$ACO_{\mathbb{R}g}$: A Gradient-Guided ACO Algorithm for Neural Network Learning

Ashraf M. Abdelbar
Dept. of Mathematics & Computer Science
Brandon University
Manitoba, Canada
Email: AbdelbarA@brandonu.ca

Khalid M. Salama
School of Computing
University of Kent
Canterbury, UK
Email: kms39@kent.ac.uk

***Abstract*—The $ACO_{\mathbb{R}}$ algorithm is an Ant Colony Optimization (ACO) algorithm for real-valued optimization, and has been applied to neural network learning. Unlike many algorithms for neural network learning, $ACO_{\mathbb{R}}$ does not use gradient information at all in its operation. Also, unlike many discrete ACO algorithms, $ACO_{\mathbb{R}}$ does not allow for the incorporation of domain-specific heuristics. In this work, we present a gradient-guided variation of $ACO_{\mathbb{R}}$, that we call $ACO_{\mathbb{R}g}$, that incorporates gradient information while retaining the core aspects of the $ACO_{\mathbb{R}}$ algorithm. Experimental results using 10-fold cross-validation with 20 UCI datasets indicate that our variation produces lower test set error than standard $ACO_{\mathbb{R}}$, after a markedly smaller number of training generations.**

I. Introduction

Feedforward neural networks are one of the most widely-used and generally-effective techniques for classification [1]. The most commonly-used techniques for neural network learning are either direct implementations of stochastic gradient descent, such as the well-known Back-Propagation (BP) algorithm [2], or are methods (such as resilient propagation [3], the conjugate gradient method [4], and others [5]) that use gradient information in a more advanced way. Non-gradient population-based methods, such as Genetic Algorithms (GA) and Particle Swarm Optimization (PSO), have also been applied to neural network training [6], [7].

Gradient-based methods have the advantage that partial derivatives can provide very direct and immediate information on the direction in which to move the network weights in order to reduce the error. But, they also have the disadvantage that they cannot directly see beyond the nearest local optimum. On the other hand, non-gradient methods have the disadvantage that the only problem-dependent information they have access to is the error function. Therefore, they will generally take longer and will sometimes not be as effective as gradient-based methods.

$ACO_{\mathbb{R}}$ [8], [9] is a fairly-recent Ant Colony Optimization (ACO) algorithm for real-variable optimization, and has been applied to training fixed-topology feedforward neural networks [10]. Like GA and PSO approaches, $ACO_{\mathbb{R}}$ does not use gradient information at all, and the only problem-dependent information that it has access to is the fitness function (error function). We propose a variation of $ACO_{\mathbb{R}}$, that we call $ACO_{\mathbb{R}g}$, for training fixed-topology neural networks that allows for the use of gradient information to an extent that is controlled by an external parameter $0 \leq \varrho \leq 1$. Setting ϱ to 1 reduces $ACO_{\mathbb{R}g}$ to $ACO_{\mathbb{R}}$, while setting it to 0 makes maximal use of gradient information. Based on initial experimentation, we have determined $\varrho = 0.05$ to be a good generic setting, and this is the setting that we use in the present work.

Using stratified 10-fold cross-validation, we compare our proposed $ACO_{\mathbb{R}g}$ to $ACO_{\mathbb{R}}$, in the training of 3-layer Multi-Layer Perceptron (MLP) neural networks on 20 UCI datasets [11]. We find that $ACO_{\mathbb{R}g}$ produces lower test set MSE (mean-squared error) than $ACO_{\mathbb{R}}$ in a clear majority of the datasets, and consistently (on 20 out of 20 datasets) converges after a smaller number of generations—on average, reducing $ACO_{\mathbb{R}}$'s number of generations by nearly a factor of 4.

This rest of the paper is structured as follows. Section II provides a brief introduction to ACO and to feedforward neural networks. Section III reviews the $ACO_{\mathbb{R}}$ algorithm, while Section IV presents our proposed $ACO_{\mathbb{R}g}$. Section V presents our experimental methodology, followed by computational results in Section VI. Finally, Section VII concludes with some final remarks and suggestions for further research.

II. Background

A. Ant Colony Optimization

Swarm intelligence studies natural systems composed of many individuals interacting locally with each other and with their environment, using forms of decentralized control and self-organization to achieve their goals. Introduced by Dorigo et al., Ant Colony Optimization (ACO) [12] is a well-studied and widely-applied type of swarm-based systems for problem solving, which mimics the real ants foraging behaviour in finding the shortest path between the food source and the nest. The search process is guided in an ACO algorithm through the accumulating pheromone information associated with the different parts of the problem's search space, which is updated by the ants according to the quality of the constructed solution during the course algorithm's execution.

