A Case Study in Neural Network Training with the Breeder Genetic Algorithm

Lluís A. Belanche
Secció d'Intel·ligència Artificial.
Dept. de Llenguatges i Sistemes Informàtics.
Universitat Politècnica de Catalunya.
c/Jordi Girona Salgado 1-3
08034 Barcelona, Spain.
belanche@lsi.upc.es

Abstract

Supervised training from examples of a feed-forward neural network is a classical problem, traditionally tackled by derivative-based methods (DBM) that compute the gradient of the error, such as backpropagation. Conventional methods for non-linear optimization, such as Levenberg-Marquardt, quasi-Newton and conjugate gradient are generally faster and more reliable, provided the objective function has continuous second derivatives. Their main drawbacks are well-known, being one of the more serious the possibility of getting caught in local minima of the error surface.

As an alternative, Evolutionary Algorithms (EA) have demonstrated their ability to solve optimization tasks in a wide range of applications. However, their use in the neural network context has concentrated on binary-coded Genetic Algorithms, often without a systematic approach and thus more than often outperformed by even simple DBMs. The result is that EAs are generally considered basically inadequate for this problem.

In this paper the possibilities of the Breeder Genetic Algorithm (BGA) to solve the numerical optimization problem are thoroughly explored. A case study is developed and used to tune the BGA for this kind of task, by searching in the space of genetic operators and their parameters, on the one hand, and as a function of selection pressure and population size, on the other. It is found that specific configurations stand out over the rest, in a way that is consistent with previous findings. The importance of precising the relationship between the last two mentioned quantities is also highlighted.

In order to assess to some extent the validity of the algorithm, a further batch of experiments is devoted to compare the BGA to a powerful DBM, a global method consisting of conjugate gradient embedded into a simulated annealing schedule. The results show a comparable performance pointing this evolutionary algorithm as a feasible alternative to derivative-based methods.

1 Introduction

Artificial Neural Networks [5, 10, 11] have been widely and most fruitfully used in a variety of applications during the last fifteen years, especially after the boosting seminal works of [12, 13, 21]. The main problem tackled in what is known as *supervised learning* is that of regression, the approximation of an (unknown) n-dimensional function $f(\vec{x})$ by a parameterised function $g_{\vec{w}}(\vec{x})$ of controllable form, assuming $f : X \subset \mathbb{R}^n \to \mathbb{R}$, with X a compact subset of $\mathbb{R}^n$.

The function $g_{\vec{w}}(\vec{x})$ is the one realized by the neural network. Usually one is interested in finding parameters $\vec{w}^*$ such that $g_{\vec{w}^*}(\vec{x})$ optimizes a cost functional $\mathcal{L}(f, g)$ called *loss*. Typically, the only information available takes the form of a set D of p fixed samples of f, $D = \{< \vec{x}^i, y^i >\}, \vec{x}^i \in \mathbb{R}^n, y^i \in \mathbb{R}$ and $|D| = p$, where $f(\vec{x}^i) = y^i$.

An initial estimation of $\mathcal{L}$ can be hence computed as $\mathcal{L}(f, g_{\vec{w}}) = \tilde{\mathcal{L}}(D, g_{\vec{w}})$. This cost measure is called *apparent loss*; it is computed separately for each sample in D,

$$\tilde{\mathcal{L}}(D, g_{\vec{w}}) = \sum_{(\vec{x}^i, y^i) \in D} l(< y^i, g_{\vec{w}}(\vec{x}^i) >) \tag{1}$$

where l takes often the form of an error function, for example the squared-error $l_{\mathrm{SE}}(< a, b >) = (a - b)^2$. The above expression can then be viewed as the (squared) Euclidean norm in $\mathbb{R}^p$ of the p-dimensional error vector $\vec{e} = (e_1, \ldots, e_p)$ with $e_i = y^i - g_{\vec{w}}(\vec{x}^i)$. The usually reported quantity $\sqrt{\frac{1}{p}||\vec{e}||}$ is called root mean square error (RMSE), and is a measure of the empirical error (as opposed to the unknown *true* error).

In this work we restrict the analysis to *feed-forward* artificial neural networks of one hidden layer of h_0 units and as many input and output units as needed by the problem (numbers n, m of input and output dimensions, respectively). We will refer to such architecture as a $n - h_0 - m$ network.

For real functions ($m = 1$), total network parameters $\vec{w}$ are split into the input-to-hidden weight vectors $\vec{w}^h$ (including the bias terms w_0^h) and the hidden-to-output weight scalars w_h (plus bias w_0), $1 \leq h_0 \leq h$. Among the most popular neuron models, the combination of scalar product plus a sigmoidal with the mentioned connectivity gives rise to the MultiLayer Perceptron (MLP). Denoting by σ_h and σ the activation functions used in the hidden units and output unit, respectively, the overall computation is expressed:

$$g_{\vec{w}}(\vec{x}) = \sigma\Big(\sum_{h=1}^{h_0} w_h \sigma_h(\vec{w}^h \cdot \vec{x} + w_0^h) + w_0\Big) \tag{2}$$

where $(\cdot)$ denotes scalar product. A particular case of the general regression problem arises when f is the *characteristic function* Ψ_C of some set $C \subset X$, with X a referential or universal set from which the points $\vec{x}$ are drawn. This can be thought of as a *classification* problem. The standard form for Ψ_C can be defined as:

$$\Psi_C(\vec{x}) = \begin{cases} 1 & \text{if } \vec{x} \in C \\ 0 & \text{if } \vec{x} \notin C \end{cases} \tag{3}$$

And thus the sample set D is formed out of pairs $< \vec{x}^i, y^i >$ with $y^i \in \{0, 1\}$ and $y^i = \Psi_C(\vec{x}^i)$, denoting the crisp class membership. The generalization to c classes $C_1, \ldots, C_c$ is straightforward, leading to pairs $< \vec{x}^i, \vec{y}^i >$, with $\vec{y}^i \in \{0, 1\}^c$. Class membership is expressed in a 1-out-of-c format as follows:

$$y_k^i = \begin{cases} 1 & \text{if } \vec{x}^i \in C_k \\ 0 & \text{if } \vec{x}^i \notin C_k \end{cases} \tag{4}$$

For this particular setting, other specific cost functions have been devised. The most prominent is the *cross-entropy*, obtained by minimizing the negative logarithm of the likelihood. Define, for $a, b \in [0, 1]$ and $\vec{a}, \vec{b} \in [0, 1]^c$:

$$l_{CE}(< a, b >) = -\{a \log b + (1 - a) \log(1 - b)\}, \tag{5}$$

and

$$l_{CE}^c(< \vec{a}, \vec{b} >) = -\sum_{k=1}^{c} a_k \log b_k, \ c \geq 2 \tag{6}$$

where $\log x$ is the natural logarithm. For $c = 2$, there are two possibilities: using the coding in (3) and (5) as the error for input pattern $\vec{x}^i$, or the coding in (4) and (6) as the error, with $C_1 = C, C_2 = X \setminus C$. For $c > 2$, the only possibility is the latter.

The extension of (1) and (2) to $m \geq 1$ outputs is straightforward. In any case, given $g_{\vec{w}}$ the task is to minimize (1) w.r.t. $\vec{w}$. Sometimes further restrictions are integrated in the cost, for example a smoothing term:

$$\lambda \sum_j w_j^2 \tag{7}$$

where $\vec{w}$ represents the overall vector of weights, with $\lambda \in \mathbb{R}, \lambda \geq 0$ a factor of proportionality. In all, two main requirements determine the goodness of a solution:

1. An acceptably balanced performance between data in D –that is, a low $\tilde{\mathcal{L}}(D, g_{\vec{w}})$– and *outside* D –a low $\tilde{\mathcal{L}}(D_{\text{out}}, g_{\vec{w}})$, to ensure that f has been correctly estimated from the data, important for generalization purposes. The set $D_{\text{out}} \subset X \setminus D$ of previously unseen data should not contain examples exceedingly different from those in D.

2. A minimal complexity, given by a small h_0 combined with a low connectivity.

The general problem of finding the most suitable solution for $g_{\vec{w}}$ can be decomposed in two steps:

(a) Select the particular parameterized form of $g_{\vec{w}}$. This involves determining a suitable representation for the inputs (and hence its number), the number of hidden layers and units per layer, the type(s) of neuron model(s) used in the different layers and the type of connectivity. With the input representation issue solved, it remains as a *combinatorial* optimization problem to determine the family of functions g belongs to. Cast in a neural network context, we refer to it as the network *architecture*.

(b) Once the form of g is fixed, the task is to determine a $\vec{w}^*$ such that the cost functional chosen is minimized. This is a *continuous* optimization problem, a search for the best instance within the previously selected family of functions.

