

Compound Swarm Optimization of Neural Network Weights

Magnus Erik Hvass Pedersen (971055)
Daimi, University of Aarhus, April 2004

1 Introduction

The purpose of this document is to verify attendance of the author to the *Knowledge Based Systems* course at DAIMI, University of Aarhus. First an introduction is given to the kind of Neural Network that is to be used in the ensuing experiments, then various algorithms for finding its parameters are described, and this is finally applied to real-world examples.

The reader is assumed to be familiar with meta-heuristics and related topics, and [10] is used throughout without explicit citation.

2 Artificial Neural Networks

Modelling the human brain, an *Artificial Neural Network* (NN) consists of a number of simple, interconnected processing units that correspond to the *neurons* of the brain.

Conceptually though, the network can be considered a *black box* that simply maps input to output, without having explicit knowledge of the given domain. We are then interested in a network that yields sensible output, when presented with input that it has never experienced before.

Training the network to do this, is done with a data-set D containing examples of input and output, that is, the elements of D are of the form (\vec{x}_d, \vec{t}_d) , with $\vec{x}_d \in \mathbb{R}^n$ being the input to the network, and $\vec{t}_d \in \mathbb{R}^m$ being its desired output.

The network is then iteratively modified so that it produces more and more accurate output \vec{o}_d for the input \vec{x}_d , as measured by the *Sum of Squared Errors* (SSE) between the actual and desired output:¹

$$\text{SSE}(\vec{t}, \vec{o}) = \sum_{i=1}^m (t_i - o_i)^2$$

The error over all data samples in $T \subseteq D$ is given simply by their sum:

$$\text{SSE}(T) = \sum_{(\vec{x}_d, \vec{t}_d) \in T} \text{SSE}(\vec{t}_d, \vec{o}_d)$$

¹Some texts also multiply this sum by $\frac{1}{2}$.

This can be normalized by the number of samples in T , allowing for easy comparison between data-sets of different sizes. This is known as the *Mean Squared Error* (MSE), and is defined as:

$$E(T) = \text{MSE}(T) = \frac{1}{|T|} \text{SSE}(T) \quad (1)$$

2.1 Classification

For many kinds of data, we are interested in determining what class some input belongs to. In the cancer experiments below, there are two classes, *benign* and *malignant*, but it could also be whether a client is credit-worthy or not, or what direction a person is looking.

One way of encoding discrete classes in our real-coded network, is of course to designate different values to different classes, so the output of the network is supposed to be a single real value: $t_d \in \mathbb{R}$, so that e.g. $t_d = 0$ corresponds to benign and $t_d = 1$ is then malignant.

Another way of encoding this, is with so-called 1-of- n classification, in which all classes have separate real-valued output. In this example, we would then have $\vec{t}_d \in \mathbb{R}^2$, with $\vec{t}_d = [1, 0]$ representing a benign tumour and $\vec{t}_d = [0, 1]$ a malignant.

One advantage to this kind of encoding, is that the degree of classification uncertainty can be assessed in terms of what classes are confused for the given data example. This becomes more evident with more possible classes, take for example the facial direction experiments described below, where the encoding being used, reveals if the network confuses e.g. whether the person is looking up or straight ahead.

2.2 Sigmoidal Feed-Forward Network

The kind of NN studied here, is a so-called *Sigmoidal Feed-Forward* network. The processing *nodes* are layered as in figure 1, with the top layer being the input of the network, and the bottom its output.² Between these there are a number of *hidden* layers, that give rise to more complex mappings.

Since it is a *feed-forward* network, the nodes in one layer only feed into the nodes of the following layer. If we further assume the network to be *fully connected*, then a node feeds *all* of the nodes in the following layer.

2.3 Mathematical Representation

More specifically, let N denote the number of layers, including input and output layers. The number of nodes in layer $i \in \{1, \dots, N\}$ is given by N_i , and the value of these nodes is $\vec{n}_i \in \mathbb{R}^{N_i}$. The first layer's values is simply the current input to the network $\vec{n}_1 = \vec{x}$, and the output of the network, is its last layer: $\vec{o} = \vec{n}_N$.

²In the literature, the word *layers* is also sometimes used to describe the connection-levels.

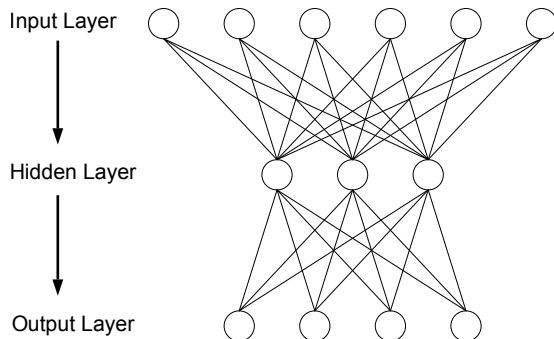


Figure 1: The structure and flow of a fully connected, feed-forward neural network having 6 input and 4 output nodes, and with a single hidden layer containing 3 nodes.

