TRAINING NEURAL NETWORKS WITH ANT COLONY OPTIMIZATION

A Project

Presented to the faculty of the Department of Computer Science

California State University, Sacramento

Submitted in partial satisfaction of
the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

by

Arun Pandian

SPRING
2013

TRAINING NEURAL NETWORKS WITH ANT COLONY OPTIMIZATION

A Project

by

Arun Pandian

Approved by:

___________________________________, Committee Chair
Dr. Scott Gordon

___________________________________, Second Reader
Dr. Chung-E Wang

____
Date

Student: Arun Pandian

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the project.

____________________________, Graduate Coordinator ____________________
Dr. Behnam Arad Date

Department of Computer Science

Abstract

of

Training Neural Networks with Ant Colony Optimization

by

Arun Pandian

Ant Colony Optimization is a meta-heuristic approach to solve difficult optimization problems. Training a neural network is a process of finding the optimal set of its connection weights. So, a Continuous Ant Colony Optimization algorithm is used to train the neural network. In this project, the continuous Ant Colony Optimization (ACO) algorithm used to train neural networks was studied, implemented and tested with known training problems. Finally, the performance of this ACO implementation was compared with that of backpropagation and found to be less effective than backpropagation.

________________________, Committee Chair
Dr. Scott Gordon

____
Date

ACKNOWLEDGEMENTS

I take this opportunity to thank Dr. Scott Gordon for this guidance and support, his suggestions and ideas were very essential for the successful completion of this project. I also thank my parents for their endless love and support throughout my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

A Neural Network consists of a network of interconnected artificial neurons that try to mimic the functioning of biological neural networks in the human brain. Artificial neural networks can be trained so as to produce the desired set of outputs for a particular set of inputs, which allows a neural network to be used to solve complex classification and prediction problems which are difficult solve by conventional methods. Backpropagation is the most common algorithm used to train neural networks.

Training using backpropagation is a very slow and uncertain process; training failure arises from factors like network paralysis and local minima [1]. To overcome these drawbacks researchers are testing heuristic methods like Genetic Algorithms, Particle Swarm Optimization, Simulated Annealing, etc. to train neural networks more efficiently. One such swarm intelligence based meta-heuristic approach to problem solving is Ant Colony Optimization (ACO), which exploits the ability of an Ant Colony to find the shortest path between its nest and food source to solve problems. In this project ACO is used to train the neural network and its performance is compared with the performance of backpropagation, in the hopes that they will train neural networks more effectively.

Chapter 2

BACKGROUND

2.1 Artificial Neural Network

A Neural Network is a computational model that tries to simulate biological neural networks structurally and/or functionally. It consists of interconnected group of artificial neurons that process information using a connectionist approach.

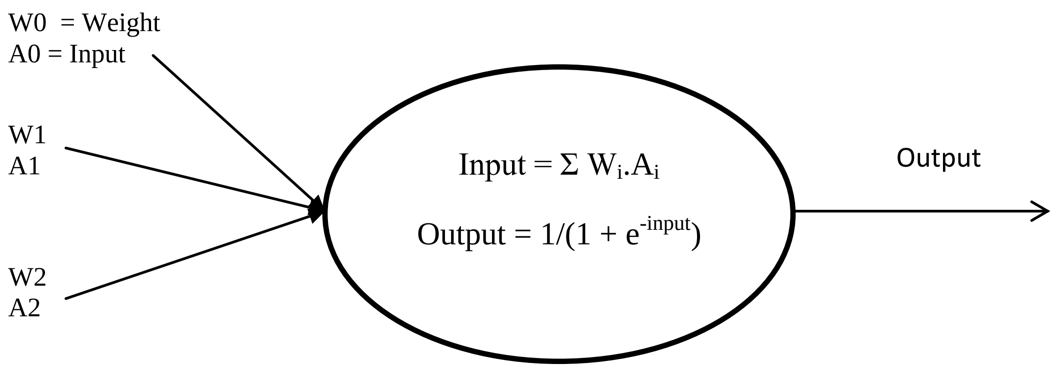

Figure 1 - Artificial Neuron

The structure of an artificial neuron is shown in Figure 1; the neuron calculates its input as

$$\text{Input} = \Sigma\, W_i\, A_i$$

Where, W_i is the weight associated with each input which has a value between 0 and 1 and A_i is either the actual input or output of another neuron in the previous layer. The output is calculated using the following sigmoid function:

$$\text{Output} = 1/(1 + e^{-\text{input}})$$

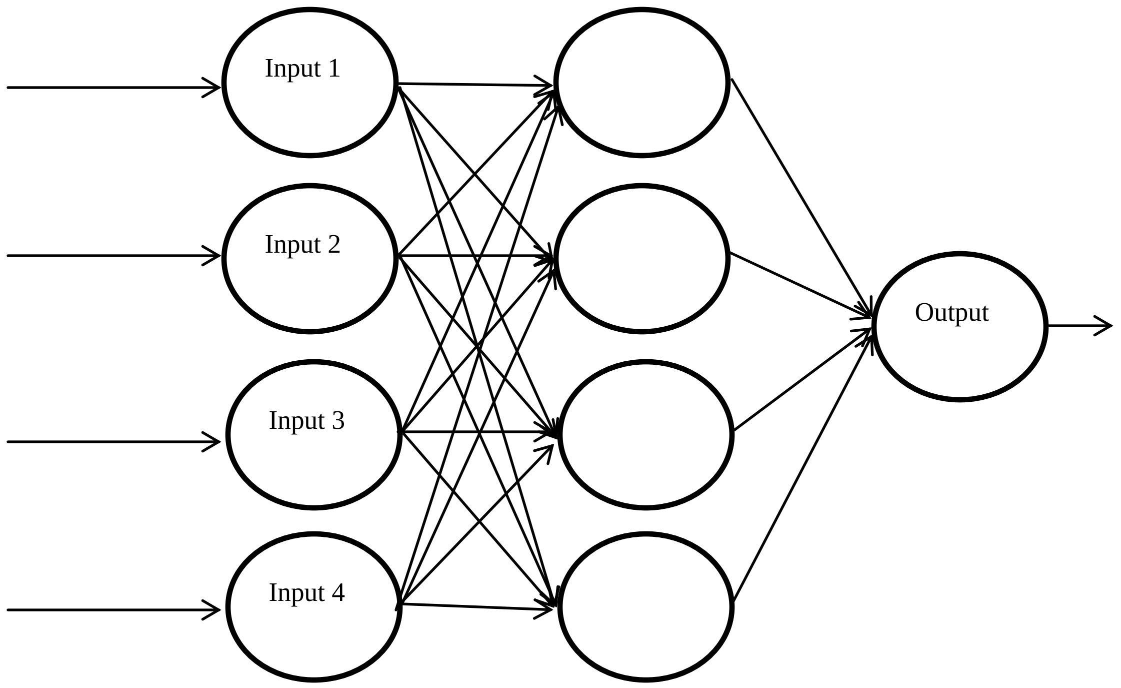

Figure 2 - Artificial Neural Network

Figure 2 shows the structure of a neural network with four inputs and one output. The input layer gets the actual input and sends it directly without any change as its output, the input then flows through the hidden layer and the final output is obtained at the output node.

Training a neural network is actually a process of finding the optimal set of weights for the links between neurons which will make the neural network produce the correct output corresponding to the given input. In backpropagation an iterative approach is used to find the correct set of weights. Backpropagation is a supervised learning method, which means a data set of inputs and its corresponding outputs for a particular problem is required to train the neural network. The inputs are applied to the neural network and the calculated output is compared with the actual output to get the error. The calculated error is then used to adjust the weights such that it is minimized. This process is repeated until the error is within a criterion value.

2.2 Ant Colony Optimization (ACO)

Ant colony optimization is a metaheuristic approach to solve difficult optimization problems. ACO is inspired by the foraging behavior of ants, which enables ants to find the shortest paths between food sources and their nest. In ACO, a set of artificial ants search the solution space for better solutions to a given problem.

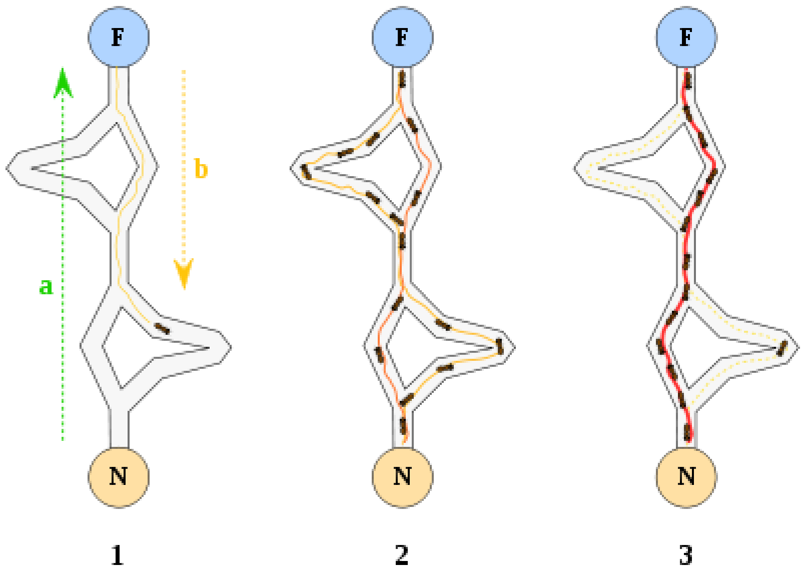

Figure 3 - Ant Colony Optimization [8]

The general idea of ACO is described in Figure 3. Initially the ants wander randomly until food is found, and when an ant finds food; it returns to its nest depositing pheromones on its way back. Pheromone has a property such that it evaporates with time and ants follow a pheromone trail which has the strongest concentration. Over a period of time more and more ants follow a trail which has the highest pheromone concentration which will also be the shortest path. Because, the pheromone deposit on longer trails evaporate faster than that of the shorter trails as it takes relatively more time for an ant to traverse that path. Finally every ant will follow the shortest path which will have the highest pheromone concentration as illustrated in step 3 of figure 4.

There are two types of ACO algorithms, continuous and discrete. ACO was originally developed to find the shortest path through a graph, which is a problem with a discrete solution space. A few ACO algorithms were developed later to solve problems with continuous solution space.

2.3 ACO and Neural Networks

The algorithm used in this project is belongs to the Continuous ACO category, as the solutions space problem of finding the optimal set of neural network weights is continuous. So, in this project the ACO algorithm for continuous optimization as defined by Krzysztof Socha and Christian Blum [2] is used. The abstract of the algorithm is given below:

1. Construct candidate solutions probabilistically using a probability distribution over the search space.
2. Use the candidate solutions to modify the probability distribution in a way such that future sampling is biased toward high quality solutions.

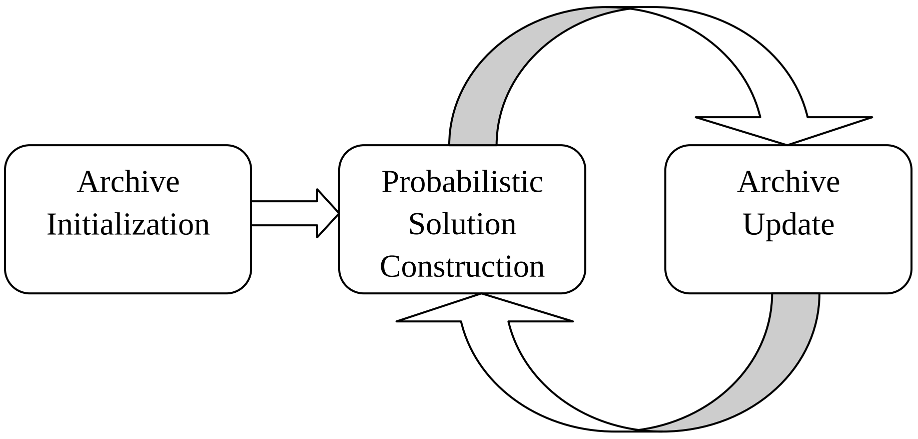

Figure 4 - ACO for Continuous Optimization

For combinatorial optimization problems ACO algorithms use the pheromone model to probabilistically construct solutions, the pheromone trail acts as the memory that stores the search experience of the algorithm. In ACO for continuous optimization the search is modeled with *n* variables, which is the number of neural network weights in our case. The candidate solutions are stored in an archive and are used to alter the probability distribution over the solution space. The probability distribution is analogous to the pheromone model used in combinatorial optimization problems.