In essence, the search space of a given problem is first transformed into a graph of solution components, whereby a combination of these components would present a valid

978-1-4799-7560-0/15 $31.00 © 2015 IEEE
DOI 10.1109/SSCI.2015.162

IEEE computer society

candidate solution to that problem. A population of ants then navigates this graph, selecting decision components to add to their path chosen throughout the graph in an attempt to build a candidate solution. In parallel to real life ants, the ants are trying to find the shortest path, where the "shortness" of their path corresponds to the "quality" of the constructed solution in the context of the problem at hand. The probability for an ant to select its next component is based on two properties: the heuristic advantage associated with it, and the amount of pheromone present.

As the ants traverse the graph, they deposit pheromone on the nodes that they have chosen in proportion to the quality of the solution constructed by the ant. The more a decision component is chosen by ants in the colony (implying that it contributes well to the quality of solutions produced), the more pheromone is deposited on it and the higher the probability that it will be picked by ants in future iterations. The iterative process of building candidate solutions, evaluating their quality and updating pheromone values allows an ACO algorithm to converge to near-optimal solutions.

Classically applied to combinatorial optimization problems, ACO has also been successful in tackling classification problems. A number of ACO-based algorithms have been introduced in the literature with different classification learning approaches. Ant-Miner [13] is the first ant-based classification algorithm, which discovers a list of classification rules. The algorithm has been followed by several extensions in [13]–[17]. Otero and Salama [18], [19] have proposed two different ACO-based algorithms for inducing decision trees for classification. ACO was also employed in [20]–[22] to learn various types of Bayesian network classifiers. ANN-Miner [23], [24] was recently introduced as an ACO-based algorithm for learning neural network topologies.

B. 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 combinatorial 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*, most commonly chosen to be the sigmoid function:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

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 mean squared error (MSE):

$$E = \frac{1}{m|P|} \sum_{p \in P} E_p \tag{4}$$

and

$$E_p = \frac{1}{2} \sum_{i=1}^{m} (y_i - y_i')^2 \tag{5}$$

where P is the set of training patterns and m is the number of classes. Note that the MSE is the total sum of squared error divided by the number of patterns and the number of classes. Thus, it will always be the case that $0 \leq MSE \leq 0.5$, with a value of 0 representing the lowest error and of 0.5 representing the highest error.

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

$$y_k = \begin{cases} 1 & \text{if } k = \hat{c} \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

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 4 neurons in the input layer, 5 neurons in the hidden layer, and 3 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.

In gradient descent methods, the partial derivative $\frac{\partial E}{\partial w_i}$ of the error E with respect to each element i of the weight vector is computed, and the weights are updated according to:

$$w_{aj}^{(new)} = w_{aj}^{(old)} + \alpha \left(-\frac{\partial E}{\partial w_i} \right) \tag{7}$$

where α is a learning rate parameter.

III. Review of the ACO$_\mathbb{R}$ Algorithm

Suppose the ACO$_\mathbb{R}$ algorithm is to be applied to an optimization problem over n real-valued variables $V_1, V_2, \ldots, V_n$. The central data structure, analogous to pheromone information in natural ants, that is maintained by ACO$_\mathbb{R}$ is an archive A of R previously-generated candidate solutions. Each element s_a in the archive, for $a = 1, 2, \ldots, R$, is an n-dimensional real-valued vector, $s_a = (s_{a,1}, s_{a,2}, \ldots, s_{a,n})$. For example, $s_{a,j}$ refers to the value of the j-th variable in the a-th solution in the archive. The archive is sorted by solution quality, so that $Q(s_1) \geq Q(s_2) \geq \ldots \geq Q(s_R)$. Each solution s_a in the archive has an associated weight ω_a that is related to $Q(s_a)$, so that $\omega_1 \geq \omega_2 \geq \ldots \geq \omega_R$.

Algorithm 1 represents the pseudo-code of ACO$_\mathbb{R}$. As shown, ACO$_\mathbb{R}$ consists of repeated iterations until some termination criteria is reached (e.g. solution cost falls below some desired threshold, some maximum number of iterations is reached, etc.). In each iteration, there are two phases: solution construction, and pheromone update. In the solution construction phase, each ant probabilistically constructs a solution based on the solution archive A (representing pheromone information). The solution archive A is initialized with R randomly-generated solutions, where the size R is an external parameter of the ACO$_\mathbb{R}$ algorithm. Then, in the pheromone update phase, the m constructed solutions (where m is the number of ants) are added to A, resulting in the size of A temporarily being $R + m$. The archive A is then sorted by solution quality, and the m worst solutions are discarded, so that the size of A returns to being R.