Apart from the weighted sum of the previous layers' nodes, we furthermore need to add a constant factor or node-*bias*, and finally saturate this sum. The complete processing required to find the value of a node, is shown in figure 2.

Let $\vec{w}_{ij} \in \mathbb{R}^{N_{i-1}}$ denote the *ingoing* weights used for calculating the sum l_{ij} for the j 'th node in the i 'th layer, that can then be written as a dot-product:

$$l_{ij} = \vec{w}_{ij} \cdot \vec{n}_{i-1}$$

Where the actual node-value n_{ij} is then calculated as:

$$n_{ij} = \sigma(l_{ij} + b_{ij}) \quad (2)$$

with $b_{ij} \in \mathbb{R}$ being the bias-value,³ and the *Sigmoid* function $\sigma : \mathbb{R} \rightarrow (0, 1)$ is defined as:

$$\sigma(y) = \frac{1}{1 + e^{-y}} \quad (3)$$

The Sigmoid function *compresses* or *saturates* its input to the restricted range $(0, 1)$. Since the network's output is also saturated with this function, we often need to encode the \vec{t}_d values of the data-set D , and interpret them in a certain way, such as the 1-of- n classification scheme described above.

2.3.1 Weight Matrix

Consider the following matrix for layer i , which has the weights for node j as the j 'th row:

$$W_i = \begin{bmatrix} -\vec{w}_{i1} - \\ -\vec{w}_{i2} - \\ \vdots \\ -\vec{w}_{iN_i} - \end{bmatrix}$$

³In the litterature commonly referred to as a weight also.

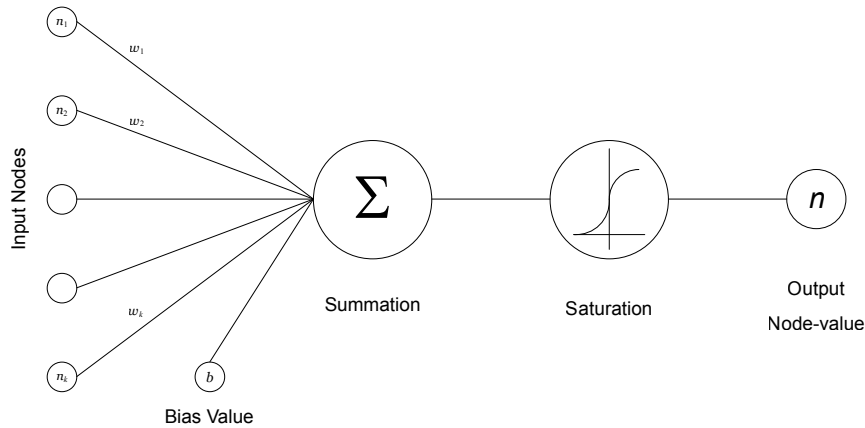


Figure 2: The calculation of a node’s value in a Sigmoidal Feed-Forward Network. The previous layer’s nodes are summed along with a bias-value. These are then saturated to the range $(0, 1)$ by the Sigmoid-function, resulting in the node-value n . Node-values for both the output and hidden layers are found this way.

This gives us a particularly simple equation for calculating the values of nodes in the output and hidden layers, that is, for $i \in \{2, \dots, N\}$ we get the intermediate weighted sums by multiplying the weight-matrix W_i with the previous layer’s node-values \vec{n}_{i-1} :

$$\vec{l}_i = W_i \vec{n}_{i-1}$$

So the actual node-values \vec{n}_i for layer i are:

$$\vec{n}_i = \hat{\sigma}(\vec{l}_i + \vec{b}_i) \tag{4}$$

Where $\vec{b}_i \in \mathbb{R}^{N_i}$ holds the bias values for that layer, and the vector-version of the Sigmoid function, $\hat{\sigma} : \mathbb{R}^n \rightarrow \mathbb{R}^n$, is simply defined as:

$$\hat{\sigma}(\vec{x}) = (\sigma(x_1), \dots, \sigma(x_n))$$

3 Architecture & Parameters

When building a neural network, we must decide upon a number of design-issues belonging to the following two categories:

- **Architecture**; is the deciding of network type (e.g. feed-forward), connectivity (e.g. fully connected), node types (e.g. sigmoidal), and the number of hidden layers and nodes.

- **Parameters**; deals with the selection of auxiliary constants to minimize the error of the network. These are typically the weights W_i on the connections between nodes, as well as the bias-values \vec{b}_i , but we may also wish to decide the optimal slope for the Sigmoid function at each node.

This document focuses on the optimization of the standard parameters, namely the weights W_i and bias values \vec{b}_i .