Given below is the high level pseudocode of the algorithm:

```
START
Initialize archive with random values
WHILE True:
      Calculate fitness and sort the archive
      Select the top k solutions
      IF (fitness of first solution is acceptable)
            BREAK LOOP
      END IF
      Calculate the weights for each solution
      Generate k more solutions by sampling the
      Gaussian Kernel PDF (explained later)
END LOOP
PRINT solution
STOP
```

Figure 5 - ACO Pseudocode

Chapter 3

CONTINOUS ANT COLONY OPTIMIZATION ALGORITHM

3.1 Components

3.1.1 Archive

The algorithm maintains a solutions archive where are all the candidate solutions are stored; each candidate solution contains the values for all the ‘n’ variables that define the search space. In this case ‘n’ is equal to the number of weights in the neural network that is trained. The archive contains ‘k’ solutions.

3.1.2 Fitness vector

The fitness vector contains the result of the objective function $f(S_l)$ for all the solutions in the archive. This is the function that we are trying to minimize. In this case we are trying to minimize the sum squared error for our training set. The archive is sorted against the values of this vector.

3.1.3 Weight Vector

This vector contains the weight assigned to each solution, which is calculated using the formula [2]:

$$\omega_l = \frac{1}{q_k \sqrt{2\pi}} e^{-\frac{(l-1)^2}{2q^2k^2}}$$

Here, if the parameter q is small high quality solutions are preferred more and if it is large then the weight distribution is almost equal (l is the index of the solution in the archive).

3.1.4 Neural Network

A feed-forward multilayer neural network similar to the one described in figure 2 is used for this project

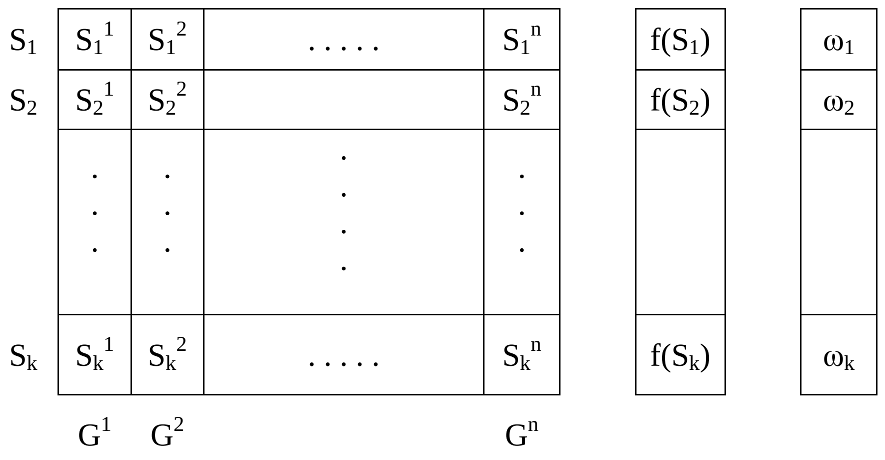

Figure 6 - Structure of Archive, Fitness vector and Weight Vector

3.2. Solution Construction

3.2.1 Archive Initialization

The archive is initialized with k random solutions with the value of each weight in the range [-1, 1].

3.2.2 Probability Density Function (PDF)

For the probabilistic construction of solutions we use the Gaussian PDF to define the probability distribution over the search space. Gaussian distribution is a bell shaped distribution centered at the mean. To create a Gaussian distribution we need two parameters μ -mean and σ -standard deviation.

3.2.3 Gaussian Kernel PDF

The Gaussian function has a disadvantage that has to be dealt with, due to the fact that the variation in the shape of the Gaussian function is limited, a single Gaussian function cannot be used to define a situation where there are two disjoint areas which are promising [2]. So, we use the enhanced Gaussian Kernel PDF which is a weighted sum of several one dimensional Gaussian functions g^i_l (x) as given below [2].

$$G^i(x) = \sum_{l=1}^{k} \omega_l g_l^i(x) = \sum_{l=1}^{k} \omega_l \frac{1}{\sigma_l^i \sqrt{2\pi}} e^{-\frac{(x-\mu_l^i)^2}{2{\sigma_l^i}^2}}$$

Where *k* = number of solutions in the archive, *l* = index of the solution and *i* = dimension.

The kernel PDF helps us to create an *n* dimensional probability distribution space as show in Figure 6 that can be sampled to generate a set of biased weights for the neural network under training.

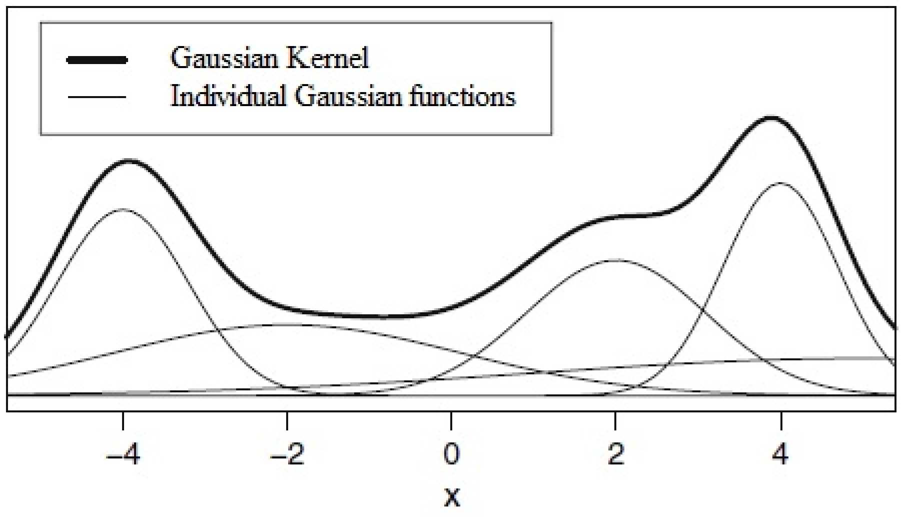

Figure 7 - Visualization of Gaussian Kernel PDF [2]

3.2.4 Sampling the Gaussian Kernel PDF

Each dimension of the solution has its own Gaussian Kernel PDF - G^i (x), constructed using the weights of the same dimension in the archive. To find the value for a dimension *i* (NN weight) of a newly constructed solution, $G^i(x)$ is sampled. Sampling is done in two phases.