The heart of the algorithm is the solution construction phase. In this phase, each ant i generates a candidate solution s_i, where s_i is an n-dimensional vector, and $s_{i,j}$ represents an assignment to the j-th variable V_j. In constructing its solution s_i, ant i is influenced by one of the R solutions in the archive A. The ant first probabilistically selects one of the R solutions in the archive according to:

$$\Pr(\text{select } s_a) = \frac{\omega_a}{\sum_{r=1}^{R} \omega_r} \tag{8}$$

Thus, the probability of selecting the a-th solution is proportional to its weight ω_a. Recall that the archive A is sorted by quality, so that solution s_a has rank a. The weights ω_a that are used in Equation (8) are constructed in each iteration as:

$$\omega_a = g(a; 1, qR) \tag{9}$$

where g is the Gaussian function:

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

Thus, Equation (9) assigns the weight ω_a to be the value of the Gaussian function with argument a, mean 1.0, and standard deviation (qR). The value of q is an external parameter of the algorithm, where smaller values of q cause the better ranked solutions to have higher weights ω (and thus makes the algorithm more exploitative), while larger values of q result in a more uniform distribution.

Let s_a be the solution of A that is selected by ant i according to Equation (8) in a given iteration. Ant i then generates each solution element $s_{i,a}$ by sampling the Gaussian probability density function (PDF):

$$s_{i,j} \sim N(s_{a,j}, \sigma_{a,j}) \tag{11}$$

where $N(\mu, \sigma)$ represents the Gaussian PDF with mean μ and standard deviation σ.

In Eq. (11), $s_{a,j}$ represents the value that the solution s_a assigns to variable V_j, and the standard deviation $\sigma_{a,j}$ is computed according to:

$$\sigma_{a,j} = \xi \sum_{r=1}^{R} \frac{|s_{a,j} - s_{r,j}|}{R - 1} \tag{12}$$

where ξ is an external parameter of the algorithm. The effect of Equation (12) is that the average distance from s_a to other solutions in the archive, for the j-th dimension, is computed, and is then multiplied by ξ. The parameter ξ plays a role in ACO$_\mathbb{R}$ similar to that of evaporation rate in other ACO algorithms. The higher the value of ξ, the less the extent to which the search is biased towards the area of the search space around the solutions stored in the archive, and the slower the algorithm will converge. Once each ant constructs its solution, the archive A is updated as described above. The process repeats until the desired termination criteria are met.

In all, the algorithm has four external parameters m, R, q, and ξ, in addition to any parameters related to the termination criteria. The parameter m determines the number of ants; the parameter R determines the number of solutions stored in the archive A; the parameter q controls the extent to which the top solutions in the archive will dominate solution construction (Eq. 9); and the parameter ξ influences the degree of diversity in solution construction (Eq. 12).

When ACO$_\mathbb{R}$ is applied to neural network learning, the n real-valued variables $V_1, \ldots, V_n$ that are optimized by the algorithm correspond to the neural network weight vector w, with $n = |w|$. Each archived solution s_i is a representation of a network weight vector w_i, and each entry $s_{i,j}$ of an archived solution s_i corresponds to the j-th element of the weight vector w_i, denoted $w_{i,j}$. In the remainder of this paper, we will use the notation s_i and w_i interchageably, when referring to the application of ACO$_\mathbb{R}$ to neural network learning.

To evaluate the quality $Q(s_i)$ of the i-th archived solution, the solution s_i is interpreted as a weight vector w_i. A neural network is then created using the weight vector w_i. The training set is presented to the network, and the training set MSE is obtained according to Eqs. (4-5). The quality $Q(s_i)$ is then set as:

$$Q(s_i) = 2 \cdot (1 - MSE) \tag{13}$$

so that $0 \leq Q(s_i) \leq 1$, with 1 representing the best quality and 0 representing the worst quality.