3.1 Genetic Programming

That it is possible to automatically *evolve* the network architecture deserves brief mentioning however.

The evolutionary paradigm known as *Genetic Programming* (GP) is a special kind of *Genetic Algorithm* (GA) that iteratively evolves more complex structures such as mathematical expressions or entire programs, whereas GAs are used for simpler problems such as fixed-length optimization problems over a subset of \mathbb{R}^n .

The underlying idea is the same though: A population P of possible solutions, or *chromosomes* as they are called in this genetic context, is randomly initialized. For each chromosome we associate a fitness which is the ability of the suggested solution to solve the problem at hand. Then at each step of the algorithm, two chromosomes are selected according to their individual fitnesses, and one or more new chromosomes are created by *crossover*, i.e. mixing the features of the parents, and by *mutation*, which is the alteration of existing features or the introduction of new features. For a more detailed description of the GA algorithm see [5].

In [1], a GP method is used for evolving both the network architecture and parameters. Although the method is applied to binary input and output, there is no apparent reason why it should not work for continuous-valued input and output.

The method works by having the chromosomes encode neural networks as so-called LISP S-expressions, which are structured as trees in a manner similar to *Abstract Syntax Trees*, the kind of data structure used by compilers for internal representations of e.g. mathematical expressions, statement-sequences, and entire programs. Here however, the leafs of such a tree are either inputs of the network or randomly chosen constants. The tree-nodes have arithmetic and weighting operators, and a *threshold* node is the binary equivalent of the continuous-valued Sigmoid (threshold) function σ in Eq.(3), which here outputs a value in $\{0, 1\}$ depending on whether its real-valued input exceeds a given threshold.

There are some restrictions on the encoding of the network, that must be enforced after recombination of the genetic parents or mutation, but the method still allows for more general network architectures than the fully connected feed-forward network studied here.

One restriction, for example, is that the root of a one-output network encoded as an S-expression, must be a threshold-node. This is natural however,

since the values flowing internally in the network are real-valued, but the output is expected to be binary, which is exactly what the given threshold unit produces. For multi-output networks, several S-expressions for single-output networks can be *concatenated* with the use of the LIST-construct.

3.2 Sigmoidal Slopes & Node Pruning

In [2] not only the weights and bias-values are adjusted, but also the slopes of the Sigmoid-function. This can be done by altering $k \in \mathbb{R}$ of the following extension to Eq.(3):

$$\sigma(y) = \frac{1}{1 + e^{-k \cdot y}}$$

And according to the value of k , the Sigmoid-function may be replaced by other processing units, for example a step-function in case of large k .

The entire node may be removed from the network if k approaches 0, where the output of the Sigmoid function thus approaches a constant value of $\frac{1}{2}$. The bias-values in the next layer are then adjusted accordingly, and so, the method can be used indirectly to prune the network architecture, thus decreasing its complexity.

3.3 Back-Propagation

A traditional approach to finding the network parameters (i.e. weights and node bias values), is known as *Stochastic Back-Propagation* (BP). The algorithm initializes the parameters with small random values, and then iteratively refines them by seeking to minimize the output error $E(T)$, when processing the training data T . The BP algorithm is then as follows:

- Initialize all weights and node bias values with small random values, e.g. uniformly in the range $[-0.05, 0.05]$.
- Until a termination criterion is met (e.g. number of iterations, stagnation of fitness, or fitness-threshold) repeat the following:⁴

– For each training pair $(\vec{x}, \vec{t}) \in T$, do the following:

- * First calculate all node-values \vec{n}_i for each layer i , using Eq.(4) or the equivalent Eq.(2).
- * Now calculate the errors $\vec{\delta}_N$ for all nodes in the output layer:

$$\delta_{Nj} = n_{Nj}(1 - n_{Nj})(t_j - n_{Nj})$$

- * For each hidden layer $i \in \{2, \dots, N - 1\}$, starting with $N - 1$, the one closest to the output layer, calculate the errors $\vec{\delta}_i$. That is, for each node j in layer i , we have:

$$\delta_{ij} = n_{ij}(1 - n_{ij})(\vec{\delta}_{i+1} \cdot \vec{w}'_{ij})$$

⁴One iteration of this loop is known as an *epoch*.

Where the weight vector \vec{w}'_{ij} holds the *outgoing* weights, that is, the weights on connections *from* node j in layer i , and *to* the nodes in the next layer $i + 1$.