To tackle the first task, model selection techniques such as cross-validation or prediction risk estimators [17] are of help. A common and simple procedure is the following: determine the inputs and outputs by the knowledge on the problem (usually resorting to some coding scheme); use a fixed neuron model (for example, scalar product plus sigmoidal) and a fixed connectivity (usually full between adjacent layers, and no shortcuts); reduce the problem of hidden unit determination to a (possibly incremental) search over a predetermined small space of combinations, employing a model selection technique, to achieve a good compromise between bias and variance [14]. Even focusing only in this task, it is a difficult one that can become a painful design problem for certain applications.

To tackle the second task –the real optimization problem– several strategies have been proposed. By far the most widely used in the neural network context are those relying in derivative information, herein called derivative-based methods (DBMs) of which the most popular is undoubtedly Backpropagation [21]. Other, faster and more powerful schemes exist such as Conjugate Gradient Descent [15, 23] or the Levenberg-Marquardt method [5]. Typical problems are well known and –besides determination of the learning parameters– include [11]:

1. Long training times, oscillations and network paralysis, on the one hand;

2. The (high) probability of getting stuck in local optima, on the other. In this last situation, the addition of a global scheme (like multiple restarts or an annealing schedule) is surely to increase the chance of finding a better solution, although the cost can become prohibitively high.

2 Evolutionary Algorithms

The alternative to these methods are Evolutionary Algorithms (EAs) [2], of which only Genetic Algorithms (GAs) [9] have been extensively used for this task. One of their main advantages over methods based on derivatives is the global search mechanism. A global method does not imply that the solution is not a local optimum; rather, it eliminates the possibility of getting *caught* in local optima. Another appealing issue is the possibility of performing steps (a) and (b) at the same time, in a search over the joint space of structures and weights (for a review, see [22, 28]). Still another possibility is to embody a DBM into a GA, using the latter to search among the space of structures (a) and the DBM to optimize the weights (b) [16]; this hybridization leads to extremely high computational costs. Finally, there is the use of EAs solely for the numerical optimization problem.

It is our conviction that, in the neural context, this is the task for which continuous EAs are most naturally suited. It is difficult to find applications in which GAs (or other EAs, for that matter) have outperformed DBMs for supervised training of feed-forward neural networks [25]. It has been pointed out [20] that this task is inherently hard for algorithms that rely heavily on the recombination of potential solutions. In addition, the training times can become too costly, even worse than that for DBMs.

The emergence of evolutionary techniques using different selection schemes and a direct representation of variables allows to bring new light on evolutionary approaches to supervised neural network training.

Among the latest algorithms we find the Breeder Genetic Algorithm (BGA) [18], which is characterised by a truncation selection procedure and direct representation of continuous variables. Truncation selection is a simple use of rank-based selection, proven to be very useful in traditional GAs [27]. The direct representation of variables eliminates the need for a coding scheme –that usually changes the search space– and permits to develop new continuous genetic operators. The BGA has already been used for some specific neural optimization tasks or application examples:

- In [6] the hybrid methodology –the BGA for step (a) combined with a DBM (BackPropagation) for step (b)– is used in an application example concerned with a non-linear system identification task. Also, the BGA is replaced by a GA and the results compared. The combination (BGA, DBM) is found to be superior to the (GA, DBM).

- In [7] the numerical optimization problem is left to the BGA, given a fixed model, to solve the Mackey-Glass time series. The results are compared to those obtained by a DBM (a globally enhanced BackPropagation). The DBM is found superior to the BGA and this, in turn, superior to the GA.

- In [29], a BGA variant called BGP or Breeder Genetic Programming is introduced. It is basically a BGA with variable-length chromosomes and a next-ascent hill-climbing mutation mechanism to improve on the individuals found. All the work is left to the BGP and the search is biased to solutions representing minimal networks. The two tasks addressed are binary problems and the focus is on the ability of the system to find the minimal known solutions.

In all, the results point to the BGA found to be markedly superior to traditional GA techniques although inferior to DBMs, especially if the latter are enhanced by a means to escape from local optima. However, these are initial studies and the application of the BGA to neural network training does not constitute yet a widely spread methodology. It is our belief that EAs –particularly, the continuous ones– are in need of specific research devoted to ascertain their general validity as alternatives to DBMs in neural network optimization. Theoretical as well as practical work oriented to tailor specific EA parameters for this task, together with a specialized operator design should pave the way to a fruitful assessment of validity, both in terms of efficacy and efficiency.

As an initial step toward this end, in this work the BGA is used for the numerical optimization task of a specific classification problem, the Pima Indians database, often used for neural benchmarking [19]. The intention is not to solve this problem to a full, or achieve the best possible solution, but rather to use it to investigate the possibilities of the BGA in the place of a DBM, in a task known not to be an easy one [30]. Specifically, this work is to address the following issues:

1. Determine whether and how can the BGA –as a particular EA technique– cope with this problem;

2. Explore the way different configurations of the algorithm (choice of genetic operators and their parameters) affect its performance, investigating the existence of configurations potentially better suited for neural network training;

3. With the experience gained, select the best of those settings and rerun the BGA with the purpose of finding good solutions;

4. In case they are found to be within reasonable values, to compare them to solutions found by DBMs.

The remainder of the paper is structured as follows. In the next Section, the basics of a BGA are surveyed, with emphasis in the available repertoire of genetic operators. Next, in Section 5 the experimental design is outlined, including a description of the task at hand, some comments on the use of a BGA to train a neural network and the particulars about the methodology to be followed. The results are presented in Section 6. In Section 7 a performance comparison to a DBM is carried out. The paper ends with a summary of findings and the Conclusions and prospects for future work.

3 The Breeder Genetic Algorithm

The Breeder Genetic Algorithm [18] is in mid position between Genetic Algorithms (GAs) [9] and Evolution Strategies (ES) [3]. While in GAs selection is stochastic and meant to mimic —to some degree— Darwinian evolution, BGA selection is named *truncation* selection, a deterministic procedure driven by a *breeding* mechanism, an artificial selection method in which only the best individuals —usually a fixed percentage τ of total population size μ— are selected

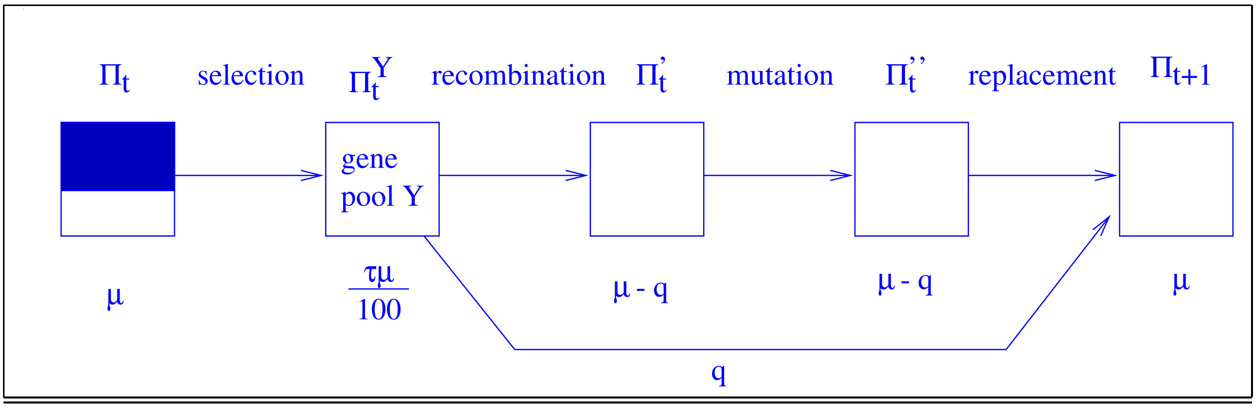

Figure 1: A scheme of the BGA procedure. Each box represents the population at different stages in the process to form a new generation. Notation on top of the boxes names the population at that point and the label from box to box (above the arrows) denotes operator sequencing (from left to right). The expressions at the bottom of the boxes indicate the population size at each step. Note how the final population size μ is formed by summing its two incoming values.

and enter the gene pool to be recombined and mutated, as the basis to form a new generation. Genetic (recombination and mutation) operators are applied by randomly and uniformly selecting two parents until the number of offspring equals $\mu - q$. Then, the former q best elements are re-inserted into the population, forming a new generation of μ individuals that replaces the previous one. This guaranteed survival of some of the best individuals is called $q - elitism$. For the BGA, the typical value is $q = 1$.

The BGA selection mechanism is then deterministic (there are no probabilities), extinctive (the best elements are guaranteed to be selected and the worst are guaranteed *not* to be selected) and 1-elitist (the best element is always to survive from generation to generation). Self-mating is always prohibited. The BGA procedure is depicted in Fig. 1, where τ is the truncation percentage for selection.