Algorithm 1 Pseudo-code of the $\text{ACO}_{\mathbb{R}}$ Algorithm

Begin
  $t = 1$;
  $s_{best} =$ **null**;
  $Q_{bset} = -\infty$;
  $RandomizeArchive(A, R)$;
  for $i = 1 \ \rightarrow R$ **do**
    $Q_i = EvaluateQuality(s_i)$;
  end for
  $RankArchiveByQuality(A)$;
  repeat
    for $i = 1 \ \rightarrow m$ **do**
      $s_a = SelectFromArchive(A, q)$;
      for $j = 1 \ \rightarrow n$ **do**
        $\sigma_{a,j} = Calculate(\xi, A, j)$;
        $s_{i,j} = N(s_{a,j}, \sigma_{aj})$; // N denotes Gaussian PDF
      end for
      $Q_i = EvaluateQuality(s_i)$;
      if $Q_i > Q_{best}$ **then**
        $s_{best} \leftarrow s_i$;
      end if
      $AddToArchive(A, s_i)$;
    end for
    $RankArchiveByQuality(A)$;
    $RemoveWorstSolutions(A, m)$;
    $t = t + 1$;
  until $t = t_{max}$ **or** $termination\_condition()$;
  return s_{best};
End

IV. Our Proposed $\text{ACO}_{\mathbb{R}g}$ Algorithm

Most ACO algorithms make use of problem-dependent heuristic information. Readers who are familiar with discrete ACO algorithms will recognize the familiar ACO probabilistic action equation:

$$\Pr(d_i) = \frac{\tau_i \cdot \eta_i}{\sum_k \tau_k \cdot \eta_k} \tag{14}$$

where τ represents pheromone information and η represents heuristic information. Perhaps the most useful heuristic information available in the case of neural network learning is gradient information. In the $\text{ACO}_{\mathbb{R}}$ algorithm, the solution archive A plays the role of pheromone information, but there is no mechanism for incorporating such information in a general way within the framework of $\text{ACO}_{\mathbb{R}}$. Perhaps, future researchers will investigate ways to incorporate heuristic information, where it is available, within $\text{ACO}_{\mathbb{R}}$ in a general way. In the present work, we propose a mechanism for including heuristic information (gradient information) in a way that is specific to the problem of training neural networks.

Our $\text{ACO}_{\mathbb{R}g}$ algorithm modifies $\text{ACO}_{\mathbb{R}}$ in several ways. **First**, in the solution quality evaluation step of $\text{ACO}_{\mathbb{R}}$, as described above, the training set is applied to the network corresponding to the solution under evaluation, pattern by pattern. For each pattern, the input is propagated forward to obtain the output of the network so that the pattern error (Eq. 5) can be determined. In $\text{ACO}_{\mathbb{R}g}$, we add a second substep in which the pattern error is propagated backwards through the network to obtain the gradient vector $\mathcal{G}^{(p)}$ where $|\mathcal{G}| = |\mathcal{G}^{(p)}| = |w|$, and

$$\mathcal{G}_i^{(p)} \equiv -\frac{\partial E_p}{\partial w_i} \tag{15}$$

The overall gradient vector is then obtained as:

$$\mathcal{G}_i = \sum_{p \in P} \mathcal{G}_i^{(p)} \tag{16}$$

(Note that we are being a little "loose" with our nomenclature in referring to $\mathcal{G}$ as the gradient vector. Technically, of course, $\mathcal{G}$ is the negative-of-the-gradient-vector, but for ease of expression, we will refer to it as the gradient vector throughout the remainder of this paper.) Thus, a side-effect of evaluating $Q(s_i)$ is that a gradient vector $\mathcal{G}_i$ is also computed, and is stored in the archive as auxiliary information attached to s_i.

Second, in the solution construction phase, before generating the j-th element of the i-th constructed solution, $s_{i,j}$, we probabilistically decide whether or not to apply gradient guidance. With probability $(1 - \varrho)$, gradient guidance is applied; with the opposite probability ϱ, gradient guidance is not applied. Note that this decision is made independently for each