- * The updating of weights is done in no particular order:

$$\vec{w}_{ij} \leftarrow \vec{w}'_{ij} + \eta \delta_{ij} \vec{n}_{i-1}$$

Where $\eta \in \mathbb{R}$ is the so-called *learning-rate* and designates the rate with which the (local) optimum is approached, and a typical value is $\eta = 0.05$. Updating of the bias values is similar, but simpler:

$$\vec{b}_i \leftarrow \vec{b}_i + \eta \vec{\delta}_i$$

3.3.1 Enumerated-Node Notation

A commonly used notational form for updating the network weights, enumerates the nodes in the network, so they are denoted by a single number h with their corresponding node-values being o_h . The above BP formulae then become:

- The error for all nodes k in the output layer, is given by:

$$\delta_k = o_k(1 - o_k)(t_k - o_k)$$

- For each hidden layer, again starting with the one closest to the output layer, we have the error for the nodes h in a given layer:

$$\delta_h = o_h(1 - o_h) \sum_{k \in H} w''_{hk} \delta_k$$

Where H is the set of nodes in the next layer that receive the value from the current node h . And w''_{hk} is the weight for inputting o_h into node k .⁵

- Finally, every network weight w''_{ij} is updated according to the following:

$$w''_{ij} \leftarrow w''_{ij} + \eta \delta_j o_i$$

And similarly for the bias value for each node h :

$$b_h \leftarrow b_h + \eta \delta_j$$

The derivation of these update rules is beyond the scope of this document, but are given in [10].

⁵The indices are sometimes exchanged in the litterature, but that would be inconsistent with the notation used in this document.

4 Swarm Optimization

In *Swarm Optimization* (SO) a number of *agents* seek to minimize or maximize a so-called *fitness* function. This is done by having each of the agents move around in the space of possible solutions directed by their changing fitnesses. That is, the agents have no explicit knowledge of say, the gradient of the fitness function, and are thus unable to perform *gradient descent* in the explicit way of e.g. the BP algorithm above.

Nevertheless, by simply communicating their better positions amongst each other, agents are still able to find optimal solutions in some cases.

For this kind of NN, the fitness function to be minimized is $E(T)$ of Eq.(1), and the search-space in which the agents move, is a subset of \mathbb{R}^M , where M is the total number of weights and bias-values:

$$M = \sum_{i=2}^N N_i(N_{i-1} + 1)$$

In this section, let \vec{x} denote a position in this search-space, instead of input to the NN.

4.1 Particle Swarm Optimization

A simple form of SO is known as *Particle Swarm Optimization* (bPSO), and is given in [2]. It works by attracting the agents (of which there are typically 20) to their own and neighbour's⁶ previous best positions. The degree of attraction is randomly chosen, and it is possible for the agents (or *particles* as they are also known) to move past these previous best points also, so that surrounding regions are also explored.

The velocity \vec{v} of the particle is updated after each time-step t , with \vec{p} being the agent's own previous best position, and \vec{g} the best of its neighbour's previous positions:

$$\vec{v}[t + 1] = \omega \vec{v}[t] + \phi_1 r_1 (\vec{p}[t] - \vec{x}[t]) + \phi_2 r_2 (\vec{g}[t] - \vec{x}[t]) \quad (5)$$

where $r_1, r_2 \in [0, 1]$ are uniform deviate random numbers, and $\phi_1, \phi_2, \omega \in \mathbb{R}$, with ϕ_1 and ϕ_2 both typically equal 2, and the so-called *inertia* ω , decreasing linearly from 0.9 to 0.4 throughout a run. Adding the velocity to the old position of the agent, results in its new position:

$$\vec{x}[t + 1] = \vec{x}[t] + \vec{v}[t + 1]$$

Both velocity and position are bounded so as to limit speed of movement and possible locations within the search-space.

⁶A neighbourhood is a fixed subset of the swarm of agents, and not the agents that are closest in regards to e.g. Euclidian distance.

4.2 Convulsive PSO

A variant of the bPSO is presented in [3], that primarily introduces two modifications to overcome the problem of premature convergence: *Swarm convulsions* that re-initialize the agents' positions in the search-space after a fixed number of steps (e.g. 4e4), and the ω factor decreases exponentially from 1 to 0.1 over these swarm-steps.

The second modification is *particle convulsions*, which is the introduction of another random factor into Eq.(5) and the use of the entire swarm's previous best position \vec{gbest} instead of just the neighbours' \vec{g} , that is:

$$\vec{v}[t + 1] = \omega r_\omega \vec{v}[t] + \phi_1 r_1 (\vec{p}[t] - \vec{x}[t]) + \phi_2 r_2 (\vec{gbest}[t] - \vec{x}[t]) \quad (6)$$

where $r_\omega \in \pm[\epsilon, 1]$ is also chosen at random for each use, e.g. with $\epsilon = 0.5$. Everything else remains the same as in the bPSO scheme above.

In the *Adaptive ω -range* cPSO (acPSO) variant as described in [4], the range of the ω sweep is adjusted according to improvement of fitness.