Unlike GAs, BGAs use a direct representation, that is, a gene *is* a decision variable (not a way of coding it) and its allele is the value of the variable. An immediate consequence is that, in the absence of other conditionings as constraint handling, the fitness function equals the function to be optimized. In addition, in a BGA chromosome there are no additional variables, i.e., the algorithm does not self-optimize any of its own parameters, as is done in ES and in some meta GAs. Chromosomes are thus potential solution vectors $\vec{x}$ of n components, where n is the problem size, the number of free variables of the function to be optimized. This issue is of crucial importance because:

1. It eliminates the need of chosing a coding function (e.g., binary, Gray);

2. It clears the way to the direct coding of different kinds of variables other than real numbers (e.g., fuzzy quantities, discrete quantities, etc).

The common aspect of BGAs with ordinary GAs is the fact that both are mainly driven by recombination, with mutation regarded as an important but background operator intending to reintroduce some of the alleles lost in the population. This view is conceptually right for GAs, because the cardinality of the alphabet used to code variables into the chromosome (the number of alleles per gene) is usually very small (two, in most cases). But in the case of algorithms that make use of real-valued alleles, like the BGA, mutation has to be seen in the double role of solution fine-tuner (for very small mutations) and as the main discovery force (for moderate ones). We will now briefly describe the different possibilities for the genetic operators. The reader is referred to [4] for a detailed description.

3.1 Recombination

Any operator Ω combining the genetic material of the parents is called a recombination operator. In BGAs, recombination is applied unconditionally, $\Pr(\Omega) = 1$. Let $\vec{x} = (x_1, \ldots, x_n)$, $\vec{y} = (y_1, \ldots, y_n)$ be two selected gene-pool individuals $\vec{x}, \vec{y}$ such that $\vec{x} \neq \vec{y}$. Let $\vec{z} = (z_1, \ldots, z_n)$ be the result of recombination and $1 \leq i \leq n$. The following are some of the more common possibilities to obtain an offspring $\vec{z}$:

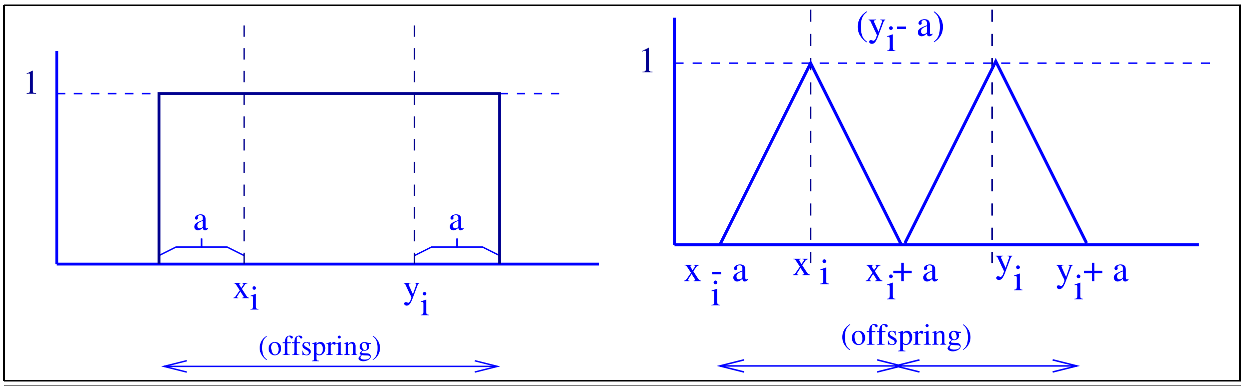

Figure 2: Potential zones for offspring and their probabilities. (Left) for the EIR operator, uniform *pdf* with $a = \delta|y_i - x_i|$. (Right) Bimodal *pdf* for the FR operator, where $a = e|y_i - x_i|$, for $0 < e < 1$ (shown for $e = 0.5$).

1. Discrete Recombination (DR).

   $z_i \in \{x_i, y_i\}$ (chosen with equal probability)

2. Line Recombination (LR).

   $$z_i = x_i + \alpha(y_i - x_i)$$

   with a fixed $\alpha \in [0,1]$. Typically, $\alpha = 0.5$.

3. Extended Intermediate Recombination (EIR).

   $$z_i = x_i + \alpha_i(y_i - x_i)$$

   with $\alpha_i \in [-\delta, 1+\delta]$ chosen with uniform probability. The δ parameter expresses to what degree an offspring can be generated out of the parents's scope, the imaginary line that joins them in $\mathbb{R}$. More precisely, it works by controlling the maximum fraction $a = \delta|y_i - x_i|$ of the distance between parents where the offspring can be placed, either left to the leftmost parent or right to the rightmost parent (Fig. 2, left). A typical value for $\delta = 0.25$, although any non-negative real number is a potential value. Reasonable values should not exceed $\delta = 0.5$, since the bigger the δ, the more the effect of the parents is diminished in creating offspring. A method for dynamically setting its value called $range_\delta$, was introduced in [4] and shown to have a remarkable effect in performance. It works as follows:

   $$z_i = y_i + \alpha_i(x_i - y_i), \qquad \text{with } x_i \geq y_i$$

   such that $\alpha_i \in [-\delta_i^-, 1+\delta_i^+]$ with uniform probability and,

   $$\delta_i^- = \frac{y_i - r_i^-}{r_i^+ - r_i^-}$$
   $$\delta_i^+ = \frac{r_i^+ - x_i}{r_i^+ - r_i^-}$$

   This procedure assigns different values for the left (δ_i^-) and right (δ_i^+) limits of the interval from which α_i is to be selected, and does never generate a value outside the range $[r_i^-, r_i^+]$ for the variable i, an aspect not fulfilled by the other methods that otherwise has to be dealt with a posteriori.

4. Fuzzy Recombination (FR). This operator, introduced in [24], basically replaces the uniform *pdf* (probability distribution function) by a bimodal one, where the two modes are located at x_i and y_i, the two parents, that is $\Pr(z_i) \in \{\Pr_{x_i}(z_i), \Pr_{y_i}(z_i)\}$ thus favouring offspring values close to them, and not in any intermediate point with equal probability, as with previous operators. The label "fuzzy" comes from the fact that the two parts $\Pr_{x_i}(t), \Pr_{y_i}(t)$ of the probability distribution resemble fuzzy numbers (triangular in the original formulation), such that they fulfill the general conditions (where $y_i \geq x_i$):

   $$x_i - e|y_i - x_i| \leq \ t \ \leq x_i + e|y_i - x_i|$$
   $$y_i - e|y_i - x_i| \leq \ t \ \leq y_i + e|y_i - x_i|$$

   stating that the offspring t lies in one (or both) of the intervals, being $e > 0$ the fuzzy number's spread, the same for both parts. The favour for offspring values near the parents is thus stronger the closer the parents are. This operator is depicted in Fig. 2 (right). In the simplest case, assuming $e = 0.5$ –that is, the two parts meet at the median and this point has zero probability, as in the figure– an offspring z_i is obtained with probability

   $$\Pr(z_i) = \mathrm{BT}(z_i)\{e|y_i - x_i|, x_i, e|y_i - x_i|, y_i\}$$

   where $\mathrm{BT}(t)$ is a Bimodal Triangular *pdf*, defined by the four notable points: the two modes and their left-right spread (these are equal in our case).

3.2 Mutation

A mutation operator Ψ is applied to each gene with some probability $\Pr(\Psi) = 1/n$ so that, on average, one gene is mutated for each individual. Let $\vec{z} = (z_1, \ldots, z_n)$ denote the result of mutation of an individual $\vec{x}$. The elements of $\vec{z}$ are formed as follows:

1. Discrete Mutation (DM).

$$z_i = x_i + sign \cdot range_i \cdot \delta$$

with $sign \in \{-1, +1\}$ chosen with equal probability, $range_i = \rho(r_i^+ - r_i^-)$, $\rho \in [0.1, 0.5]$ and

$$\delta = \sum_{i=0}^{k-1} \varphi_i 2^{-i}$$

where $\varphi_i \in \{0, 1\}$ from a Bernouilli probability distribution where $\Pr(\varphi_i = 1) = 1/k$. In this setting $k \in \mathbb{N}^+$ is a parameter originally related to the *precision* with which the optimum was to be located, a machine-dependent constant. Modern machines, capable of `double` precision, would in principle allow for higher values of k (e.g. 24, 32) than those traditionally used (e.g. 8, 16). In practice, however, the value of k is related to the *expected* value of mutation steps: the higher k is, the more fine-grained is the resultant mutation operator [4]. The factor ρ is the *range ratio*, related to the *maximum* step that mutation is allowed to produce as a ratio of variable range. This scheme favours small values but cannot generate all possible representable points[1], but only a discrete amount and prefers small values in an approximately (on average) logarithmic ($\log_2$) scale, always up to a precision of $range_i \cdot 2^{-k+1}$.

2. Continuous Mutation (CM). Same as DM but with

$$\delta = 2^{-k\beta}$$

where $\beta \in [0, 1]$ with uniform probability.

