Soft Computing, 13*(1), 667–675.

Salerno, J. (1997). Using the particle swarm optimization technique to train a recurrent neural model. In *Proceedings IEEE international conference on tools with artificial intelligence* (pp. 45–49). Piscataway, NJ: IEEE Press.

Saravanan, N., & Fogel, D. (1995). Evolving neural control systems. *IEEE Expert*, *10*(3), 23–27.

Schliebs, S., & Kasabov, N. (2013). Evolving spiking neural network: A survey. *Evolving Systems*, *4*(2), 87–98.

Settles, M., Rodebaugh, B., & Soule, T. (2003). Comparison of genetic algorithm and particle swarm optimizer when evolving a recurrent neural network. In *Genetic and evolutionary computation (GECCO-2003), lecture notes in computer science* (Vol. 2723, pp. 148–149). Berlin, Heidelberg: Springer.

Socha, K., & Blum, C. (2007). An ant colony optimization algorithm for continuous optimization: Application to feed-forward neural network training. *Neural Computing & Applications*, *16*, 235–247.

Socha, K., & Dorigo, M. (2008). Ant colony optimization for continuous domains. *European Journal of Operational Research*, *185*, 1155–1173.

Sohangir, S., Rahimi, S., & Gupta, B. (2014). Neuroevolutionary feature selection using NEAT. *Journal of Software Engineering and Applications*, *7*, 562–570.

Song, Y., Chen, Z., & Yuan, Z. (2007). New chaotic PSO-based neural network predictive control for nonlinear process. *IEEE Transactions on Neural Networks*, *18*(2), 595–601.

Stanley, K. (2007). Compositional pattern producing networks: A novel abstraction of development. *Genetic Progamming and Evolvable Machines*, *8*(2), 131–162.

Stanley, K. (2015). The neuroevolution of augmenting topologies (NEAT) users page. http://www.cs.ucf.edu/~kstanley/neat.html.

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

Stanley, K., & Miikkulainen, R. (2004). Evolving a roving eye for Go. In *Genetic and evolutionary computation (GECCO-2004), lecture notes in computer science* (Vol. 3103, pp. 1226–1238). Berlin, Heidelberg: Springer.

Stanley, K., Bryant, B., & Miikkulainen, R. (2005). Evolving neural network agents in the NERO video game. In *Proceedings IEEE symposium on computational intelligence and games (CIG-2005)* (pp. 182–189). Piscataway, NJ: IEEE Press.

Stanley, K., Bryant, B., & Miikkulainen, R. (2005b). Real-time neuroevolution in the NERO video game. *IEEE Transactions on Evolutionary Computation*, *9*(6), 653–668.

Stanley, K., D'Ambrosio, D., & Gauci, J. (2009). A hybercube-based encoding for evolving large-scale neural networks. *Artificial Life*, *15*(2), 185–212.

Stützle, T., & Hoos, H. (2000). MAX–MIN ant system. *Future Generation Computer Systems*, *16*, 889–914.

Tan, P. N., Steinbach, M., & Kumar, V. (2005). *Introduction to data mining*. Boston, MA: Addison Wesley.

Valian, E., Mohanna, S., & Tavakoli, S. (2011). Improved cuckoo search algorithm for feedforward neural network training. *International Journal of Artificial Intelligence and Applications*, *2*(3), 36–43.

Valsalam, V. K., & Miikkulainen, R. (2011). Evolving symmetry for modular system design. *IEEE Transactions on Evolutionary Computation*, *15*(3), 368–386.

Valsalam, V. K., Hiller, J., MacCurdy, R., Lipson, H., & Miikkulainen, R. (2012). Constructing controllers for physical multilegged robots using the ENSO neuroevolution approach. *Evolutionary Intelligence*, *5*(1), 45–56.

Werbos, P. J. (1994). *The roots of backpropagation: From ordered derivatives to neural networks and political forecasting*. New York, NY: Wiley-Interscience.

Whiteson, S., Stone, P., Stanley, K., Miikkulainen, R., & Kohl, N. (2005). Automatic feature selection in neuroevolution. In *Proceedings genetic and evolutionary computation conference (GECCO-2005)* (pp. 1225–1232). New York, NY: ACM Press.

Whitley, D., Starkweather, T., & Bogart, C. (1990). Genetic algorithms and neural networks: Optimizing connections and connectivity. *Parallel Computing*, *14*(3), 347–361.

Whitley, D., Dominic, S., Das, R., & Anderson, C. (1993). Genetic reinforcement learning for neurocontrol problems. *Machine Learning*, *13*(2–3), 259–284.

Witten, I. H., Frank, E., & Hall, M. A. (2010). *Data mining: Practical machine learning tools and techniques*. San Francisco, CA: Morgan Kaufmann.

Yang, J. M., & Kao, C. Y. (2001). A robust evolutionary algorithm for training neural networks. *Neural Computing and Applications*, *10*, 214–230.

Yao, X., & Liu, Y. (1997). A new evolutionary system for evolving artificial neural networks. *IEEE Transactions on Neural Networks*, *8*(3), 694–713.

Yeh, C. Y., Jeng, W. R., & Lee, S. J. (2011). Data-based system modeling using a type-2 fuzzy neural network with a hybrid learning algorithm. *IEEE Transactions on Neural Networks*, *22*(12), 2296–2309.

Yeh, W. C. (2013). New parameter-free simplified swarm optimization for artificial neural network training and its application in the prediction of time series. *IEEE Transactions on Neural Networks and Learning Systems*, *24*(4), 661–665.

Yu, J., Xi, L., & Wang, S. (2007). An improved particle swarm optimization for evolving feedforward artificial neural networks. *Neural Processing Letters*, *26*(3), 217–231.

Yu, J., Wang, S., & Xi, L. (2008). Evolving artificial neural networks using an improved PSO and DPSO. *Neurocomputing*, *71*(4), 1054–1060.