4.3 Stud GA

A variant of the GA optimization paradigm that is briefly described in section 3.1, is found in [6]. There, the so-called *Stud* GA (StudGA) always chooses the population's best solution so far, \vec{gbest} , as one of the chromosomes for mating, and the other is chosen by *Strict Tournament Selection*, which considers two of the chromosomes and selects the one having best fitness.

The kind of crossover operation being used, performs a random linear inter- and extra-polation between the two parental positions in the search-space. Mutation of the dimensions of a position is done according to a probability. If a dimension of a given agent is chosen for mutation, it has a random value added, which is a fraction of the dynamic range of that dimension.

This optimization scheme performs well on some benchmark problems and poorly on others. One problem is again premature convergence, which is remedied in [8] by adding *Stochastic Dispersing* of the population - i.e. simply re-initialize the positions of the agents according to some probability.

4.4 Compound Swarm Optimization

Different optimization schemes outperform eachother on different problems. Therefore it is desirable to combine them in a way where they benefit from the capabilities of each other.

There are numerous ways to combine optimization schemes, but [7] advocates the use of simple frameworks, dubbed *Compound Swarm Optimization* (CSO).

One particularly simple form of CSO, works by defining the *compound* swarm C out of n so-called *component* swarms S_i :

$$C = (S_1, \dots, S_n)$$

The agents of a swarm S_i are then executed according to a probability p_i , and the best previous position for that swarm is denoted \vec{gbest}_i . The compound best \vec{cbest} , is then defined as follows, for minimization problems:

$$\vec{cbest} = \underset{\vec{gbest}_i}{\operatorname{argmin}}(gbest_i) \quad (7)$$

Which may then be used instead of \vec{gbest} , for example in Eq.(6).

4.5 Back-Propagation Swarm

The experiments below show that SO schemes by themselves are not sufficient for optimizing the weights and bias-values of NNs in general.

The CSO framework may then be used to combine SO schemes with the BP algorithm. Two things must be noted though: The BP algorithm does not guarantee a better or equal $E(T)$ after having updated the network’s weights and bias-values. And, the only randomization of the BP algorithm, is in its initialization.

The obvious idea for incorporating the BP algorithm as part of a CSO scheme, would have been to create a BP-swarm with one agent, which performs the backpropagation directly on \vec{cbest} . This would require a guaranteed equal-or-better error-measure of the updated network, otherwise \vec{cbest} would be inconsistent with its definition in Eq.(7).

A method that is not only faster than BP, but also guarantees equal or better error-measures after each update of the network, is known as *Scaled Conjugate Gradient Descent* (SCG) and is given in [11]. It does however use intermediate values that could prove problematic when combining the method with SO schemes, and it is beyond the scope of this document to resolve those issues.

What is done instead, is simply to have the single agent of the BP-swarm to synchronize its position with \vec{cbest} according to a probability (e.g. $p_{\text{sync}} = 0.1$). This allows for temporarily worse paths to be followed without disrupting the other component swarms, meanwhile the BP algorithm may still benefit from improvements found by the other swarms, by jumping to their best position once in a while.

Potentially however, the agent of the BP swarm may restart in the same position many times, and thus perform the same calculations over and over. This could be alleviated by keeping track of whether \vec{cbest} was improved since last synchronization, but is ignored in the present work.

4.6 Pre-Emptive Fitness Evaluation

When the calculation of a fitness function is iterative and non-improving, certain SO schemes can take advantage of this and pre-emptively abort the fitness evaluation [9], so as to save computational time.

For NNs, both bPSO and acPSO can abort the evaluation of $E(T)$ when an agent discovers that its current position yields worse error than what it was, at

its previous best position. And indeed, when comparing test-set performance (see section 4.7), it is also possible to pre-emptively abort the evaluation of $E(\bar{T})$.

4.7 Training & Test Sets

It is customary to split the data-set D into a training-set T , and the test-set \bar{T} which is its complement. Using the BP algorithm of section 3.3, we seek to minimize the error over the training-set $E(T)$, but we actually end up using the network configuration that minimizes the error over the test-set, $E(\bar{T})$.

The reason for this is that the data samples may be noisy, and we do not wish to *overfit* the network to recognize these anomalies. Therefore T is the only data available in the modification of the network, and once the underlying structure is being recognized by the network, it is believed to perform best on the independent test-set also. Continuing further to train the network with T will decrease $E(T)$ but increase $E(\bar{T})$, so when selecting the network configuration that minimizes $E(\bar{T})$, it leaves us with a good compromise between accuracy and generalization.

This method for preventing overfitting, is reported in [12] to only work well with some algorithms however. Thus, comparing different algorithms in terms of their ability to find weights and bias-values that generalize the network well to unseen data, requires adequate overfitting prevention methods for each of the algorithms. In these experiments, this matter is ignored though, so the user may still decide how many data samples should be in the training-set T , and consequently in the test-set \bar{T} .