4 Experimental Design

The Pima Indians Database has been taken from the Proben Archive [19]. It consists of 768 examples of a two-class classification task, a positive or negative diagnose of diabetes. There are 8 input real variables, normalized to lie in the interval $[0, 1]$. Class membership comes coded as a 1-out-of-2 scheme. The number of training cases used is half this quantity –actually the first half, which happens to be class-split into 145 (37.76%) and 239 (62.24%), totaling $p = 384$ training cases. The number of hidden units chosen is fixed to $h_0 = 6$, that is, a neural architecture $8 - 6 - 2$ is the one to be used throughout all of the experiments performed. This means that a total of 68 free parameters are to be optimized. In general, this number is computed as $nf(n, h_0, m) = h_0(n + 1) + m(h_0 + 1)$. The activation function is the logistic for all of the hidden units (σ_h), and the identity for the output ones (σ). The cost function to be optimized (equal to the fitness) is the l_{SE} accumulated over all training examples. All the weights are let to vary within the (possibly too generous) real interval $[-10, 10]$.

Note at this point that the BGA (or any other algorithm for that matter) is likely to find different results for different numbers of hidden units, basically due to two compromising reasons:

1. A different form for g is going to approximate (or, better, to accommodate) D more easily the higher is h_0, in theory regardless of the training algorithm.

2. Given that n, m are fixed, an increasing number h_0 of hidden units means a linearly increasing number of free parameters (the length of $\vec{w}$) and any training algorithm is likely to have more trouble optimizing a function in a higher number of dimensions.

Provided we are not concerned in this work with selecting the best architecture but on assessing BGA performance for the numerical optimization task with varying parameters, h_0 is kept constant for all the experiments. An important hypothesis is then that should we select a different h_0, the overall results would have been different, but only in *absolute* terms. In our case, the value of $h_0 = 6$ has been chosen deliberately small –although not to the point of being too restrictive– so that a learning algorithm has a difficult time accommodating the rather large training set of cases. This way, differences in performance are to show themselves more markedly. In what concerns the BGA, we are primarily interested in addressing the following issues:

- Choice of mutation operator Ψ and its parameters ρ and k.

[1] By this we mean *machine*-representable. We assume that there is a machine-dependent floating point constant ϵ equal to the smallest positive representable number in a chosen precision. For example, in our machines, such number for `double` precision is $\epsilon \approx 2.22 \cdot 10^{-16}$.

- Choice of recombination operator Ω and its parameter δ (only for EIR).
- Determination of the truncation threshold τ.
- Study of performance as a function of the population size μ.

The stopping criterion is based on the number of fitness evaluations permitted (given by the variable `FFEvals`). In particular, given a finite number of `FFEvals`, the algorithm will stop each run whenever $\lfloor \frac{\texttt{FFEvals}}{\mu} \rfloor$ generations are reached. This stopping criterion allows to compare different general settings in a fair way, since, for example, a smaller population will be allotted more generations, but always keeping the number of fitness evaluations in similar values. For each configuration, a number of independent runs are performed –denoted by `NRuns`– keeping track of the mean and best solutions found. For all of the experiments, unless otherwise stated, `FFEvals`=$40,000$ and `NRuns`=20. Following the initial studies performed on some classical optimization problems [4] the following representative subset of possibilities is to be explored:

Mutation operator $\Psi \in \{\text{CM, DM}\}$; parameters $\rho \in \{0.1, 0.5\}$, $k \in \{8, 16, 24, 32\}$. Number of different configurations: 16.

Recombination operator $\Omega \in \{\text{DR, LR } (\alpha{=}0.5), \text{EIR } (\delta \in \{0, 0.15, 0.25, 0.35, 0.45\}), \text{EIR } (\delta = range_\delta), \text{FR } (\text{e}{=}0.5)\}$ and a simple random recombination. Number of different configurations: 10.

Truncation threshold $\tau \in \{5, 8, 11, \ldots, 50\}$. Number of different configurations: 16.

Population size $\mu \in \{2, 4, 6, \ldots, 100\}$. Number of different configurations: 50.

This experimental design is suboptimal because not all the possible combinations of mutation with recombination or with values of τ are tested. As we shall see, this number is too high to allow a full study to be performed and, furthermore, it is our belief that many of the combinations can readily be discarded *a priori* by a smaller but more effective experimental design. Also, even with the full results, the conclusions could not in any case be general ones since there is probably no configuration that is optimal for every conceivable (even reasonably) network optimization task, not even if, as in this work, the study is circumscribed to a specific search algorithm. Clearly, an exhaustive search over configurations is computationally infeasible, even limiting ourselves to a finite number of possibilities for every issue tackled (in our case, as in can be readily checked, this number amounts to 128,000). Hence, instead of performing an exhaustive (in a sense, multiplicative) number of experiments, a greedy (additive) strategy can be applied in the following way:

1. Set all parameters constant across all the experiments to their values.
2. Set an initially *neutral* setting for the issues to be explored, except for the first.
3. Select a suitable ordering for the issues to be explored;
4. Perform the experiment for the *first* of the issues according to the order chosen. Determine the *best* (or better two) setting, and hold it constant for the rest of the experiments, replacing the old value;
5. Perform the experiment for the *next* of the issues according to the order chosen. Determine the *best* (or better two) setting, and hold it constant for the rest of the experiments, replacing the old value;
6. If not done (all issues explored), go to step 5.

The strategy is then to perform several consecutive experiments that are kept simple and use the knowledge found up to the point for the remaining experiments. By proceeding this way, the number of configurations is to shrink to a minimum of 92, assuming that only the best setting is selected for each issue. If the better two are kept, this number will be between 92 and 184 (the double), depending on the ordering. In any case the number of configurations is now manageable. The order chosen is that of the previous description. First: selection of mutation operator; next: selection of recombination operator, truncation threshold and finally population size. There is no strong reason behind this ordering: it has been the case because we are primarily interested in finding out whether there are mutation and recombination settings generally better suited for neural network training, and *then*, to explore their performance across different truncation thresholds and population sizes.

5 Experimental Results

5.1 Experimental results on mutation

For this experiment, $\mu = 100$ and `NRuns`=10 for each setting of the algorithm shown in Fig. 3, where the "neutral" choice for recombination, Ω = EIR (δ = 0.25), fairly standard, has been selected. The results are presented as follows. Two tables are given, separated in continuous (CM, Table 1) and discrete (DM, Table 2) mutation. For each configuration (ρ, k) the *average* and *best* solution found throughout the `NRuns` are kept. Instead of giving a separate entry for each such configuration, some simple additional computations are performed to compact the information. The entries in the column for ρ are obtained averaging out the results **forall** k **in** {8, 16, 24, 32}. Similarly, the entries in the column for k are those obtained averaging out **forall** ρ **in** {0.1, 0.5}. By proceeding this way, one has to deal with less information and the obtained values are more representative. For instance, the entries for ρ are averages over 20 runs and those for k over 40 runs. Top value shown is the average and bottom value (in parentheses) is the best result.

```
Procedure Mutation-Test ()
{
  Ω := EIR (δ = 0.25);
  forall k in {8, 16, 24, 32}
    forall ρ in {0.1, 0.5}
      forall Ψ in {CM, DM}
        BGA (μ, k, ρ, Ψ, NRuns, FFEvals, Ω, τ);
}
```

Figure 3: `Mutation-Test` Algorithm pseudocode.

$-\rho-$		$-k-$			
0.1	0.5	8	16	24	32
130.0	**124.7**	**123.4**	**124.9**	131.4	129.8
(120.1)	**(115.7)**	**(114.9)**	**(116.8)**	(120.2)	(119.6)

Table 1: Results for Continuous Mutation (CM). See text for an explanation of entries. Markedly good results are boldfaced.

$-\rho-$		$-k-$			
0.1	0.5	8	16	24	32
130.2	**127.1**	**126.1**	128.6	131.1	128.6
(117.3)	(117.1)	(118.6)	(120.1)	**(115.3)**	**(114.9)**

Table 2: Results for Discrete Mutation (DM). See text for an explanation of entries. Markedly good results are boldfaced.

All figures correspond to the direct fitness values obtained, corresponding to the accumulated square error, to be minimized. This value is a monotonic function of the usually reported error and computationally cheaper. To get the actual root mean square error (RMSE), just divide each entry by 768 (number of patterns times number of outputs) and take the square root of the result. The results are of course non-conclusive because the number of runs is not too high and the explored possibilities are also a small – though representative– sample. However, by looking at Tables 1 and 2 several aspects are noteworthy.

- Firstly, the results for CM are very neatly defined: performance is superior for $\rho = 0.5$ over $\rho = 0.1$, and for $k \in \{8, 16\}$ over $k \in \{24, 32\}$, both on average and for the best value obtained. This corresponds to the boldfaced rectangular region in Table 1.
- For DM, the results are less vigorously defined, but in general accordance. It can be checked that again $\rho = 0.5$ is superior to $\rho = 0.1$ and performance for $k \in \{8, 16\}$ is *at least as good* as $k \in \{24, 32\}$. This last point, however, is not fulfilled by the absolute best values found; though these are less representative quantities than the averages, this behaviour is in need of an explanation.
- The overall results (high values of ρ combined with low values of k) are also in strong accordance with previous studies on non-trivial functions [4].
- Secondly, CM seems to be slightly but consistently superior to DM, a fact that shows itself especially for the best results, that is, for $\rho = 0.5$ and $k \in \{8, 16\}$. For $k \in \{24, 32\}$, CM is better but only marginally and in any case the results for these k are worse than those for $k \in \{8, 16\}$.