Algorithm 2 Pseudo-code of the $\text{ACO}_{\mathbb{R}g}$ Algorithm

Begin
  $t = 1$;
  $s_{best} =$ **null**;
  $Q_{bset} = -\infty$;
  $RandomizeArchive(A, R)$;
  for $i = 1 \ \rightarrow R$ **do**
    $(Q_i, \mathcal{G}_i) = EvaluateQualityAndComputeGradient(s_i)$;
  end for
  $RankArchiveByQuality(A)$;
  repeat
    for $i = 1 \ \rightarrow m$ **do**
      $s_a = SelectFromArchive(A, q)$;
      for $j = 1 \ \rightarrow n$ **do**
        if $Random(0, 1) \leq \varrho$ **then**
          $\sigma_{a,j} = Calculate(\xi, A, j)$;
          $s_{i,j} = N(s_{a,j}, \sigma_{a,j})$;
        else
          $S_{a,j} = ComputeInfluenceSet(A, \mathcal{G}_a, j)$;
          $\sigma_{a,j} = Calculate(\xi, S_{a,j}, j)$;
          if $\sigma_{a,j}/s_{a,j} < 0.05$ **then**
            $\sigma_{a,j} = 0.05 * s_{a,j}$;
          end if
          $s_{i,j} = s_{a,j} + \phi(\mathcal{G}_{t,j}) * |N(s_{a,j}, \sigma_{a,j}) - s_{a,j}|$
        end if
      end for
      $(Q_i, \mathcal{G}_i) = EvaluateQualityAndComputeGradient(s_i)$;
      if $Q_i > Q_{best}$ **then**
        $s_{best} \leftarrow s_i$;
      end if
      $AddToArchive(A, s_i)$;
    end for
    $RankArchiveByQuality(A)$;
    $RemoveWorstSolutions(A, m)$;
    $t = t + 1$;
  until $t = t_{max}$ **or** $termination\_condition()$
  return s_{best};
End

dimension j of each constructed solution. Thus, for a given constructed solution, it is possible that some of its dimensions will have been constructed with gradient guidance and some without. This can be expected to increase the diversity of the solution construction process. In the present work, we fix ϱ at 0.05.

Third, in the solution construction phase, if it is decided to apply gradient guidance for a particular element $s_{i,j}$, then Eq. (12) is modified as follows. Eq. (12) computes $\sigma_{a,j}$, which is the standard deviation to be used with the j-th element of the weight vector w_a in the Gaussian PDF of Eq. (11). The standard deviation $\sigma_{a,j}$ is obtained by first obtaining the average distance, in the j-th dimension, between w_a and all other archived solutions, and then multiplying this average by the external parameter ξ. In the case of $\text{ACO}_{\mathbb{R}g}$, we would like to make use of the value $\mathcal{G}_{a,j}$. If the value of $\mathcal{G}_{a,j}$ is positive, then this indicates that the signal contained within the gradient information is that an increase in the value of the j-th element of weight vector w_i will result in a decrease in the error E. If $\mathcal{G}_{a,j}$ is negative, then the gradient is signalling that $w_{i,j}$ should be decreased in order to decrease the error E.

We identify the set $S_{a,j}$ of archived solutions s_r for which $(s_{r,j} - s_{a,j})$ has the same **sign** as $\mathcal{G}_{a,j}$:

$$S_{a,j} = \{s_r | \phi(s_{r,j} - s_{a,j}) = \phi(\mathcal{G}_{a,j})\} \tag{17}$$

where ϕ is the sign function,

$$\phi(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases} \tag{18}$$

For example, if $\mathcal{G}_{a,j}$ is positive, then $S_{a,j}$ will include those archived solutions s_r for which $s_{r,j} > s_{a,j}$; if $\mathcal{G}_{a,j}$ is negative,

then $S_{a,j}$ will include those solutions s_r for which $s_{r,j} < s_{a,j}$. Eq. (12) is then replaced with

$$\sigma_{a,j} = \xi \sum_{s_r \in S_{a,j}} \frac{|s_{a,j} - s_{r,j}|}{|S_{a,j}|} \tag{19}$$

This modification will ensure that, for the j-th dimension, solution construction will be guided only by those elements of the archive that would move $s_{a,j}$ in the direction indicated by gradient information.

Fourth, if gradient guidance is employed, then Eq. (11) is replaced by:

$$t \sim N(s_{a,j}, \sigma_{a,j}) \tag{20}$$

and

$$s_{i,j} = s_{a,j} + \phi(\mathcal{G}_{a,j}) \cdot |t - s_{a,j}| \tag{21}$$

In other words, if $\mathcal{G}_{a,j}$ is positive, then $s_{i,j}$ is set to $s_{a,j}$ plus the non-negative value $|t - s_{a,j}|$, while if $\mathcal{G}_{a,j}$ is negative, then $s_{i,j}$ is set to $s_{a,j}$ minus the non-negative value $|t - s_{a,j}|$. Thus, this ensures that, in all cases,

$$\phi(s_{i,j} - s_{a,j}) = \phi(\mathcal{G}_{a,j}) \tag{22}$$

Fifth, it will rarely but occasionally happen that the application of Eq. (17) will result in a set $S_{a,j}$ consisting of a single vector (or a small number of similar vectors). In such a case, $\sigma_{a,j}$, computed according to Eq. (19) will be zero (or very small), resulting in no diversity (or very little diversity) in solution construction. To avoid this, after computing $\sigma_{a,j}$ according to Eq. (19), we check to make sure that

$$\frac{\sigma_{a,j}}{s_{a,j}} \geq 0.05 \tag{23}$$

Otherwise, we set

$$\sigma_{a,j} = 0.05 \cdot s_{a,j} \tag{24}$$

Algorithm 2 summarizes the overall behaviour of the gradient-guided ACO Algorithm $\text{ACO}_{\mathbb{R}g}$.