5 Problems

The algorithms are tested on a number of publicly available classification problems of varying complexity.

- **Recognizing Facial Direction:** The first classification problem uses a network with 960 inputs and 4 outputs. The input to the network is a 32 times 30 pixels gray-scale image, and the output classifies what direction the person on the image is looking, either left, straight, right, or up. There are a total of 624 of such images. A detailed explanation is given in [10], along with curious observations, for example that the weights increasingly resemble facial patterns during training.
- **Prechelt's Proben1:** This is a collection of problems, including 1-of- n classifications. For these experiments the *breast-cancer* and *diabetes* data-sets are used. Both of these are real observation-sets, where the first has measurements for tumours, such as size, cell uniformity, etc., along with their classifications as benign or malignant. The diabetes data-set is collected from female *Pima* indians, and records age, body-mass-index,

blood pressure, etc., and with a classification of either having diabetes or not.

These data-sets can be found at <http://www.daimi.au.dk/~u971055>, or at their original locations: Facial images at <http://www-2.cs.cmu.edu/afs/cs.cmu.edu/user/mitchell/ftp/faces.html>, and <ftp://ira.uka.de/pub/neuron/proben.tar.gz> for the Proben1 collection.

6 Implementation

Implementation is carried out in *MS Visual C++* and compiles to a *Windows*-executable, which is available at <http://www.daimi.au.dk/~u971055/> along with the source-code.

Functions are provided in `SO.h` for instantiating the various *SO* algorithms. These operate on the class `LVector` which represents a position in a search-space with corresponding fitness. Deriving from this, the class `LVectorNDim` implements a real-valued space, which is then further derived by `LVectorNN` which holds the NN parameters, i.e. weights and bias-values, that are *flattened* into the array-based data-structure of `LVectorNDim`.

The NN including the actual BP algorithm is implemented in `LNeuralNetwork`, that decodes the array of `LVectorNN` into usable parameters.

The user may input various parameters when running the program, including number of runs and iterations per run. The network parameters that minimize $E(\bar{T})$ are kept throughout these runs, by evaluating and comparing $E(\bar{T})$ after each update of the network parameters.

The examples in the training-set T are chosen at random from the entire data-set D . Diversity of the elements in T is required in order for the network to generalize well - imagine for example if T in the problem of determining facial direction, consisted only of people who looked straight-ahead, then how could the network be trained to recognize the other directions also. The randomly chosen training- and test-sets, are kept fixed throughout all of the runs.

6.1 Facial Direction Image Files

The facial images are stored in the *Portable Gray-Map* (PGM) file-format. As the NN takes floating points as its input, each gray pixel is first converted to a value in the range $[0, 1]$. The functions for reading and writing PGM files are provided in `YggImagePGM`.

A text-file containing a list of filenames for the PGM images, is supplied by the user. An example file is `faces.txt` which assumes the images are unpacked to the path `faces` relative to where the program is executed.

The classification of an image is created from finding substrings in the filenames, e.g. if the string *left* is found somewhere in the filename, then the image is assumed to portray a person looking to his/her left.

6.2 Proben1 Boolean & Real Valued Coding

Although the Proben1 data-sets distinguish between boolean- and real-coded data, there is no such distinction in this implementation. A boolean false is simply a value of 0, and a boolean true is then a 1. This way, the fitness used in optimizing network parameters for classification problems, can both be the MSE and the ratio of incorrect classifications - both of which are minimization problems - although the resolution of the latter depends on the size of the training-set $|T|$.

6.3 Output Scaling

The Sigmoid function is unable to output values of 0 and 1, but larger weights will cause the Sigmoid-function's output to approach these boundaries. Therefore the output t_d of the data-set, is sometimes scaled to the range $[0.1; 0.9]$ to avoid this endless weight growth. In practice however, although the weights have a tendency to get bigger when the data-set output is unscaled, the classification rate does not suffer. So whether to do this, is also user-decidable when running the test-program.

6.4 Determining Classification

The classification of network output, is given by the index of the highest-valued element of the output. Thus, an output of say $[0.49, 0.51]$ is classified the same as $[0, 1]$, even though the first has a high *uncertainty* in its classification. Similarly, it is relevant in many problems how high the rates of incorrect classifications are in terms of *false positives* and *false negatives*. This is the case in the classification of tumours, where a diagnosis as benign when the tumour is really malignant, is worse than than the other way around. These issues are presently ignored.

7 Experimental Results

Parameters for the SO algorithms are as described above. For the compound swarms, the probability of executing a step of each of the component swarms is $p = 1$, except for the StudGA which is $p = \frac{1}{2}$. There are 20 agents for the bPSO and acPSO swarms, 5 for StudGA, and a single *agent* for the BP-swarm. For all problems, the network had a single hidden layer with 3 nodes.