By looking at the average results for individual settings (not shown) we find that performance for CM (0.5, 8) is equal to 122.2, while that for CM (0.5, 16) is 121.5, and hence this last setting is the one finally selected, although we conclude that a value of $k = 8$ is an almost equally valid choice.

5.2 Experimental results on recombination

In the case of recombination operators, up to ten different settings are tested, as follows: DR, LR with $\alpha = 0.5$, EIR with $\delta = 0, 0.15, 0.25, 0.35$ and 0.45, EIR with $range_\delta$ and FR with $e = 0.5$. The mutation operator is fixed to CM with $\rho = 0.5, k = 16$, as a result of the previous experiment on mutation. The BGA procedure in the algorithm of Fig. 4 is run for this experiment with $\mu = 100$ and `NRuns`=20 for each setting. As an additional reference, a completely uninformed recombination operator is included, which simply yields

RR	DR	LR ($\alpha = 0.5$)	EIR ($\delta = 0$)	EIR ($\delta = 0.15$)	EIR ($\delta = 0.25$)	EIR ($\delta = 0.35$)	EIR ($\delta = 0.45$)	EIR ($range_\delta$)	FR ($e = 0.5$)
399.5 (193.6)	138.2 (114.4)	129.8 (119.5)	124.5 (114.0)	121.7 (112.1)	124.2 (115.8)	122.5 (111.2)	124.9 (110.6)	117.6 (114.8)	127.4 (112.8)

Table 3: Results for the different settings of recombination operators. Each entry shows the average and best results (in parentheses) across `NRuns`=20 runs.

an offspring generated randomly within the range (not taking into account the parents's information). This operator is denoted RR (Random Recombination).

Procedure `Recombination-Test` ()
{
  Ψ := CM ($\rho = 0.5$, $k = 16$);
  forall Ω **in** { RR, DR, LR ($\alpha = 0.5$),
    EIR ($\delta = 0, 0.15, 0.25, 0.35, 0.45$),
    EIR ($range_\delta$), FR ($e = 0.5$) }
      BGA (μ, F_i, k, ρ, Ψ, `NRuns`, `FFEvals`, Ω, τ);
}

Figure 4: `Recombination-Test` Algorithm pseudocode.

The results are presented in a single table for ease of reading –Table 3– summarizing the information for all of the recombination variations tested. For each configuration, the average and best (in parentheses) solutions found throughout the `NRuns` are shown. Several points are noteworthy:

1. The computed average performance of EIR across a fixed $\delta \in \{0.15, 0.25, 0.35, 0.45\}$ is equal to 123.6. This figure compares favourably to those for the other operators (RR, DR, LR and FR), not counting EIR ($range_\delta$).

2. FR is somewhat in between on average. Note that its best result (112.8) is very similar to that for the averaged EIR with fixed δ (112.7). It seems then that this last operator (EIR) can yield potentially better results, provided we find the right δ. In any case, averaged EIR mean behaviour (123.6) is better than FR (127.4).

3. With respect to EIR ($range_\delta$), it is clearly the best setting on average (117.6) though it achieves only middling results for the best value. A closer study on performance with a greater number of runs should give more definite results. In this line of thinking, it is interesting to observe that EIR ($\delta = 0.45$) has the absolute best result, although its mean performance is the worst across all EIR. All this confirms that the best values found are subject of much greater variability and cannot be readily predicted by the average results, this last quantity being the one to be primarily taken into account to judge among different settings.

To sum up, note that the average performance of EIR is *always* better than the rest (RR, DR, LR and FR), for all δ. This shows EIR as a robust operator, and notably when using the method $range_\delta$.

5.3 Experimental results about the truncation threshold

Sixteen different values for the truncation threshold $\tau = \{5, 8, 11, \ldots, 50\}$ are tested, as samples of the full (discrete) interval $[5, 50]$. As before, the mutation operator is fixed to CM with $\rho = 0.5, k = 16$. As a result of the previous experiment on recombination two settings, EIR ($\delta = 0.35$) and EIR ($range_\delta$), are to be used. The latter is the one with the best average performance, while the former –besides being a representative of EIR with a fixed δ– has the best all-round performance (crudely defined as the mean of average and max). The BGA procedure (algorithm of Fig. 5) is run for this experiment with $\mu = 100$ and `NRuns`=20 for each setting. The results are presented graphically in Fig. 6 for EIR ($\delta = 0.35$) (left) and EIR ($range_\delta$) (right). As usual, for each configuration, average and best performance throughout the `NRuns` are shown.

Procedure `Truncation-Test` ()
{
  Ψ := CM ($\rho = 0.5$, $k = 16$);
  forall Ω **in** {EIR ($\delta = 0.35$), EIR ($range_\delta$)}
    forall τ **in** $\{5, 8, 11, \ldots, 50\}$
      BGA (μ, F_i, k, ρ, Ψ, `NRuns`, `FFEvals`, Ω, τ);
}

Figure 5: `Truncation-Test` Algorithm pseudocode.

The most prominent observation is the radically different behaviour of both settings: whereas for EIR (δ=0.35) performance firmly increases up to a bending zone (about [26, 35], peaking[2] at 29 and 32) and then

[2] The peak points are orientative because not all points have been sampled. They should be read as "around" marks.

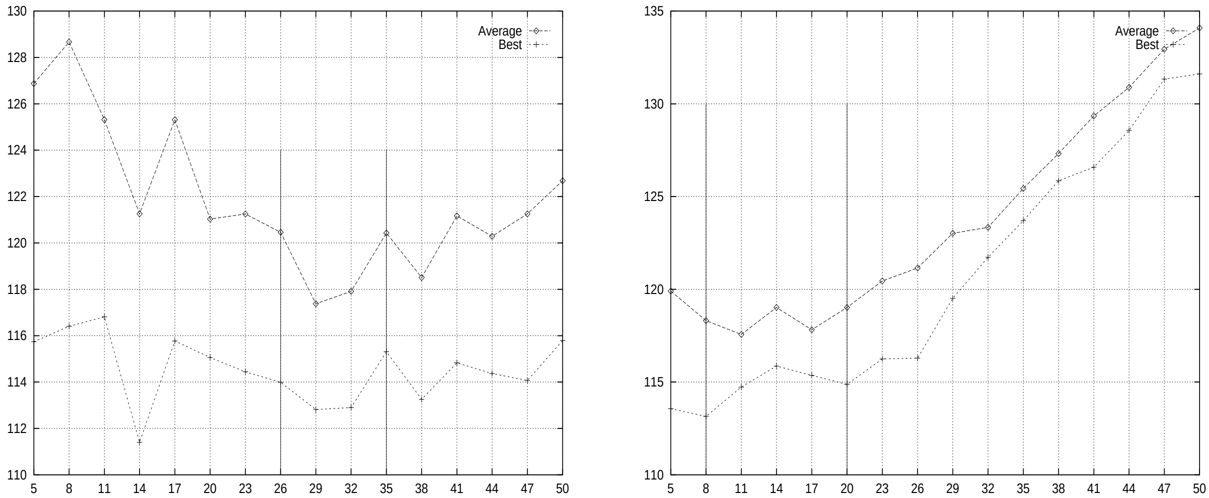

Figure 6: Results as a function of truncation parameter τ. Left: EIR ($\delta = 0.35$). Right: EIR ($range_\delta$). Each point is the result of `NRuns`=20 runs.

begins to decline, for EIR ($range_\delta$) the best region appears very soon, in [8, 20], peaking at 11 and 17. These zones are marked in the plots of Fig. 6 (left and right) by two vertical bars. In between (in [20, 26]) there is a transition zone, where performance for both is in the [118, 122] mark. More precisely, the crossing lies in [23, 26], where both operators are in the [120, 122] mark. This comparative behaviour is clearly seen in Fig. 7, where average results for the two operator settings are plotted together. It is also clear from this last plot that both configurations achieve comparable average results, although for quite opposite values of τ.

Our motivation here is clear: no single value is going to be always the best suited, because the particular threshold for truncation is likely to be tailored to a specific problem; instead, it makes more sense to ascertain a generally correct interval and then choose a value or values inside it. In our case, the two intervals are very definite but different for the two settings. For the sake of avoiding duplicate experiments or favour one of the two settings, the point of equal performance is selected: the midpoint of [23, 26] being 24.5, we take 25. This approximate cross-point value happens to be consistent with usual proposals (usually $\tau \in [20, 25]$) in the neural network context [7, 29].