1. One of the Gaussian functions that compose the Gaussian Kernel PDF is chosen with the probability of choosing the *l* th Gaussian function given below [2] .

$$p_l = \frac{\omega_l}{{}_{r=1}^{k}\,\omega_r}$$

2. The chosen Gaussian function is sampled to get the value for the dimension.

Once a Gaussian function is chosen, the same function (i.e. index l of the archive) is used to generate values for all the dimensions of that solution, each time by plugging in the values for mean μ$_l$ and σ$_l$ from each *i* of solution *l* in the archive.

- The mean is equal to the value of the variable i.e. the *i* th variable of all the solutions in the archive become the elements of vector μ^i.
- The standard deviation is calculated as the average distance between the chosen variable of the chosen solution to the same variable in all other solutions in the archive [2].

$$\sigma_l^i = \xi \sum_{e=1}^{k} \frac{x_e^i - x_l^i}{k-1}$$

Here, the standard deviation is multiplied by ξ, which is analogous to the pheromone evaporation rate in ACO for combinatorial optimization. High value for ξ results in low convergence speed [2].

Thus, all the n Gaussian Kernel PDF's are sampled to create a solution. This sampling is process is repeated to create *k* solutions (sets of neural network weights) for each training iteration.

3.3. Design and Implementation

The algorithm defined in section 3.2 is implemented in Java, the Apache commons math library [4] has been extensively used for parameterized random number generation and other mathematical operations.

The implementation of the training framework consists of four classes which are described in section 3.4 with suitable class diagrams. Source code is available in the Appendix section of this report.

3.4. Class Diagrams

3.4.1 Neuron

This class implements a simple neuron which can calculate its output and stores its output and all the weights for the incoming synapses.

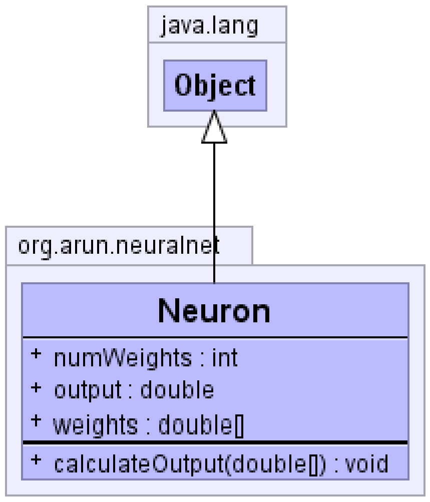

Figure 8 - Class Diagram - Neuron

3.4.2 NeuralNetwork

This class creates a neural network of any specified structure using a collection of neuron objects. Methods to set the connection weights of the neural network and calculate the output/error for input training/test data are defined in this class.

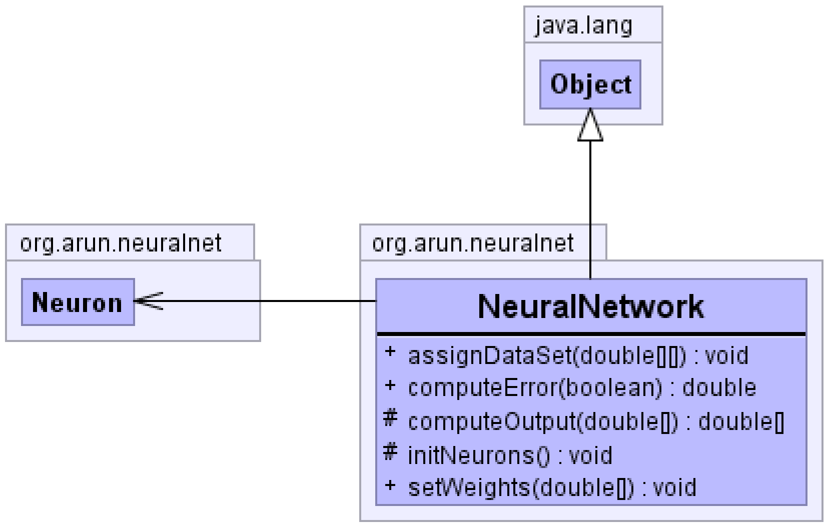
java.lang
Object
org.arun.neuralnet
Neuron
org.arun.neuralnet
NeuralNetwork
+ assignDataSet(double[][]) : void
+ computeError(boolean) : double
# computeOutput(double[]) : double[]
# initNeurons() : void
+ setWeights(double[]) : void

Figure 9 - Class Diagram - Neural Network

3.4.3 ACOFramework

The ACOFramework class contains all the data structures described in the continuous ACO algorithm. At a high level it performs the following tasks.

1. Initiates the archive with random values.
2. Creates and uses an instance of the NeuralNetwork object to determine the fitness of a solution.
3. Sorts the archive and generates biased candidate solutions by sampling a parameterized Gaussian random number generator.
4. Iterates the above process until the maximum number of iterations allowed is reached or if the training error criterion is satisfied.