Although an epoch of the BP algorithm requires significantly more computation than an agent-step in one of the SO schemes, there is made no distinction between the two. Therefore, each algorithm is tested on each problem for 50 runs, each having 4000 agent-steps: That is, an epoch for the agent of the BP-swarm, and the updating of position and recalculation of $E(T)$ for the SO algorithms.

The output $\vec{t}_d \in \mathbb{R}^m$ was scaled to the range $[0.1; 0.9]^m$, and the fitness to be minimized was the MSE and not the number of incorrect classifications. The range of the weights and bias-values was ± 100 , and the velocity boundary for

the particles in the PSO schemes, and the maximum range of mutation for the StudGA, was $\frac{1}{20}$ th of the entire dynamic range. All weights and bias-values were initialized in the $[-0.1; 0.1]$ range.

The size of the training-set was 260 for all problems, which means that there were 508 data-samples in the test-set for the diabetes-problem, 439 for the cancer-problem, and 364 for the facial-direction problem. All data-sets are assumed to be valid with correct classifications; even though the NN is robust to incorrect examples, it will affect the measures.

The results are displayed in table 1. The first thing to notice is the poor performance of bPSO, acPSO, and StudGA on the facial-direction problem, but the introduction of CSO combining the BP-algorithm with any one of these yields significant time-reductions compared to BP, but also improved classification rates over the test-set, for the combination of BP and bPSO, or BP and StudGA.

The reason that the CSO consisting of BP and StudGA spends more time than e.g. BP and bPSO, is because the StudGA swarm has only 5 agents and a probability of 0.5 for execution, and thus executes many epochs of the BP-swarm relative to agent-steps of the StudGA-swarm.

The small standard deviations in the MSE for the training-set, indicate that all of the algorithms perform consistently in regards to the results they obtain in each of their runs. And although the different CSO combinations exhibit different strengths and weaknesses - e.g. less time-usage but slightly worse classification rate - it is still evident that the introduction of the CSO technique improves the results as a whole.

8 Conclusion

Compound Swarm Optimization (CSO) was shown to be useful for finding weights and bias-values in Neural Networks. They were found in a fraction of the computational time compared to Back-Propagation, and yielded comparable and sometimes even better performance of the resulting network, in terms of the number of correct classifications.

References

- [1] Genetic Generation of Both The Weights and Architecture For A Neural Network
John R. Koza, James P. Rice
<http://citeseer.nj.nec.com/>
- [2] Evolving Artificial Neural Networks
Russell C. Eberhart, Yuhui Shi
Indiana University Purdue University Indianapolis, 1998
eberhart,shi@engr.iupui.edu