5.4 Experimental results varying the population size

Once the best configurations for mutation and recombination and a reasonable value for the truncation threshold have been set, it is of interest to perform an experiment on how the population size μ affects performance. In this line of argument, it is very clear that no single value for μ is going to be generally "best", even for a particular task. Rather, we are concerned with studying the curves of performance as a function of μ, keeping other parameters constant. It is of added interest that all the experiments are to be allotted the same total number of fitness evaluations (given by `FFEvals`), thus allowing a truly fair comparison. Two questions arise: is a bigger population size going to give (significatively) better results, provided it is allowed to resort on the same number of fitness evaluations? Is there any clear trend in performance as a function of μ?

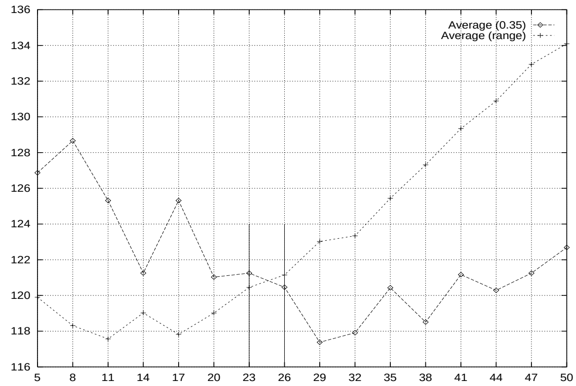

Figure 7: Average results for the two recombination settings. The cross-point is around 25.

To give an answer, fifty different values for the population size are tested, sampling the discrete interval [2, 100] of even numbers, to get a smooth transition between consecutive results. To keep the computational burden within reasonable values, the limit **FFEvals** is set to 4000, thereby allowing for 2000 generations for $\mu = 2$ and only 40 for $\mu = 100$. In practical applications, this would constrain the absolute performance figures to unacceptably low values. However, in this experiment (as in the majority of experiments in this study) we are primarily interested in the *relative* performance with varying μ. In a sense, we are going to explore how does the algorithm behave *in the first 4,000* evaluations of what would normally have been a longer execution. It has been observed [8, 29] that the typical performance curves for a BGA follow the trends shown in Fig. 8 (depending on problem difficulty). Hence the initial performance is able to illustrate how the algorithm is going to behave for a particular setting. Expected variations due to specially lucky (or unlucky) runs are to be mitigated by computing the average over the **NRuns** runs.

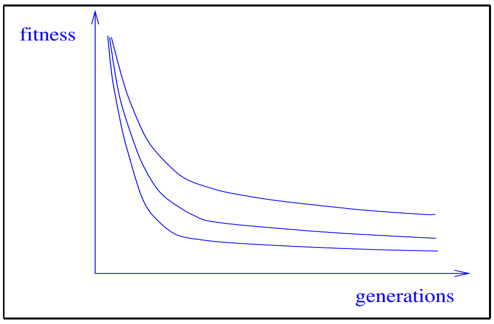

Figure 8: Typical (idealized) BGA performance curves.

Procedure `PopulationSize-Test` ()
{
  Ψ := CM ($\rho = 0.5$, $k = 16$);
  forall Ω **in** {EIR ($\delta = 0.35$), EIR ($range_\delta$)}
    forall $\mu \in \{2, 4, 6, \ldots, 100\}$
      BGA (μ, F_i, k, ρ, Ψ, **NRuns**, **FFEvals**, Ω, τ);
}

Figure 9: `PopulationSize-Test` Algorithm pseudocode.

Operators are set to CM with $\rho = 0.5, k = 16$ for mutation, and EIR ($\delta = 0.35$), EIR ($range_\delta$) for recombination, with $\tau = 25$ and **NRuns**=20 for each setting. The procedure is that of algorithm in Fig. 9. The results are again presented graphically, in Fig. 10 (left) for EIR ($\delta = 0.35$) and (right) for EIR ($range_\delta$). For each configuration, average and best performance curves across the **NRuns** are shown. In addition, a quartic function (polynomial of degree 4) is fitted to the data and plot.

Both sets of curves are similar in that they exhibit the same trend: a sudden initial decrease (more marked for the average performance) up to a turning point from which further improvements are slow but consistent. This clear tendency to better performance answers both questions affirmatively: the first one because of the trend, and the second one because of its smoothness. The bigger μ, the better the results even though the effective number of generations is getting lower and lower.

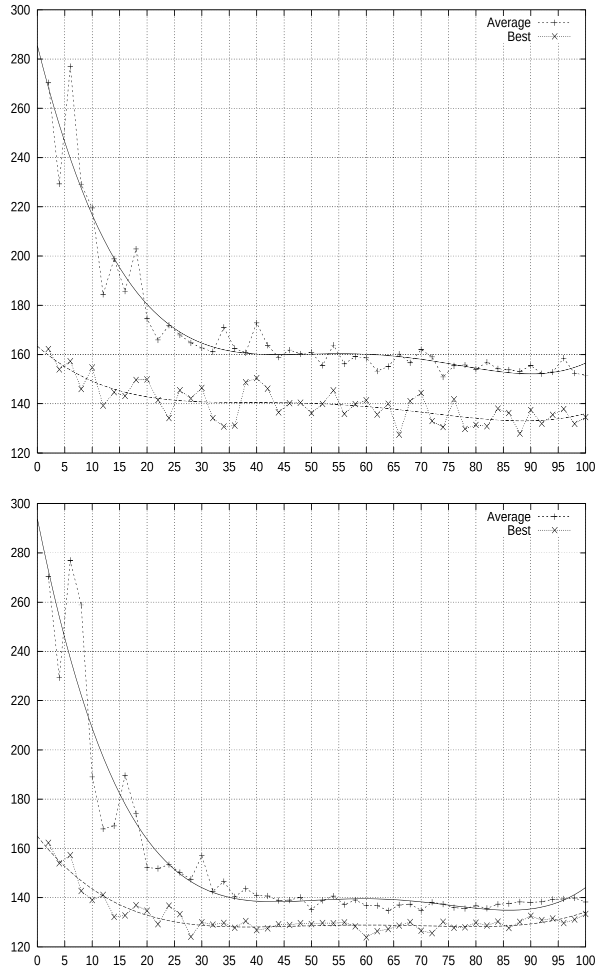

Figure 10: Average and Best BGA performance as a function of μ. Top: for EIR ($\delta = 0.35$). Bottom: for EIR ($range_\delta$). Each point is the result of **NRuns**=20 runs.

More precisely, the interval [35, 40] seems to be the turning point in both cases, then there is a flat plateau ([40, 65]) and from 65 on, the curve decreases in both cases, reaching the lowest zone at [85, 95] for EIR ($\delta = 0.35$) and at [60, 80] for EIR ($range_\delta$). It finally seems to slightly raise again, possibly because

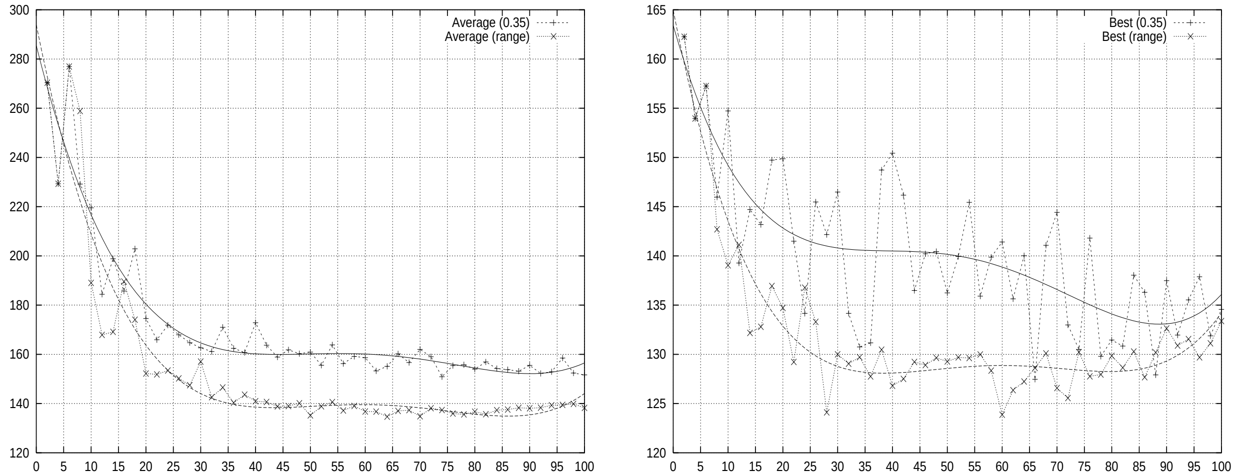

Figure 11: Compared BGA performance as a function of μ. Left: Average. Right: Best results. Each point is the result of `NRuns`=20 runs.

the number of **`FFEvals`** is too small for $\mu = 100$. This collective behaviour is more clearly seen in Fig. 11 (left), where average curves for both settings are plotted side by side. Both average curves have nonetheless similar trends, showing the slight increase at the end. It is also clear from these plots that the curve for EIR ($range_\delta$) is consistently better than that of EIR ($\delta = 0.35$). The same arguments are valid for the Best result curves (Fig. 11, right). In this case, the increase at the end is more marked.