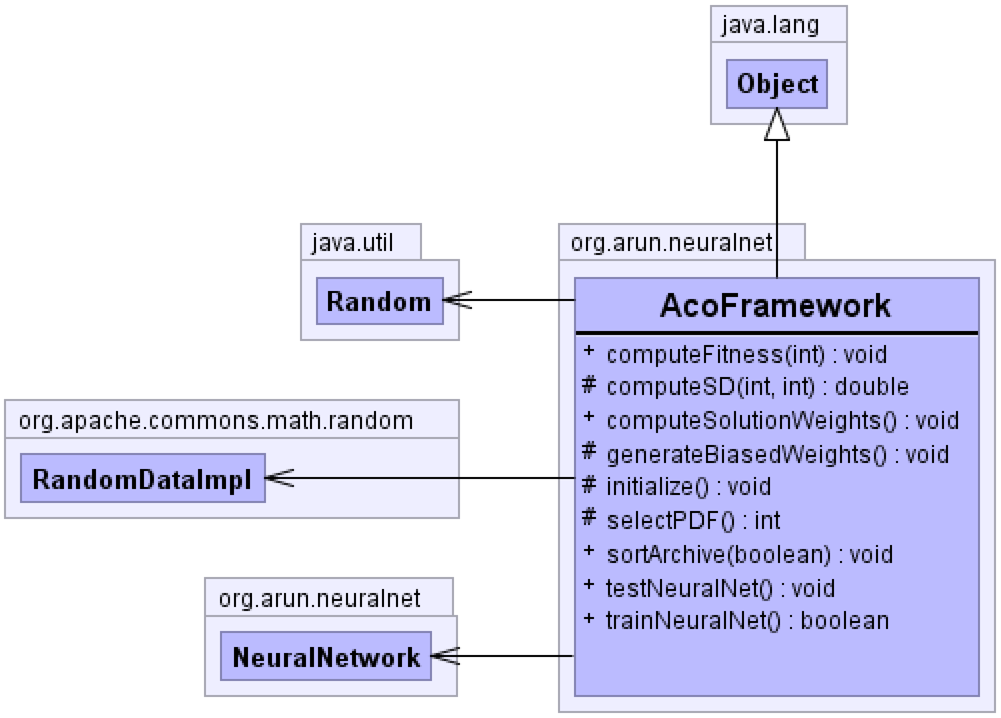
java.lang
Object
java.util
Random
org.arun.neuralnet
AcoFramework
+ computeFitness(int) : void
# computeSD(int, int) : double
+ computeSolutionWeights() : void
# generateBiasedWeights() : void
# initialize() : void
# selectPDF() : int
+ sortArchive(boolean) : void
+ testNeuralNet() : void
+ trainNeuralNet() : boolean
org.apache.commons.math.random
RandomDataImpl
org.arun.neuralnet
NeuralNetwork

Figure 10 - Class Diagram - ACOFramework

3.4.4 InputParser

This class contains methods to parse the training and test cases from a file of a predefined format. The data from the input files can then be scaled down up or scaled down for normalization purpose.

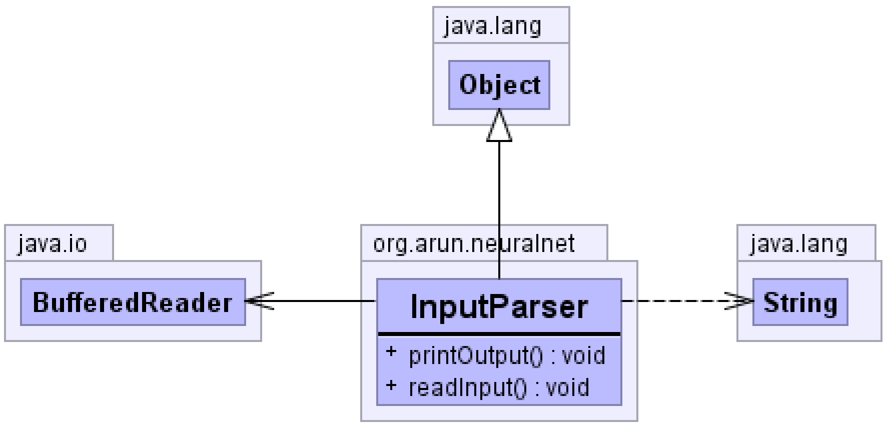

Figure 11 - Class Diagram - InputParser

Chapter 4

COMPARISON WITH BACKPROPAGATION

4.1 Datasets

Two test data sets given below are used from University of California, Irvine machine learning database.

1. **Seeds dataset** [6] - Measurements of geometrical properties of kernels belonging to three different varieties of wheat. A soft X-ray technique and GRAINS package were used to construct all seven, real-valued attributes.
2. **Glass dataset** [7] - From USA Forensic Science Service; 6 types of glass; defined in terms of their oxide content (i.e. Na, Fe, K, etc)

Both these datasets are used for comparing the performance of the described continuous ACO algorithm with that of backpropagation.

4.2 Comparison Metric

The metric used for comparing continuous ACO with back propagation is the number of passes performed on the neural network for training. The structure of the neural network used in both the algorithms is same. The back propagation algorithm used for comparison needs three runs through the neural network for each set of weights. One forward pass to calculate the output of the neural network, one backward pass to propagate the error backwards to all the nodes and another backward pass to adjust the weights of each connection.

In ACO, we just have one pass for each set of weights and the adjusted weights are generated by the algorithm. Each set of weights in the solution repository is given a forward run to calculate the error and the algorithm uses the error as a variable to calculate another biased set of weights. So, a training iteration has three passes for backpropagation and *k* passes for ACO, where *k* is the size of the repository used to store the weights.

So, the number of training iterations required for the neural network to converge to the acceptable error rate for ACO can be considered to be equivalent to that of backpropagation as given below.

$$N_{BP} * 3 \equiv N_{ACO} * N_{train} * N_{archive}$$

Where,

N_{BP} is the number of backpropagation iterations.

N_{ACO} is the number of ACO iterations

N_{train} is the number of training cases in the dataset.

$N_{archive}$ is the number of solutions in the archive.

4.3 Continuous ACO Parameter analysis

The parameter *q* which is used when each solution in the archive is assigned a weight using the expressing described in 3.1.3. When the value is q is small solutions with high fitness are highly preferred, when it’s large the probability of choosing a solution is uniformly distributed, allowing more randomness. The optimal value is found to be 0.85

for both the datasets. The number iterations required for the neural net to converge with different values of *q* is described in the table below.

Table 1 - Analysis of parameter q

q	**Iterations for Seeds dataset**	**Iterations for glass dataset**
0.06	59056	Did not converge
0.07	31789	56534
0.08	23788	37789
0.09	29682	46523

The parameter ξ is analogous to the pheromone evaporation rate in ACO; this affects the long term memory implicit to the archive. Higher the value of ξ, slower the convergence speed, i.e. low quality solutions are forgotten faster and will have a lower influence on constructing new solutions. Performance of the algorithm with different values for ξ with the optimal value 0.08 for *q* is compared in Table 2.

Table 2 - Analysis of parameter ξ

ξ	**Iterations for Seeds dataset**	**Iterations for Glass dataset**
0.65	Did not converge	Did not converge
0.75	56789	61945
0.80	29378	47834
0.85	23788	37789
0.9	36784	54578

An archive size of 20 is used for all the test data generated. After studying different values, the results indicate that the archive size itself does not affect the efficiency of the algorithm for the test data sets used for comparison.