V. Experimental Methodology

TABLE I
Parameter settings for the $\text{ACO}_{\mathbb{R}}$ algorithm.

Parameter	Description	Values
m	# of ants	5
R	archive size	90
q	controls exploration/exploitation	0.05
ξ	controls speed of convergence	0.68

We evaluate the performance of $\text{ACO}_{\mathbb{R}g}$ relative to $\text{ACO}_{\mathbb{R}}$. In [9], Liao et al. used the Iterated F-Race [25] automated parameter optimization procedure to optimize the parameters of $\text{ACO}_{\mathbb{R}}$. In our experimental results, we use the optimized $\text{ACO}_{\mathbb{R}}$ parameters reported by Liao et al. [9], rounded to two decimal places; these values are shown in Table I. For $\text{ACO}_{\mathbb{R}g}$, we used the same values for the parameters that $\text{ACO}_{\mathbb{R}g}$ has in common with $\text{ACO}_{\mathbb{R}}$; for the ϱ parameter, that is unique to $\text{ACO}_{\mathbb{R}g}$, we used a value of $\varrho = 0.05$, based on some initial *ad hoc* experimentation along with the intuition that we would like to inject a 5% element of noise.

We set the termination criteria for both $\text{ACO}_{\mathbb{R}}$ and $\text{ACO}_{\mathbb{R}g}$ to be the occurrence of one of the following two conditions:

1) the number of generations exceeds 5000,
2) 100 generations elapse during which there is no improvement in the quality of the best solution in the archive R.

We used 20 datasets, obtained from the publically-available University of California Irvine (UCI) dataset repository [11]. Table II reports the main characteristics of these datasets.

Each of the two algorithms ($\text{ACO}_{\mathbb{R}}$ and $\text{ACO}_{\mathbb{R}g}$) was applied to training a feedforward Multi-Layer Perceptron (MLP) neural network. The number of network inputs and network outputs are of course determined by characteristics of the dataset. Continuous attributes are scaled to the range [0,1], and any missing values are set to the average value for that attribute. Each categorial attribute, with k category labels, is mapped to k network inputs, where one of the inputs has a value of 1, and each of the other $(k-1)$ inputs has a value of 0. Any missing values for a categorial attribute are set to the mode for that attribute. If the dataset has m possible classes, then the network will have m outputs, whose target values are set according to Eq. (6). The last three columns of Table II show the topology of the neural network for each dataset. As described above, the number of network output neurons is the same as the number classes. The number of network inputs is equal to the number of continuous attributes plus the sum of the number of category labels of the categorical attributes. In the present work, we set the number of hidden neurons to be equal to the number of input neurons plus the number of output neurons.

The experiments were carried out using the *stratified* ten-fold cross validation procedure. This means that a dataset is divided into ten mutually exclusive partitions (folds), with approximately the same number of instances and roughly the same class distribution in each fold. Then, each classification algorithm is run ten times, where each time a different fold is used as the test set and the other nine folds are used as the training set. Performance on each of the test set folds is recorded, and the average test set performance, aggregated over all 10 folds, is reported as representative of the performance of each algorithm.

VI. Experimental Results

Table III reports the results for the $\text{ACO}_{\mathbb{R}}$ and $\text{ACO}_{\mathbb{R}g}$ algorithms on the datasets shown in Table II. For each dataset, the table reports the average test set MSE for each of the two algorithms, with the best test MSE for each dataset indicated in boldface, along with the number of training generations, again with the smaller number of generations shown in boldface. The penultimate row in both tables indicates the number of datasets for which each algorithm had the best performance. The last row indicates the average rank of each algorithm for test set MSE along with the average number of training generations. The average rank for a given algorithm g for test set MSE is obtained by first computing the rank of g on each dataset

TABLE II
CHARACTERISTICS OF THE DATASETS AND THE NEURAL NETWORK TOPOLOGIES.