Algorithm	– MSE –	
	Average $\pm \frac{\sigma}{\sqrt{n}}$	Best
BGA (EIR, $\delta = 0.35$)	0.1461 ± 0.0016	0.1354
BGA (EIR, $\delta = range_\delta$)	0.1471 ± 0.0012	0.1407
Annealing + ConjGD	0.1409 ± 0.0053	0.1134

Table 4: Comparative results found by the three optimization algorithms.

6 A comparison of performance

With the knowledge gained so far, it is of interest to perform a further batch of experiments with the clear intention of finding relevant solutions, and to compare them to those found by DBMs. To this end, we select the two settings that have been used so far, but this time allowing the algorithm for `FFEvals` = 100,000 across `NRuns` = 10. The two recombination settings are used with the following parameters: ($\mu = 90$, $\tau = 30$) for EIR ($\delta = 0.35$) and ($\mu = 50$, $\tau = 18$) for EIR ($range_\delta$). As can be seen, these are rather temptative values chosen *inside the regions* of best performance, following the results of sections 5.3 and 5.4.

These two representative BGA configurations are to be compared to a DBM, in as much equal conditions as possible. The method chosen is a powerful combination of two classical optimization methods. It consists of a Conjugate Gradient descent coupled to a Simulated Annealing schedule [1]. This hybrid method is allowed to resort to 10 tries to match the 10 runs of the BGA, and it optimizes the same cost on the same data set, using an identical network architecture. The results are shown in Table 4. This time, data for MSE are directly shown in the format Average $\pm \frac{\sigma}{\sqrt{n}}$, where σ is the standard deviation and n the number of runs, along with the best solution found. There are some interesting observations:

1. The average values found by the two BGAs and the Annealing + Conjugate Gradient are very close indicating a comparable average performance, which is by itself very notable, given the sophisticated nature of the chosen DBM. However, the latter method finds an extremely good solution not matched by any BGA setting. In spite of this, the comparable average behaviour and the markedly lower variability shown by the BGAs indicate a general feasibility for the task, perhaps contrary to what one could pessimistically expect of an evolutionary algorithm.

2. The fact that EIR ($range_\delta$) gives poorer results

is a little deceptive in light of its previous results. However, the lower deviations may be an indication that it has not been used in or near its optimal population sizes and truncation thresholds. This could also apply to EIR ($\delta = 0.35$).

3. In what regards the duration of training, the DBM took about six times (actually six hours) the time used by any of the BGA executions (counting all the `NRuns`), on a shared SUN^{tm} Ultra-60 System.

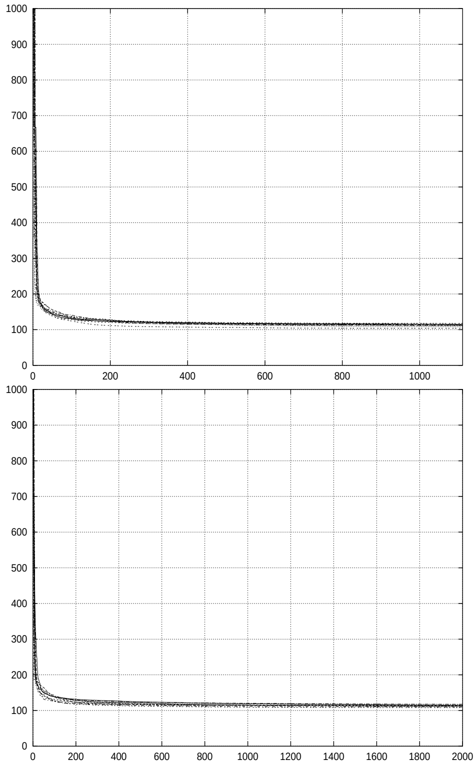

Figure 12: Actual BGA performance curves. Top: for EIR ($\delta = 0.35$). Bottom: for EIR ($range_\delta$). The x-axis shows the generations and the y-axis shows the fitness. Note the different number of generations due to the different values of μ used. In each case, all ten curves are plotted, illustrating in both situations the very low variability.

The obtained performance curves are shown in Fig. 12 for the two selected recombination settings. Each of the curves traces the evolution of the best solution found up to a given generation. They can be compared against those in Fig. 8. By looking at them, it can be seen how the BGA quickly (in a couple of hundred generations) finds relatively good solutions, then stabilizes and continues to minimize the cost but at a very low pace.

7 Summary of findings

We are now in a position to summarize the main findings of the experiments. For some aspects, specific configurations (one or two) are shown to be specially better while, in other cases, a range of values can be proposed. The intention is then to provide an interval from which a careful problem-dependent experimental setup could draw samples. For instance, in our case study, it is improbable that we have chosen the best possible value for τ or μ in the comparison to the DBM, but the chances that we are in a meaningful interval are high and a specialised search for practical solutions to this problem would surely improve on the solutions temptatively obtained. Recall that the same τ has been used for two radically different recombination settings to obtain the results on μ. In a real application, separate experiments should be performed. A summary of recommended BGA operators and parameters follows:

Mutation operator: Continuous Mutation with parameters $\rho = 0.5$ and $k \in \{8, 16\}$. Discrete Mutation can also be a choice, although not for the problem studied. For both operators, this choice of parameters is in general favour of bigger mutation steps.

Recombination operators: Extended Intermediate Recombination (EIR) with $\delta = 0.35$ (good behaviour on average and w.r.t. the best result found) and EIR with dynamic δ calculation (best average behaviour).

Truncation threshold τ: found to be approximately $\tau \in [26, 35]$ for EIR ($\delta = 0.35$) and $\tau \in [8, 20]$ for EIR ($\delta = range_\delta$) for $\mu = 100$. This value is likely to depend on μ.

Population size: for the selected τ of 25, performance reaches the optimum zone in the [85, 95] interval for EIR (0.35) and within [60, 80] for EIR ($range_\delta$). Again, the interaction between τ and μ prevents from drawing far sighted conclusions, but the performance trends found for both parameters (sections 5.3 and 5.4) and their relationship are very definite and deserve closer attention.

8 Concluding remarks

The search for generally adequate genetic operators and other BGA parameters —like optimal population size μ and truncation threshold for selection τ— for supervised training of a multilayer perceptron (MLP) has to be carried in a principled way, if the aim is to tailor the BGA specifically for this kind of problem. The reasons for this research are numerous, and among them we mention the following:

1. We believe that there effectively *are* BGA configurations (specially, genetic operators) in favour of the particularities of this task. Specialised operators can be devised, but the knowledge gained on the existing ones constitutes a good departure point. Moreover, genetic operators behave at their best for certain balanced combinations of μ and τ. Thus, the relationship between these two important parameters must be further clarified in this context, specially in light of the quite opposite influence they have on the two recombination settings that have stood out –EIR ($\delta = 0.35$) and EIR ($range_\delta$).

2. It is also our belief that the potential of EAs to solve this kind of task has not yet been fully employed; the experiments in this work are initial steps in this direction. Hybrid methods in which an EA takes the model selection part and the numerical optimization is left to traditional DBMs not only are very time consuming but they do not make the most of EAs. This is because, due to the high computational demands, only small populations for a very low number of generations can be run. Also, the search is biased by an external factor –the DBM– that, due to its nature, is to yield a suboptimal solution that also depends heavily on the initial weights, learning rate, momentum, and other DBM parameters (not controlled by the EA). In addition, the (reasonable) model space is discrete and usually small, and it makes sense to take profit of incremental/decremental steps until some criterion is fulfilled.

3. In the same vein, the hybridization of methods involves a double set of parameters to be optimized at the same time. Not only the learning parameters of the DBM have to be set, but the parameters of the EA have also to be chosen over a rich variety. And it is likely that interactions arise between both methods, making the search on the joint parameter space an unfeasible task.

   For these reasons, we postulate for EAs to solve the numerical optimization problem only, among which continuous EAs (like the BGA or ES) are possibly better suited than traditional binary-coded GAs. To this end, the classical test suites for EAs should be widened with those used in neural benchmarking. This would surely open new directions in EA development.

As an initial step to simplify towards this end, we have carried a thorough study over genetic operators and their parameters, aimed at selecting those setting that seem to be in favour of better solutions. Arguments about the use of recombination operators range from good news: recombination is good and necessary because it compensates/repairs deviations, to not so good: recombination is to handle with care because, in the neural context, offspring is likely to be generally bad, whether the two parents were very good or very bad. Only in the situation where one parent is fairly good and the other is not, the offspring has a chance to improve on both, with luck playing an important role. In this situation, really good solutions have a little chance to be generated and will have to be exploited by mutation.