4.4 Comparison with Backpropagation

From the data collected in Table 1 and Table 2, the optimal values for ξ and q for the studied datasets are 0.85 and 0.08 respectively. A continuous ACO algorithm with the optimal parameter values are compared with backpropagation algorithm run for the same datasets, the raw iterations required for the neural network to converge within the error criterion is are populated in Table 3.

Table 3 – ACO vs. Backpropagation

Training Algorithm	**Raw Iterations - Seeds dataset**	**Raw Iterations - Glass dataset**
Backpropagation	3936163	5678934
Continuous ACO	23788	37789

As described in section 4.2 the number of passes through the neural network that is trained is used as the comparison metric. For backpropagation, there are three passes through the neural network for each training iteration. So the number of passes is three times the number of iterations needed for the network to converge. For ACO, one iteration equals the number of raw ACO iterations times the number of training case passes times the archive size. The raw iterations required by ACO for each dataset is normalized and populated in Table 4 for comparison against backpropagation.

From the comparison results in Table 4 it is clear that the ACO algorithm used in this project on the selected datasets performed poorly when compared with backpropagation. The ACO algorithm was only 12% and 3% efficient as backpropagation for the Seeds and Glass datasets respectively.

Table 4 - Continuous ACO performance analysis

Dataset	**Training cases**	**Backpropagation Iterations**	**Raw ACO Iterations**	**Normalized ACO Iterations**	**ACO efficiency**
Seeds	210	3936163	23788	33303200	12%
Glass	214	5678934	37789	161736920	3%

Chapter 5

CONCLUSIONS AND FUTURE WORK

Training a neural network is the process of finding the optimal set of weights for connections between neurons in the network; the values for the neural network weights form a continuous solutions space. An ACO algorithm to solve continuous optimization problems has been identified and studied. The studied algorithm is implemented in Java and is used to train the neural network for the Seeds and Glass datasets. The parameters that govern the algorithm behavior are studied and the findings are documented to find their optimal values for the dataset under study. A metric is defined for comparing ACO with backpropagation and their performance is compared against the same datasets and documented. From the test results it is clear that this implementation of the continuous ACO algorithms performs poorly when compared to backpropagation for the datasets used.

ACO parameters can further be investigated and optimized. Other probability distribution methods need to be investigated for solution weight calculation and neural network weight calculation. Also, each solution in the archive can be improved using other neural network training methods to investigate the performance of algorithms hybridized with ACO.

Appendix: Source Code

Main.java

```
package org.arun.neuralnet;

/**
 *
 * @author arun
 */
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        double[][] trainingData, testData;
        int numInputs = 7;
        int numOutputs = 1;
        int numTrainCases = 210;
        int numTestCases = 20;
        int repoSize = 20; //size of the repository

        String trainingFilePath = "C:\\project\\bp\\seeds_dataset.dat";
        String testFilePath = "C:\\project\\bp\\seeds_dataset_test.dat";

        /*
         * read training and test data.
         */
        InputParser training_cases =
                new InputParser(trainingFilePath, numInputs, numOutputs,
                numTrainCases);
        InputParser test_cases =
                new InputParser(testFilePath, numInputs, numOutputs,
                numTestCases);
        training_cases.readInput();
        test_cases.readInput();

        /*
         * Scale down training and test data.
         */
        trainingData = training_cases.scaleDown();
        testData = test_cases.scaleDown();

        System.out.println("Scaled down test data:\n");
        for (int i=0;i<numTrainCases;i++) {
            for (int j=0;j<numInputs+numOutputs;j++)
                System.out.print(trainingData[i][j] + " ");
            System.out.println("");
        }
        System.out.println("");
        System.out.println("End: Test data:");

        int[] numPerLayer = new int[3];
        numPerLayer[0] = numInputs;
        numPerLayer[1] = numInputs;
        numPerLayer[2] = numOutputs;
        NeuralNetwork nn = new NeuralNetwork(3, numInputs, numPerLayer);
        nn.assignDataSet(
trainingData);
        AcoFramework AF = new AcoFramework(nn, repoSize);

        if (AF.trainNeuralNet()) {
            System.out.println("******Test Output*****");
            nn.assignDataSet(testData);
            AF.testNeuralNet();
        }
        else {
            System.exit(0);
        }
    }
}
```

InputParser.java

```
package org.arun.neuralnet;

import java.io.FileReader;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
/**
 *
 * @author arun
 */
public class InputParser {

    BufferedReader file;
    int numInput;
    int numOutput;
    int numCases;
    double[][] trainingCases;
    double[] output;
    double[][] extrema;


    InputParser(String fileName, int numIns, int numOuts, int numLines) {
        this.numInput = numIns;
        this.numOutput = numOuts;
        this.numCases = numLines;
        trainingCases = new double[numCases][numInput+numOutput];
        extrema = new double[numIns+numOuts][2];

        for (int i=0;i<(numIns+numOuts);i++) {
            extrema[i][0] = 10000.0;
            extrema[i][1] = -10000.0;
        }

        try {
            file = new BufferedReader(new FileReader(fileName));
        }
        catch (FileNotFoundException ex) {
            System.out.println("Invalid input file!");
        }
    }

    public void readInput() {
        try {
            for (int i=0;i<numCases;i++) {
                String[] line = (file.readLine().split("\\s+"));
                //String[] line = (file.readLine().split(","));
                for(int j=0;j<(numInput+numOutput);j++) {
                    trainingCases[i][j] = Double.parseDouble(line[j]);

                    if (trainingCases[i][j] < extrema[j][0])
                        extrema[j][0] = trainingCases[i][j];

                    if (trainingCases[i][j] > extrema[j][1])
                        extrema[j][1] = trainingCases[i][j];
                }
            }
            //test whether both extrema are equal
            for (int i=0; i < (numInput+numOutput); i++)
```

```
                if (extrema[i][0] == extrema[i][1])
                    extrema[i][1]=extrema[i][0]+1;


            System.out.println("*************Extrema values*************");
            for (int i=0; i<(numInput + numOutput); i++)
                System.out.println(extrema[i][0] + " " + extrema[i][1]);
            System.out.println("*************End*************");

        } catch (Exception ex) {
            ex.printStackTrace();
            System.out.println("Error while reading file!");
            System.exit(0);
        }
    }

    public void printOutput() {
        for (int i=0;i<numCases;i++) {
            for (int j=0;j<(numInput+numOutput);j++)
                System.out.print(trainingCases[i][j] + " ");

            System.out.println("");
        }
    }

    /*****************************************
    Scale Desired Output to 0..1
    *****************************************/
    double[][] scaleDown() {
        double[][] scaledDownInput;
        int i,j;
        scaledDownInput = new double[numCases][numInput+numOutput];

        for (i=0;i<numCases;i++)
            for (j=0;j<numInput+numOutput;j++) {
                scaledDownInput[i][j]  =
                        .9*(trainingCases[i][j]-extrema[j][0])/
                        (extrema[j][1]-extrema[j][0])+.05;
            }
        return scaledDownInput;
    }

    /*****************************************
    Scale actual output to original range
    *****************************************/
    double scaleOutput(double X, int which) {
        double range = extrema[which][1] - extrema[which][0];
        double scaleUp = ((X-.05)/.9) * range;
        return (extrema[which][0] + scaleUp);
    }
}
```