Dataset	Instances	Classes	Features			NN Topology		
			Total	Numeric	Categorical	Input	Hidden	Output
balance	625	3	4	0	4	20	23	3
breast-p	198	2	32	32	0	32	34	2
breast-tissue	106	6	9	9	0	9	15	6
breast-w	569	2	30	30	0	30	32	2
cylinder	540	2	35	19	16	79	81	2
ecoli	336	8	7	7	0	7	15	8
hay	132	3	4	0	4	15	18	3
heart-c	303	5	13	7	6	23	28	5
heart-h	293	5	13	7	6	23	28	5
ionosphere	351	2	34	34	0	34	36	2
iris	150	3	4	4	0	4	7	3
liver-disorders	345	2	6	6	0	6	8	2
parkinsons	195	2	22	22	0	22	24	2
pima	768	2	8	8	0	8	10	2
pop	90	3	8	0	8	24	27	3
segmentation	2,273	7	19	19	0	19	26	7
transfusion	722	2	4	4	0	4	6	2
ttt	958	2	9	0	9	27	29	2
wine	178	3	13	13	0	13	16	3
zoo	101	7	16	0	16	36	43	7

TABLE III
EXPERIMENTAL RESULTS.

Dataset	Test Set MSE		Training Generations	
	$ACO_{\mathbb{R}}$	$ACO_{\mathbb{R}_g}$	$ACO_{\mathbb{R}}$	$ACO_{\mathbb{R}_g}$
balance	0.0190	**0.0141**	3,809	**726**
breast-p	0.0683	**0.0653**	3,627	**1,623**
breast-tissue	0.0335	**0.0307**	3,654	**1,447**
breast-w	0.0263	**0.0244**	3,264	**625**
cylinder	0.0757	**0.0713**	4,222	**1,696**
ecoli	**0.0181**	0.0300	3,550	**1,016**
hay	0.0311	**0.0238**	3,447	**648**
heart-c	**0.0337**	0.0448	4,266	**1,604**
heart-h	**0.0329**	0.0407	4,183	**1,469**
ionosphere	**0.0310**	0.0352	3,923	**881**
iris	0.0244	**0.0118**	3,713	**1,013**
liver-disorders	0.0811	**0.0730**	3,467	**883**
parkinsons	**0.0445**	0.0541	3,630	**1,465**
pima	0.0598	**0.0546**	2,534	**777**
pop	**0.0564**	0.0750	4,368	**1,080**
segmentation	0.0204	**0.0112**	4,417	**1,013**
transfusion	0.0628	**0.0612**	2,989	**1,082**
ttt	0.0333	**0.0258**	3,994	**934**
wine	0.0165	**0.0098**	3,621	**712**
zoo	0.0026	**0.0021**	4,741	**233**
#wins	6	**14**	0	**20**
avg. rank/avg. gens.	1.70	**1.30**	3,771.0	**1,046.4**

individually, and then averaging the individual ranks across all datasets to obtain the overall average rank for algorithm g. Note that the lower the value of the rank, the better the algorithm.

The results indicate that $\text{ACO}_{\mathbb{R}_g}$ had better test set performance on 14 out of 20 datasets, a clear majority, with an average rank of 1.3. The results also indicate that $\text{ACO}_{\mathbb{R}_g}$ had a smaller number of generations on every single dataset, with the average number of generations for $\text{ACO}_{\mathbb{R}_g}$ being 27.7% of the number of generations for $\text{ACO}_{\mathbb{R}}$.

VII. Concluding Remarks

In this proof of concept paper, we have shown that gradient information can be effectively used within the $\text{ACO}_{\mathbb{R}}$ algorithm, when applied to neural network training. Most discrete ACO algorithms include heuristic information in their solution construction procedure, and it makes sense that gradient information should play a similar role in neural network learning.

Using 20 publicly-available UCI datasets, we found that the use of gradient information improves test set performance in 14 out of 20 datasets, and also substantially and consistently reduces the number of training generations in all 20 datasets.

In future work, we would like to consider a hybridization in which the algorithm has two concurrent threads. One thread runs the $\text{ACO}_{\mathbb{R}_g}$ algorithm, as described in this paper. The other thread repeatedly takes the top weight-vector in the archive (or perhaps a random vector from the archive) and uses it to launch a gradient-based algorithm, such as the Scaled Conjugate Gradient algorithm [4] or the Levenberg-Marquardt algorithm [5], and then return the resultant "improved" weight vector to compete for a place in the archive based on its quality. Such a hybrid approach would benefit from the advantages of both gradient and population-based non-gradient methods.

Acknowledgment

Partial support of a grant from the Brandon University Research Council is gratefully acknowledged.