A partial solution for this –or, at least one that alleviates the problem– is the use of recombination operators with high variance, able to leave the scope of the parents, possibly in a controlled way, depending on how close they are. In addition, the mutation operator should be vigorous enough as to tune (exploit) a solution proposing non-trivial changes, since a fairly small change (e.g. in the fourth decimal) in one of the weights is not going to affect network performance. However, a stronger change (e.g. in the second decimal) *can* affect it, either for the better (it will be thus kept) or for the worse (it will be kept if it still marks as one of the best τ percent).

This intuitive thinking is supported by the results of this work. For recombination, the EIR family of operators stands out from the rest. These operators are the only ones that allow to exit the parents's scope, the amount of which is controlled by their parameter δ. Specifically, a big value of $\delta = 0.35$ and the highly-allowing method $range_\delta$, which dynamically sets it to the maximum feasible value, have been shown to be the best. For mutation, high values of ρ combined with low values of k (actually, the respective high and low extreme of the intervals sampled) yield the best results. This setting favours higher average mutation steps, up to reasonable values. For example, some simple calculations show that, for the selected values $\rho = 0.5$ and $k = 16$, and the weight range of $[-10, 10]$, the expected mutation step is around 0.04 (positive or negative), very approximately the mean change in the second decimal.

With respect to the values of μ, τ, the rationale is not to keep very specific values (that would be particular to the case study at hand) but to look for an average behaviour, a trend or tendency, or an interval of values that is consistently yielding better performance than others. Although some temptative settings have been devised out of the specific behaviours found, making precise the relation between μ and τ, will give a more principled substrate on top of which further work can be developed.

In the same vein, a specific search through the space of architectures (model selection) would certainly have raised overall performance. A higher number of hidden units, the use of a sigmoidal activation in the output units, a different cost function (e.g. cross-entropy) or a different range of weights would have had an impact on performance. In fact, a series of experiments using $\sigma =$ logistic, consistently yielded superior results, but this is not solely attributable to the BGA, but to the beneficial effect of a bounded output.

The choice of the cost function and the range of weights is likely to certainly exert an influence in BGA performance because, although the underlying task is basically the same, the BGA sees a different fitness function and has a different range where to generate solutions to it. For instance, it is specially likely that a function not suitable for DBM training, the number of correctly classified examples, could give good results. All this remains to be worked out in the future.

In all, the results, corresponding to an initial study –and specifically a study of relative performance– should still not be taken as a useful comparison to those obtained by DBMs. Moreover, it may be argued that the two selected settings have been excessively tailored to the task at hand. We have no argument for this until further experimentation, with a rich variety of neural problems, and a theoretical as well as practical work towards the characterization of the relationship between μ and τ has been done.

However, in light of the obtained results, it is our hypothesis that, once the BGA is better tailored to the general task of supervised learning from examples –in particular, devising other fitness functions or specialised operators– the time will come to perform a more rigorous comparison in a carefully devised experimental design, possibly resorting to a test-bed of known neural benchmarking problems.

Bibliography

[1] Ackley, D. *A connectionist machine for genetic hillclimbing.* Kluwer Acad. Press, 1987.

[2] Bäck, Th. *Evolutionary Algorithms in Theory and Practice.* Oxford University Press, New York, 1996.

[3] Bäck, Th. Evolution Strategies: An alternative Evolutionary Algorithm. Technical Report of the Informatik Centrum, Dortmund.

[4] Belanche, Ll. A Study in Function Optimization with the Breeder Genetic Algorithm. LSI Research Report LSI-99-36-R. Universitat Politècnica de Catalunya, 1999.

[5] Bishop, C. *Neural Networks for Pattern Recognition.* Clarendon Press, 1995.

[6] De Falco, I., Della Cioppa, A., Natale, P., Tarantino, E. Artificial Neural Networks Optimization by means of Evolutionary Algorithms. In *Soft Computing in Engineering Design and Manufacturing*, Springer-Verlag, London, 1997.

[7] De Falco, I., Iazzetta, A, Natale, P., Tarantino, E. Evolutionary Neural Networks for Nonlinear Dynamics Modeling. In Procs. of Parallel Problem Solving From Nature V (PPSN V), Amsterdam, The Netherlands, 1998.

[8] De Falco, I., Del Balio, R., Della Cioppa, A., Tarantino, E. A Comparative Analysis of Evolutionary Algorithms for Function Optimisation. In Procs. of the Second Workshop on Evolutionary Computing (WEC2), Nagoya, JAPAN.

[9] Goldberg, D.E. *Genetic Algorithms for Search, Optimization & Machine Learning.* Addison-Wesley, 1989.

[10] Haykin, S. *Neural Networks: A Comprehensive Foundation.* MacMillan, 1994.

[11] Hertz, J., Krogh, A., Palmer R.G. *Introduction to the Theory of Neural Computation*, Addison-Wesley, Redwood City, 1991.

[12] Hopfield, J.J. Neural Networks and Physical Systems with Emergent Collective and Computational Abilities. In Procs. of the Natl. Academy of Sciences, USA, Vol. 79, pp. 2554-2558, 1982.

[13] Kohonen, T. *Self-Organization and Associative Memory.* Springer-Verlag, Berlin, 1988.

[14] Lawrence, S., Giles, C. L., Tsoi, A. C. What Size Neural Network Gives Optimal Generalization? Convergence Properties of Backpropagation. Technical Report UMIACS-TR-96-22 and CS-TR-3617. Institute for Advanced Computer Studies. Univ. of Maryland, 1996.

[15] Møller, M. A scaled conjugate gradient algorithm for supervised learning. *Neural Networks*, 6(4): 525-533, 1993.

[16] Montana, D.J., Davis, L. Training FeedForward Neural Networks using Genetic Algorithms. In Procs. of the 11th Intl. Joint Conf. on Artificial Intelligence. Morgan Kaufmann, 1989.

[17] Moody, J. Prediction Risk and Architecture Selection for Neural Networks. In *From Statistics to Neural Networks: Theory and Pattern Recognition Applications.* Cherkassky, Friedman and Wechsler (eds.), NATO ASI Series F, Springer-Verlag, 1994.

[18] Mühlenbein, H., Schlierkamp-Voosen, D. Predictive Models for the Breeder Genetic Algorithm. *Evolutionary Computation*, 1 (1): 25-49, 1993.

[19] Prechelt, L. Proben1: A set of Neural Network Benchmark Problems and Benchmarking Rules. Facultät für Informatik. Univ. Karlsruhe. Technical Report 21/94, 1994.

[20] Radcliffe, N.J. Genetic set recombination and its application to neural network topology optimization. Technical Report EPCC-TR-91-21. Edinburgh Parallel Computing Centre. Univ. of Edinburgh, Scotland, 1991.

[21] Rumelhart, D.E., McClelland J.L. (eds.) *Parallel Distributed Processing: explorations in the microstructure of cognition; vol. 1: Foundations.* MIT Press, Cambridge, MA, 1986.

[22] Shaffer, J.D., Whitley, D., Eshelman, L.J. Combination of Genetic Algorithms and Neural Networks: A Survey of the State of the Art. In *Combination of Genetic Algorithms and Neural Networks.* Shaffer, J.D., Whitley, D. (eds.), pp. 1-37, 1992.

[23] Shewchuck, J.R. An Introduction to the Conjugate Gradient Method without the Agonizing Pain. Technical Report of the School of Computer Science, Carnegie Mellon Univ, 1994.

[24] Voigt, H.M., Mühlenbein, H., Cvetkovic, D. Fuzzy recombination for the continuous Breeder Genetic Algorithm. In Procs. of the 6th Intl. Conf. on Genetic Algorithms, (ed.) L. Eshelman, Morgan Kaufmann: San Mateo, 104-113, 1995.

[25] Whitley, D. Genetic Algorithms and Neural Networks. In *Genetic Algorithms in Engineering and Computer Science.* Periaux, Galán, Cuesta (eds.), John Wiley, 1995.

[26] Whitley, D., Starkweather, T., Bogart, C. Genetic Algorithms and Neural Networks: Optimizing Connections and Connectivity. *Parallel Computing*, 14: 347-361, 1990.

[27] Whitley, D., The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best. In Procs. of the 3rd Intl. Conf. on Genetic Algorithms, (ed.) Schaffer, J.D., Morgan Kaufmann: San Mateo, 116-121, 1989.

[28] Yao, X. A Review of Evolutionary Artificial Networks. *Intl. J. of Intelligent Systems*, 8(4): 539-567, 1993.

[29] Zhang, B.T., Mühlenbein, H. Evolving Optimal Neural Networks Using Genetic Algorithms with Occam's Razor. *Complex Systems*, 7(3): 199-220, 1993.

[30] Zheng, Z. A Benchmark for Classifier Learning. Technical Report 474. Basser Dept. of Computer Science. The Univ. of Sydney, Australia, 1993.