AcoFramework.java

```
package org.arun.neuralnet;

import java.util.Random;
import org.apache.commons.math.random.RandomDataImpl;

/**
 *
 * @author arun
 */
public class AcoFramework {
    double[][] archive; /*Archive of NN weights*/
    int numLayers = 3; /*number of layers in the Neural Network*/
    /*array containing the number of nodes in each layer*/
    int[] nodesPerLayer;
    int numIns; /*number of inputs*/
    int numOuts; /* number of outputs*/
    int archiveSize; /*number of sets of NN weigthts in the archive*/
    int numWeights = 0; /*total number of weights in the neural network*/
    /*array to store the fitness value of each NN weight set*/
    double[] fitness;
    double[] solWeights; /*weight for each solution */
    double sumSolWeights = 0; /*sum of all solution weights*/
    double[] solution;
    NeuralNetwork nn; /*Neural network to be trained*/
    double epsilon = .85; /*affects pheromone evaporation rate*/
    double q = .08;
    Random rand;
    RandomDataImpl grand;
    double test_error = -1;
    double constant_sd = 0.1;
    int maxIterations = 100000;
    double errorCriteria = 0.09;

    /*
     *Constructor that creates the ACO framework.
     *Takes a neural network and the size of the archive as parameters.
     */
    AcoFramework(NeuralNetwork neuralNet, int archive_Size) {

        nn = neuralNet;
        archiveSize = archive_Size;
        nodesPerLayer = nn.nodesPerLayer;
        numIns = nodesPerLayer[0];
        numOuts = nodesPerLayer[nn.columns-1];
        numWeights = nn.numWeights;
        initialize();
    }

    /*
     * Method to create a initialize the archive with random weights.
     */
    protected void initialize() {
        int i,j;
        archive = new double[archiveSize*2][numWeights];
        fitness = new double[archiveSize*2];
        solWeights =  new double[archiveSize];
        rand = new Random();
    grand = new RandomDataImpl();

    /*
     * fill archive with random values for weights
     */
    for (i=0;i<archiveSize*2;i++)
        for (j=0;j<(numWeights);j++) {
            archive[i][j] = rand.nextDouble()*2-1;
        }
}

/*
 * Method to compute the fitness of all the weight sets in the archive.
 */
public void computeFitness(int type) {
    if (type == 0) {
        for (int i=0;i<archiveSize*2;i++) {
            nn.setWeights(archive[i]);
            fitness[i] = nn.computeError(true);
        }
    }
    if (type == 1) {
        nn.setWeights(solution);
        test_error = nn.computeError(false);
        System.out.println("Test error: " + test_error);
    }
}

/*
 *Implementation of bubble sort algorithm to sort the archive according
 *to the fitness of each solution. This method has a boolean parameter,
 *which is set to true if the method is called to sort the just initialized
 * array.
 */

 public void sortArchive(boolean init) {
    int i,j;
    int n = archive.length;

    for (i=0;i<n;i++)
        for (j=0;j<n-1;j++) {
            try {
            if (fitness[j] > fitness[j+1]) {

                double temp = fitness[j];
                fitness[j] = fitness[j+1];
                fitness[j+1] = temp;

                double[] tempTrail = archive[j];
                archive[j] = archive[j+1];
                archive[j+1] = tempTrail;
            }
            }
            catch(Exception e) {
                System.out.println("error: " + j+" n "+n);
                System.exit(0);
            }
        }
}
```

```
      /*
       * Method to compute the weights for each set of weights in the archive
       */
      public void computeSolutionWeights() {
          sumSolWeights = 0;
          for (int i=0;i<archiveSize;i++) {
              double exponent = (i*i)/(2*q*q*archiveSize*archiveSize);
              solWeights[i] =
              (1/(0.1*Math.sqrt(2*Math.PI)))*Math.pow(Math.E, -exponent);
              sumSolWeights += solWeights[i];
          }
      }


      /*
       * Method to calculate the standard deviation of a particular weight of a
       * particular weight set
       */
      protected double computeSD(int x,int l) {
          double sum=0.0;
          for (int i=0;i<archiveSize;i++) {
              sum += Math.abs(archive[i][x] - archive[l][x])/(archiveSize-1);
          }
          if(sum ==0) {
              System.out.println("sum = 0 "+ l + "archivesize = " +
archiveSize);
              return constant_sd;
          }
          return (epsilon*sum);
      }

      /*
       * select a Gaussian function that compose the Gaussian Kernel PDF
       */
      protected int selectPDF() {
          int i, l=0;
          double temp=0, prev_temp = 0;
          double r = rand.nextDouble();

          for (i=0;i<archiveSize;i++) {
              temp += solWeights[i]/sumSolWeights;
              if (r<temp) {
                 l=i;
                 break;
              }
          }

          return l;
      }

      protected void generateBiasedWeights() {
          int i,j,pdf;
          double sigma; /*standard deviation*/
          double mu; /*mean*/

          pdf = 0;
          for (i=archiveSize;i<archiveSize*2;i++) {
              pdf = selectPDF();
              for (j=0;j<numWeights;j++) {
                  sigma = computeSD(j,pdf);
```

```
                mu = archive[pdf][j];
                archive[i][j] = grand.nextGaussian(mu, sigma);
            }
        }
    }

    public boolean trainNeuralNet() {
        computeFitness(0);
        sortArchive(true);
        computeSolutionWeights();
        generateBiasedWeights();
        sortArchive(false);

        for (int j=0;j<maxIterations;j++) {
            computeFitness(0);
            sortArchive(false);
            if (j%1000 == 0)
               System.out.println(fitness[0]);
            if (fitness[0] < errorCriteria) {
                System.out.println("Solution found in iteration" + (j+1));
                solution = archive[0];
                for (int i=0;i<numWeights;i++)
                    System.out.print(archive[0][i]);
                return true;
            }
            computeSolutionWeights();
            generateBiasedWeights();
        }
        System.out.println("Network did not converge!");
        return false;
    }

    public void testNeuralNet() {
        computeFitness(1);
    }
}
```