	Minimal Training-Set MSE			Minimal Test-Set MSE								
	Algorithm	Time	Avg MSE (Stddev)	Training-Set		Test-Set						
				MSE	Class.	MSE	Class.					
faces.txt	BP	1	9.84e-4 (1.64e-3)	1.19e-4	100%	5.66e-2	89.84%	2.19e-4	100%	5.05e-2	90.38%	
	bPSO	0.36	0.24 (6.26e-3)	0.22	42.69%	0.22	41.48%	0.23	45%	0.22	46.43%	
	acPSO	0.33	0.23 (7.63e-3)	0.21	52.69%	0.21	46.98%	0.21	55%	0.21	46.15%	
	StudGA	0.63	0.24 (7.88e-3)	0.22	45.77%	0.24	31.59%	0.23	29.23%	0.23	37.64%	
	BP, bPSO	0.25	8.10e-3 (3.83e-3)	3.33e-3	100%	5.20e-2	91.21%	7.56e-3	99.23%	4.36e-2	92.86%	
	BP, acPSO	0.22	9.12e-3 (2.23e-3)	4.16e-3	100%	8.04e-2	84.62%	9.19e-3	99.23%	6.16e-2	88.74%	
	BP, StudGA	0.64	1.81e-3 (2.07e-3)	1.52e-4	100%	5.42e-2	89.56%	3.08e-3	99.62%	4.59e-2	90.93%	
	All	0.26	3.30e-2 (1.91e-2)	8.88e-3	99.62%	7.55e-2	85.99%	2.72e-2	98.08%	6.92e-2	88.19%	
	cancer1.dt	BP	1	1.19e-2 (4.68e-4)	1.12e-2	98.46%	1.84e-2	96.81%	1.64e-2	97.31%	1.55e-2	97.27%
		bPSO	0.28	1.68e-2 (2.35e-3)	1.20e-2	97.69%	2.11e-2	96.36%	1.27e-2	98.08%	1.88e-2	97.04%
acPSO		0.23	1.18e-2 (3.39e-3)	6.88e-3	99.62%	3.84e-2	94.53%	1.57e-2	96.92%	1.95e-2	97.04%	
StudGA		0.45	7.22e-2 (2.81e-2)	1.51e-2	98.08%	2.62e-2	94.53%	1.55e-2	98.08%	2.42e-2	94.99%	
BP, bPSO		0.30	1.94e-2 (3.45e-3)	1.52e-2	96.92%	1.47e-2	97.27%	1.66e-2	97.69%	1.40e-2	97.04%	
BP, acPSO		0.20	1.72e-2 (2.92e-3)	1.27e-2	98.08%	2.33e-2	94.99%	1.68e-2	96.92%	1.46e-2	97.49%	
BP, StudGA		0.71	1.08e-2 (4.98e-4)	9.18e-3	98.85%	2.18e-2	95.22%	1.32e-2	98.08%	1.61e-2	97.04%	
All		0.25	1.52e-2 (2.38e-3)	1.12e-2	98.08%	2.83e-2	94.53%	1.41e-2	97.31%	1.64e-2	97.27%	
diabetes1.dt		BP	1	8.46e-2 (1.61e-3)	8.29e-2	82.31%	1.31e-1	70.67%	9.51e-2	78.08%	9.91e-2	77.95%
		bPSO	0.43	1.22e-1 (8.02e-3)	1.03e-1	77.69%	1.07e-1	75.79%	1.06e-1	75.38%	1.05e-1	75.59%
	acPSO	0.38	1.02e-1 (1.05e-2)	8.73e-2	81.54%	1.13e-1	74.61%	9.12e-2	79.62%	1.08e-1	75%	
	StudGA	0.65	1.35e-1 (9.37e-3)	1.08e-1	77.31%	1.14e-1	72.64%	1.12e-1	76.15%	1.10e-1	75.20%	
	BP, bPSO	0.39	1.08e-1 (1.52e-3)	1.03e-1	76.54%	9.79e-2	77.76%	1.08e-1	75.77%	9.75e-2	77.76%	
	BP, acPSO	0.35	8.87e-2 (2.67e-3)	8.17e-2	80.77%	1.04e-1	76.38%	8.76e-2	80.38%	1.01e-1	75.98%	
	BP, StudGA	0.8	8.80e-2 (5.69e-4)	8.63e-2	78.46%	1.03e-1	77.36%	8.88e-2	79.23%	1.01e-1	78.35%	
	All	0.38	9.72e-2 (4.73e-3)	9.15e-2	79.23%	1.07e-1	74.41%	9.85e-2	79.23%	1.02e-1	76.57%	

Table 1: For each of the problems, a row displays the results for 50 runs of the given algorithm performing 4000 agent-steps. First the time-usage relative to Back-Propagation, then the average MSE over the training-set and its standard deviation. The solution that minimized fitness over the training-set has its MSE and classification rate displayed for both the training- and test-sets, and following this is the same figures for the solution that minimized fitness over the test-set. Compare algorithms by their time-usage and the right-most classification rates. The algorithm *All* designates a compound swarm consisting of BP, bPSO, acPSO, and StudGA.

- [3] Convulsive Particle Swarm Optimization
Magnus Pedersen (971055)
Daimi, University of Aarhus, August 2003
<http://www.daimi.au.dk/~u971055/>
- [4] Convulsive Particle Swarm Optimization Addendum
Magnus Pedersen (971055)
Daimi, University of Aarhus, September 2003
<http://www.daimi.au.dk/~u971055/>
- [5] Genetic Algorithms for Rule Discovery in Data Mining
Magnus Pedersen (971055)
Daimi, University of Aarhus, October 2003
<http://www.daimi.au.dk/~u971055/>
- [6] Stud Genetic Algorithm
Magnus Pedersen (971055)
Daimi, University of Aarhus, December 2003
<http://www.daimi.au.dk/~u971055/>
- [7] Compound Swarm Optimization
Magnus Pedersen (971055)
Daimi, University of Aarhus, March 2004
<http://www.daimi.au.dk/~u971055/>
- [8] Stochastic Dispersing of Agents
in Multi-Agent Function Optimization
Magnus Pedersen (971055)
Daimi, University of Aarhus, December 2003
<http://www.daimi.au.dk/~u971055/>
- [9] Pre-Emptive Fitness Evaluation in
Randomized Function Optimization
Magnus Pedersen (971055)
Daimi, University of Aarhus, February 2004
- [10] Machine Learning
Tom M. Mitchell
McGraw-Hill 1997
ISBN 0-07-042807-7
- [11] A Scaled Conjugate Gradient Algorithm
for Fast Supervised Learning
Martin Fodslette Møller, University of Aarhus
Neural Networks, Vol. 6, pp. 525-533, 1993
- [12] An Analysis of Particle Swarm Optimizers
Frans van den Bergh
Ph.D. Thesis, University of Pretoria, Nov 2001
<http://www.cs.up.ac.za/cs/fvdbergh/publications.php>