References

[1] S. Haykin, *Neural Networks and Learning Machines*. New York, NY, USA: Prentice Hall, 2008.

[2] P. J. Werbos, *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*. New York, NY: Wiley-Interscience, 1994.

[3] M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning: The RPROP algorithm," *IEEE International Conference on Neural Networks*, pp. 586–591, 1993.

[4] M. F. Møller, "A scaled conjugate gradient algorithm for fast supervised learning," *Neural Networks*, vol. 6, no. 4, pp. 525–533, 1993.

[5] D. W. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters," *Journal of the Society for Industrial & Applied Mathematics*, vol. 11, no. 2, pp. 431–441, 1963.

[6] N. Saravanan and D. Fogel, "Evolving neural control systems," *IEEE Expert*, vol. 10, no. 3, pp. 23–27, 1995.

[7] X. Cai, G. Venayagamoorthy, and D. Wunsch, "Evolutionary swarm neural network game engine for capture Go," *Neural Networks*, vol. 23, pp. 295–305, 2010.

[8] K. Socha and M. Dorigo, "Ant colony optimization for continuous domains," *European Journal of Operational Research*, vol. 185, pp. 1155–1173, 2008.

[9] T. Liao, K. Socha, M. Montes de Oca, T. Stützle, and M. Dorigo, "Ant colony optimization for mixed-variable optimization problems," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 4, pp. 503–518, 2014.

[10] K. Socha and C. Blum, "An ant colony optimization algorithm for continuous optimization: Application to feed-forward neural network training," *Neural Computing & Applications*, vol. 16, pp. 235–247, 2007.

[11] A. Asuncion and D. Newman, "University of California Irvine Machine Learning Repository," 2007. [Online]. Available: http://www.ics.uci.edu/ mlearn/MLRepository.html

[12] M. Dorigo and T. Stützle, *Ant Colony Optimization*. Cambridge, MA, USA: MIT Press, 2004.

[13] R. S. Parpinelli, H. S. Lopes, and A. A. Freitas, "Data mining with an ant colony optimization algorithm," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 4, pp. 321–332, 2002.

[14] D. Martens, M. De Backer, R. Haesen, J. Vanthienen, M. Snoeck, and B. Baesens, "Classification with ant colony optimization," *IEEE Transactions on Evolutionary Computation*, vol. 11, no. 5, pp. 651–665, 2007.

[15] K. Salama, A. Abdelbar, and A. Freitas, "Multiple pheromone types and other extensions to the Ant-Miner classification rule discovery algorithm," *Swarm Intelligence*, vol. 5, no. 3-4, pp. 149–182, 2011.

[16] F. Otero, A. Freitas, and C. Johnson, "Handling continuous attributes in ant colony classification algorithms," in *Proceedings IEEE Symposium on Computational Intelligence and Data Mining (CIDM-2009)*, 2009, pp. 225–231.

[17] ——, "A new sequential covering strategy for inducing classification rules with ant colony algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 17, no. 1, pp. 64–76, 2013.

[18] ——, "Inducing decision trees with an ant colony optimization algorithm," *Applied Soft Computing*, vol. 12, no. 11, pp. 3615–3626, 2012.

[19] K. Salama and F. Otero, "Learning multi-tree classification models with ant colony optimization," in *Proceedings International Conference on Evolutionary Computation Theory and Applications (ECTA-2014)*, 2014.

[20] K. Salama and A. Freitas, "Learning Bayesian network classifiers using ant colony optimization," *Swarm Intelligence*, vol. 7, no. 2-3, pp. 229–254, 2013.

[21] ——, "Ant colony algorithms for constructing Bayesian multi-net classifiers," *Intelligent Data Analysis*, vol. 19, no. 2, pp. 233–257, 2015.

[22] ——, "ABC-Miner+: Constructing Markov blanket classifiers with ant colony algorithms," *Memetic Computing*, vol. 6, no. 3, pp. 183–206, 2014.

[23] K. Salama and A. Abdelbar, "A novel ant colony algorithm for building neural network topologies," in *Proceedings ANTS-2014, Lecture Notes in Computer Science Vol. 8667*. Springer, 2014, pp. 1–12.

[24] ——, "Learning neural network structures with ant colony algorithms," *Swarm Intelligence*, 2015, to appear.

[25] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle, "F-Race and iterated F-Race: An overview," in *Experimental Methods for the Analysis of Optimization Algorithms*, T. Bartz-Beielstein, M. Chiarandini, L. Paquete, and M. Preuss, Eds. Berlin: Springer, 2010, pp. 311–336.