NeuralNetwork.java

```
package org.arun.neuralnet;
/**
 *
 * @author arun
 */
public class NeuralNetwork {
    /* 2-dimensional array of neurons that represent the neural network*/
    Neuron[][] neuralNet;
    int columns;            /*number of columns including input layer*/
    int maxRows;            /*maximum number of rows in any column*/
    int[] nodesPerLayer;    /*number of nodes in each layer*/
    int numIns;
    int numOuts;
    int numWeights=0;
    double[][] dataSet;


    NeuralNetwork(int numLayers, int mRows, int[] numPerLayer) {
        this.maxRows = mRows;
        this.columns = numLayers;
        this.nodesPerLayer = numPerLayer;
        this.neuralNet = new Neuron[columns][maxRows];
        numIns = nodesPerLayer[0];
        numOuts  = nodesPerLayer[numLayers-1];

        for (int i=1;i<columns;i++)
            numWeights = numWeights + (nodesPerLayer[i]*nodesPerLayer[i-1]);
        initNeurons();
    }

    protected void initNeurons() {
        int i,j;
        for(i=0;i<numIns;i++)
            neuralNet[0][i]  = new Neuron(0);

        for (i=1;i<columns;i++)
            for (j=0;j<nodesPerLayer[i];j++)
                neuralNet[i][j] = new Neuron(nodesPerLayer[i-1]);
    }

    public void setWeights(double[] weights) {
        int x=0;

        if(weights.length != numWeights) {
            System.out.println("mismatch in number of weights");
            System.exit(1);
        }

        for (int i=1;i<columns;i++)
            for (int j=0;j<nodesPerLayer[i];j++)
                for (int k=0;k<neuralNet[i][j].numWeights;k++)
                    neuralNet[i][j].weights[k] = weights[x++];
    }

    public void assignDataSet(double[][] data_set) {
        this.dataSet = data_set;
    }
    protected double[] computeOutput(double[] inputs) {
        double sum;
        int i,j,k;
        double[] output = new double[nodesPerLayer[columns-1]];

        /*Input Layer*/
        for (i=0;i<nodesPerLayer[0];i++)
            neuralNet[0][i].output = inputs[i];

        /*Hidden Layers*/
        for(i=1;i<columns-1;i++)
            for (j=0;j<nodesPerLayer[i];j++) {
                sum = 0.0;
                for(k=0;k<nodesPerLayer[i-1];k++)
                    sum += neuralNet[i][j].weights[k]*neuralNet[i-1][k].output;
                neuralNet[i][j].output = 1.0/(1.0 + Math.exp(-sum));
                }

        /*output Layer*/
        for (i=0;i<nodesPerLayer[columns-1];i++) {
            sum=0.0;
            for (j=0;j<nodesPerLayer[columns-2];j++)
                sum += neuralNet[columns-1][i].weights[j]
                        *neuralNet[columns-2][j].output;
            output[i] =
                    neuralNet[columns-1][i].output = 1.0/(1.0+Math.exp(-sum));
        }
        return output;
    }

    public double computeError(boolean training) {
        int i,j;
        double[] input = new double[nodesPerLayer[0]];
        double[] output = new double[nodesPerLayer[columns-1]];
        double desiredOutput[] = new double[nodesPerLayer[columns-1]];
        double totalError = 0.0;
        double temp;

        for (i=0;i<dataSet.length;i++) {
            for (j=0;j<numIns;j++)
                input[j] = dataSet[i][j];
            for (j=0;j<numOuts;j++)
                desiredOutput[j] = dataSet[i][numIns+j];
            output = computeOutput(input);

            if (!training) {
               temp = (((output[0]-0.05)/0.9)*2) + 1;
               System.out.println(Math.round(temp));
            }

            for (j=0;j<numOuts;j++) {
                double error = output[j] - desiredOutput[j];
                totalError += error*error;
            }
        }
        return totalError;
    }
}
```

Neuron.java

```
package org.arun.neuralnet;

/**
 *
 * @author arun
 */
public class Neuron {
    public double output;
    public double[] weights;
    public int numWeights;

    Neuron(int numWeights) {
        this.numWeights = numWeights;
        this.weights = new double[this.numWeights];
    }

    public void calculateOutput(double[] input) {
        output=0;
        for (int i=0;i<numWeights;i++) {
            output+=weights[i]*input[i];
        }
    }
}
```

Bibliography

[1] Philip D. Wasserman, Neural Computing: Theory and Practice, Coriolis group (1989)

[2] Krzysztof Socha and Christian Blum. “An ant colony optimization algorithm for continuous optimization: application to feed-forward neural network training”, in Springer London (2007).

[3] M.Dorigo, V.Maniezzo, and A.Colorni. “Ant System: Optimization by a colony of cooperating agents”, in IEEE Transactions on Systems, Man, and Cybernetics, 1996.

[4] Apache Commons Math 1.2 API (05/14/2010), *http://commons.apache.org/math/api-1.2/overview-summary.html.*

[5] Normal Distribution (05/14/2010), *http://mathworld.wolfram.com/NormalDistribution.html.*

[6] Glass Identification Data Set from USA Forensic Science Service, 6 types of glass defined in terms of their oxide content (09/01/1987). *http://archive.ics.uci.edu/ml/datasets/Glass+Identification*

[7] Seeds dataset - Measurements of geometrical properties of kernels belonging to three different varieties of wheat (09/20/2012). *http://archive.ics.uci.edu/ml/datasets/seeds*

[8] Wikipedia – Ant Colony Optimization Algorithms, http://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms