Evolutionary Training of CLP-Constrained Neural Networks

Joost N. Kok[1], Elena Marchiori[1,2], Massimo Marchiori[3], Claudio Rossi[4]

[1]*Dept. of Computer Science, University of Leiden*
P.O. Box 9512, 2300 RA Leiden, The Netherlands
joost@cs.leidenuniv.nl

[2]*CWI*
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
elena@cwi.nl

[3]*Dept. of Pure and Applied Mathematics, University of Padova*
via Belzoni 7, 35131 Padova, Italy
max@hilbert.math.unipd.it

[4]*Dip. di Matematica Applicata e Informatica, Università di Ca' Foscari*
via Torino 155, 30173 Mestre-Venezia, Italy
crossi@moo.dsi.unive.it

Abstract

The paper is concerned with the integration of constraint logic programming systems (CLP) with systems based on genetic algorithms (GA). The resulting framework is tailored for applications that require a first phase in which a number of constraints need to be generated, and a second phase in which an optimal solution satisfying these constraints is produced. The first phase is carried by the CLP and the second one by the GA. We present a specific framework where ECL^{i}PS^e (ECRC Common Logic Programming System) and GENOCOP (GEnetic algorithm for Numerical Optimization for COnstrained Problems) are integrated in a framework called CoCo (COmputational intelligence plus COnstraint logic programming). The CoCo system is applied to the training problem for neural networks. We consider constrained networks, e.g. neural networks with shared weights, constraints on the weights - for example domain constraints for hardware implementation - etc. Then ECL^{i}PS^e is used to generate the chromosome representation together with other constraints which ensure, in most cases, that each network is specified by exactly one chromosome. Thus the problem becomes a constrained optimization problem, where the optimization criterion is to optimize the error of the network, and GENOCOP is used to find an optimal solution.

Note: The work of the second author was partially supported by SION, a department of the NWO, the National Foundation for Scientific Research. This work has been carried out while the third author was visiting CWI, Amsterdam, and the fourth author was visiting Leiden University.

1 Introduction

There are two major paradigms for problem solving that play a distinguished role in Computing Science and Artificial Intelligence: Reasoning and Search [16]. Although every automated reasoning process involves search aspects (e.g. how to traverse the derivation tree of a logic program), the two paradigms and the corresponding research communities are rather disjoint. As a consequence, high level paradigms for reasoning generally incorporate rather inefficient search techniques, due to the exponential size of the search space they use. In this paper we try to go beyond the borders of each individual paradigm by applying them in combination. More specifically, the paper is concerned with the integration of constraint logic programming systems (CLP) with systems based on genetic algorithms (GA). The resulting system is tailored for applications that require a first phase in which a number of constraints need to be generated, and a second phase in which an optimal solution satisfying these constraints is produced. The first phase is carried by a suitable CLP and the second one by a suitable GA.

In order to test the adequacy of such an integration, we have combined ECL^{i}PS^e (ECRC Common Logic Programming System) and GENOCOP [14] (GEnetic algorithm for Numerical Optimization for COnstrained Problems) in a framework called CoCo (COmputational intelligence plus COnstraint logic programming). We have applied the CoCo system to solve the training problem for constrained neural networks. Neural networks have been used in many applications, for example in planning, control, content-addressable memory, optimization, constraint satisfaction, and classification (see e.g. [5]). They are being promoted for their robustness, massive parallelism, and ability to learn. The training of a neural network is a major design step: Roughly, one has to find a set of weights that minimizes the neural network's error on an initial set of input-output examples called the training set. The standard method for that problem is a local gradient search method known as the back-propagation algorithm [15]. Since the problem's error surface is highly dimensional and usually it contains many local minima, this method can get stuck in local minima. Moreover, it needs gradient information. Alternative approaches based on genetic algorithms have been proposed, which apply for instance to the following types of neural networks.

1. recurrent networks;
2. networks with non-differentiable error criteria (for example due to non-differentiable transfer functions, error measures that use absolute values, bonuses for small weights etc.).

For this kind of neural networks the standard back-propagation learning rule is not in general applicable. Nevertheless, this type of networks can be very useful in applications (for example in applications based on time sequences [6]). Genetic algorithms usually avoid local minima by searching several regions simultaneously. They act on a population of trial solutions, and use information on some performance value describing the 'quality' of a set of weights. Thus they do not use gradient information, and do not require restrictions on the network topology. However, the drawback of genetic algorithms is that they seem to have difficulties in the fine tuning of the parameters [9]. Moreover, there is the "competing conventions problem" [4, 17]: when one chooses a representation (in this case for a neural network), then it can be the case that the same individual has more than one representation. This enlarges (in an artificial way) the search space and affects the convergence speed of the algorithms. Therefore, the programmer has to find a clever representation such that this does not happen.

In this paper we introduce an automatic tool for dealing with the competing conventions problem, based on the following novel approach. We consider constrained networks, e.g. neural networks with shared weights, constraints on the weights - for example domain constraints for hardware implementation - etc. Moreover, we introduce other constraints which ensure that in most cases each network is specified by exactly one chromosome. Thus the problem becomes a constrained optimization problem. The optimization criterion is to optimize the error of the network (usually this error includes a sum over all the training patterns of the network), and the constraints specify the domain constraints on the weights and some constraints for avoiding the competing conventions problem. More precisely, a constraint logic program in ECL^{i}PS^e is used, which generates constraints on the weights such that each network has in most cases a unique chromosome representation. These constraints and the optimization function are given as input to GENOCOP, which searches for an optimal solution that satisfies the constraints. The advantages of this approach are:

1. we do not have to find a special representation;
2. we can easily integrate through the constraint logic program the constraints for the competing conventions problem with other constraints (for example shared weights, domain constraints etc.);
3. the CLP system checks for the satisfiability of the constraints.

So we consider constrained optimization problems for which a set of relatively simple constraints, generated by a suitable constraint logic program, restricts the search space, and for which the optimization criterion can be rather complex (e.g. a non-differentiable, non-linear function). For this class of optimization problems, genetic algorithms are valuable search tools. Since the weights are real numbers, and they are constrained, a natural choice is to employ a system that can handle constraints and where the data are encoded using real numbers, instead of the original bit-encoding. The GENOCOP system satisfies both these requirements. We have used ECL^{i}PS^e as constraint logic programming system. A nice feature of ECL^{i}PS^e is the possibility to structure programs by means of modules. We have used modules for describing various kinds of constraints that one can impose on the weights of a neural network, depending for instance on the form of the activation function, or on the symmetry of the problem. Moreover, modules have been used to specify various kinds of chromosome representations.

The paper is organized as follows. In the next section the integration of ECL^{i}PS^e and GENOCOP is discussed. Section 3 introduces the new approach for the training of CLP-constrained neural networks. In Section 4 some computational results and evaluation are given. Section 5 discusses related work on the integration of Constraint Logic Programming and Computational Intelligence. Finally, Section 6 contains some conclusions and future work.

2 CoCo: Integrating ECL^{i}PS^e and GENOCOP

In this section we describe the platform that is used for performing our experiments on the evolutionary training of constrained neural networks. First, we motivate the need for a framework based on two different systems. Next, we briefly discuss the two systems used in the application of this paper, ECL^{i}PS^e and GENOCOP, and their integration.

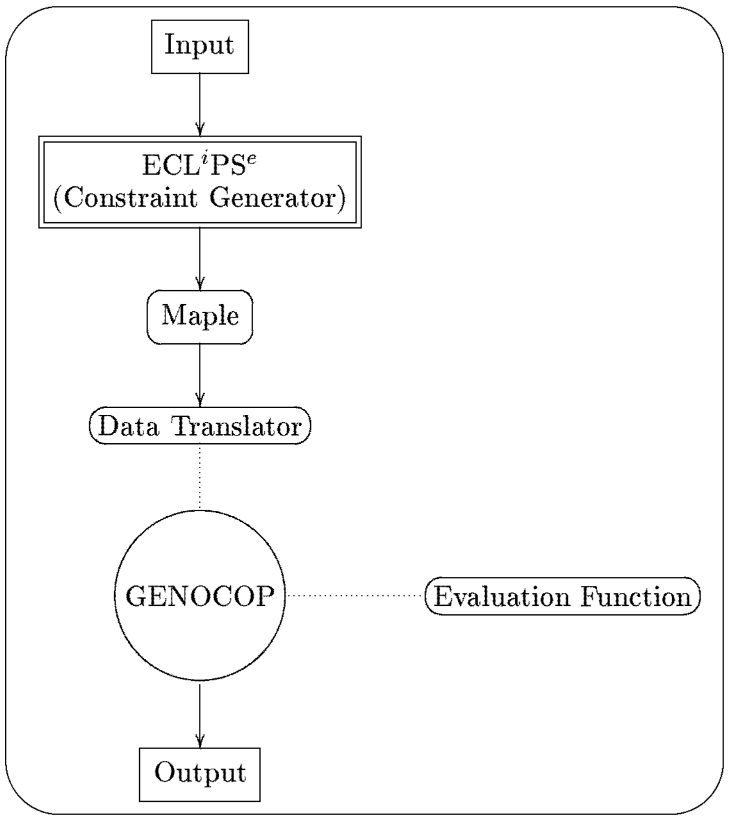

Figure 1: The CoCo Framework

We consider applications that require a first phase in which a number of constraints need to be generated, and a second phase in which an optimal solution, w.r.t. a given objective function, satisfying these constraints is produced. Examples of such applications include design of VLSDC, finance, medical engineering and many others (cf. e.g. [8]). These applications have been tackled using different methods, for instance approximation methods, like genetic algorithms and neural networks, or exact methods like those of operational research or constraint logic programming. Recent works have shown the advantages of an integrated use of exact and approximation methods. For instance, constraint logic programming and neural networks have been integrated in a system called PROCLANN [12], which exploits the elegant formalism of CLP and the efficiency of constraint solvers based on stochastic systems, like GENET [18].

The specific integration we consider for our application, called CoCo (COmputational Intelligence plus COnstraint Logic Programming), employs the CLP system ECL^{i}PS^e for generating constraints, and the GA system GENOCOP for the optimization phase. We describe briefly the two systems. The reader is referred to the rich documentation on ECL^{i}PS^e available from ECRC at the ftp address ftp.ecrc.de/pub/ECRC_tech_reports/reports, and to the book by Michalewicz [14] for GENOCOP. ECL^{i}PS^e (ECRC Common Logic Programming System) is a Prolog based system built to facilitate the integration of various programming extensions. In particular, it includes a number of libraries, like the one for treating lists, and a constraint solver over the rationals, available in the library `r.pl`. The solver uses a combination of the Simplex algorithm and Gaussian elimination to solve a system of arithmetic constraints consisting of linear equalities and inequalities. We have used the rational constraint solver in CoCo. Moreover, ECL^{i}PS^e allows to structure programs by means of modules. We have used modules for describing various kinds of constraints for our application. GENOCOP (GEnetic algorithm for Numerical Optimization for COnstrained Problems) is an evolutionary program which can handle a large class of optimization problems with linear constraints and any objective function. It uses a floating point representation for chromosomes, and special 'genetic' operators which guarantee that all the chromosomes remain within the constrained solution space. This is possible because only a set of linear constraints, including domain constraints, equalities and inequalities, is allowed. The relevance of this approach is that it provides a general and problem independent way to handle constraints in optimization problems. Hence the system can be easily integrated in our framework.

We conclude this section with issues on the integration of ECL^{i}PS^e and GENOCOP. In order to pass the set of constraints produced by the ECL^{i}PS^e program to GENOCOP, we have designed an ECL^{i}PS^e program that gets the constraints and write them into a file, together with other information about the system, like the number of variables, the number of 'slack variables' (these are variables introduced by the constraint solver when using the Simplex algorithm and Gaussian elimination), and their names. In this first stage of the integration we use files for the exchange of information between ECL^{i}PS^e and GENOCOP, but in the future we plan to use ECL^{i}PS^e's External Language Interface to incorporate GENOCOP into ECL^{i}PS^e. One of the difficulties we encountered was that the rational constraint solver of ECL^{i}PS^e is based on the Simplex Algorithm, hence is tailored towards equalities (i.e. every inequality is replaced by an equality using 'slack variables'). However, the GENOCOP system is based on inequalities. We solved this problem by using the Maple system to remove the slack variables, and to re-introduce the inequalities. However, a more elegant approach we intend to investigate in the future, is the use of an interval constraint solver.

Next, we modified GENOCOP in order to allow it to read the constraints in a different format. This has been done replacing the GENOCOP function which reads the data from the file with a new one that reads the equations in the ECL^{i}PS^e form.

Finally, the evaluation function, which must be provided separately as a C function and is part of the GENOCOP source code, was also modified in order to avoid the need of the re-compilation each time the parameters of the network are modified.

3 Training a Constrained Neural Network

We have introduced the CoCo framework, which integrates ECL^{i}PS^e and GENOCOP. In this section we apply CoCo to a specific problem, namely the training of a constrained neural network. We consider feedforward neural networks (FFNN) (cf. [5]), where there are only connections from nodes of one layer to nodes of the successive layer. For the generalization of the results to recurrent neural networks, the reader is referred to [10].

The program is composed by several modules. At top level, there are three main modules (see Figure 2):

1. The first **Start Module** asks the user the representation of the FFNN, and builds up the corresponding data that will be used in the sequel.

2. The second **Constraining Module** produces constraints on the weights in order to avoid the competing conventions problem.

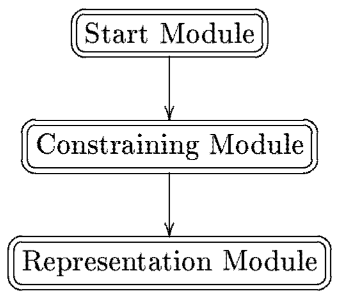

Figure 2: The Constraint Generator System (Top Level)

3. The third **Representation Module** is responsible for the translation of these constraints into the chosen genetic representation.

We now analyze the behaviour of the three components in more detail.

3.1 The Start Module

The user has to give as input the (constrained) network. As far as the network is concerned, instead of giving the whole graph, a compact and user-friendly codification can be used: the input is a list of integers specifying the number of nodes occurring in the network layers (the i-th element of the string corresponds to the i-th layer). So, for instance, the string $[5, 4, 7]$ represents a network with 5 nodes in its first layer, 4 in the second and 7 in the third. Note that this compact representation of the network graph is possible since we employ layered feedforward neural networks.

Furthermore, the user can provide the constraints on the network weights. Since we are dealing with a constraint programming language, the constraints can be very complex, due to the power of the ECL^{i}PS^e rational constraint system. Each node is assigned a unique indexing number, counting progressively from 1 till the last node. The order is from left to right, and from the first layer of the network (the input one) to the last one (the output). The weight from the i-th node to the j-th one of the next layer is indicated with $WiTj$. Then, every constraint can be expressed using these variables, the ECL^{i}PS^e operations for rational constraints (e.g. sum, difference, multiplication etc.), and the ordering relations (equality, disequality, $>$, $<$, $<=$, $>=$).

From this input, the data structures for the corresponding network are built. All this work is done by the Start module.

Observe that using constraints of the form $WiTj = 0$ we can eliminate arcs: hence, the system is also able to cope with *partially connected* layered feedforward neural networks.

3.2 The Constraining Module

In the second module, the effective computation of the constraints solving the competing conventions problem is performed. As mentioned in the introduction, this problem consists in the fact that many structurally different networks can represent the same functional mapping (see [17, 4]). This lack of a unique representation, as well known, leads to serious drawbacks with evolutionary training algorithms (see e.g. [1]).

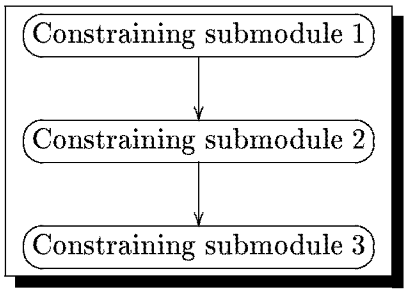

Figure 3: The Constraining Submodules Chain

The first problem is due to genetic recombination: if two functionally similar but structurally different parents are recombined via a crossover operator, the produced offspring is very likely to be completely inappropriate, since the two different encodings clash.

The second crucial problem is, of course, the explosion of the search space: with a domain much larger than the effective problem requires, the efficiency of the genetic system badly degrades (the chance of successful crossover recombinations, the prime factor to consider in genetic algorithms, decreases by far).

The importance of the above two points is substantiated by the fact the competing conventions problem affects *exponentially* the size of the domain: if a network has n hidden neurons, there are always $n!$, and in most of the cases at least $n!2^n$, different networks which are functionally the same.

3.2.1 The Constraining Submodules Chain

The idea then is to produce a suitable set of constraints that in most of the cases eliminates the competing conventions problem.

Abstractly, we can see the competing conventions problem as the existence of some transformations that take a network and give a structurally different but functionally equivalent one (*ccp-transformations* for short).

There are two main classes of ccp-transformations (see [17, 4, 1, 10]). The first class is the permutation of k nodes belonging to the same hidden layer. This affects the search space with a maximal total complexity of $n!$ (n is the number of hidden neurons).

The second class is present if the activation function is odd: given some subset of hidden nodes, flip the sign of all the weights incoming or outcoming from these nodes. This affects the search space with a maximal total complexity of 2^n.

The approach we have developed is to generate constraints by a chain of submodules, called *constraining submodules* (see Figure 3):

- A first submodule has the task to produce constraints preventing the problems arising from the first class of ccp-transformations. The idea is that for every layer but for the last two there must be at least a neuron with all of its output weights totally ordered. That is to say, if such neuron has output weights $o_1, \ldots, o_m$, there must be a permutation π of $\{1, \ldots, m\}$ such that $o_{\pi(1)} < o_{\pi(2)} < \ldots < o_{\pi(m)}$. This ordering chain will be a produced constraint. Such total ordering prevents the neurons of the successive layer to be permuted, and so repeating the argument layer by layer ensures that the first class of ccp-transformations cannot be applied. Note that a strict ordering on some weights is generated. However, one can replace $<$ by $\leq$ in those cases where the resulting constraints are not satisfiable.

Note that at first sight it could seem that it suffices to take as π the identity permutation and to select a fixed neuron for each layer (e.g. the leftmost one). However, where the real power and flexibility of constraint logic programming plays a role, is that we have also to ensure that the produced constraints are compatible with the set of those constraints on the network weights already present: the *input constraints* provided by the user (since we are dealing with *constrained networks*), and also the constraints that can be possibly imposed *afterwards*, to get rid of other possibilities of competing conventions problems (e.g. the next two submodules). This is dealt with in a completely elegant and transparent way by integrating sorting techniques and usage of backtracking in the constraint logic program. We will come back to this point later on.

- A subsequent submodule faces the second class of ccp-transformations. The imposed requirement is that for every layer there must be a neuron with two output weights o_i and o_j that are constrained by a strict inequality $o_i < o_j$. This prevents flipping on that neuron since then it should hold $-o_i < -o_j$, that is $o_i > o_j$, a contradiction.

- Finally, a third submodule further cuts the search space by imposing ordering conditions that avoid flipping of arcs coming from the input nodes: it is imposed that there is an ordering of the input nodes such that for every input node there is an outcoming arc whose weight is strictly less than all the outcoming arcs of the successive input node.

The advantages of such implementation, besides the clarity of the program, are several.

First, it allows easy control over the constraint generations: if the user is aware of some particular competing conventions problems due to the particular activation function chosen, or due to symmetries in the data, etc., (s)he can safely add a submodule performing this task. This is also the case if (s)he knows that some cases will *not* occur, in which case some submodules which are not needed and that will maybe restrict too much the search space can be safely removed.

Second, modules are not stand-alone objects increasingly pruning the search space, but they flexibly interact. Indeed, suppose the first submodule has produced its constraints, and the second submodule is activated. It may be the case that this second submodule cannot find a consistent set of constraint, because of the constraints produced by the first submodule. In this case, the failure makes the execution *backtrack* into the first submodule, where a different constraint is generated, and the execution of the second submodule starts again. The flexibility of backtracking so allows automatic re-setting of the constraints produced by a submodule in response to the requests of the next submodules.

In this case, i.e. if the second module finitely fails, the execution is re-started from a suitable state, among the intermediate states obtained during the execution of the first module, and from a suitable point in the second submodule. This flexible technique allows the automatic re-setting of the constraints produced by a submodule in response to the requests of the next submodules.

Furthermore, we plan to implement a technique, called forward-tracking, for dealing with over-constrained information. Suppose that, for the given constrained network, the set of the constraints produced by all the submodules is inconsistent. A naïve implementation would in this case yield failure, thus being of no help. It may however be the case that the execution of the first k submodules succeeds, and that the inconsistency arises from some of the requirements imposed by the $k+1$-th submodule. In this case we can still derive some information in order to prune the search space. The idea is to consider the constraints produced at a suitable point afterwards the execution of the k-th module, and to re-start the execution from some point forwards, in the $k+1$-th submodule. This amounts to impose a precedence among the submodules: submodules occurring in the chain in earlier positions have precedence on those coming afterwards, in the sense that they are executed before, and their results still hold in case some submodule with lower precedence yields failure. Moreover, there are also precedences *inside* a single submodule. This are deduced by exploiting the particular structure of the algorithms used by the submodules. All of them produce constraints on the network 'layer-by-layer', and so a kind of hierarchical precedence *over layers* can be introduced: if the submodule managed to produce constraints for k layers, and then yields a global inconsistency, then the execution is resumed in a forward point by considering only the set of constraints that have been produced till the k-th layer. The reader is referred to [13] for a formal definition and operational semantics of this mechanism.

Finally, let us spend some words on the way each of the two available constraining submodules is implemented. When a constraint is added, the consistency of the obtained store is automatically checked by the constraint solver of ECL^{i}PS^e. It is however important that the store is kept to a reasonable size, to avoid the introduction of too much overhead in the system. The submodules are implemented in such a way that only the constraints on the weights are mantained in the store, that is there is *no* extra information, e.g. on the structure of the network, present on the store.

3.3 The Representation Module

The choice of a suitable representation of the data is of fundamental importance for the successful application of genetic techniques. This is in general a creative task: in the field of evolutionary training, there are some studies that introduced some suitable effective representation for this application (e.g. [17, 19]), but of course it is still questionable what representation is the best one. Therefore, we have neatly separated the building of the constraint part from an actual genetic representation, devoting a module to the first task (as we have previously seen), and a stand-alone module that translates the constraints produced by the first module to constraints that employ a particular representation of the genes.

Thus, the choice of the called submodule is actually parametric: depending on the chosen representation, it is automatically activated the submodule corresponding to the desired representation. So far, we have implemented submodules for the Yoon *et al.* representation ([19]), and for another representation which progressively encodes the output weights of every node layer by layer, from left to right.

4 Computational Results and Evaluation

In this section we give some results of experiments done with the CoCo system. We did two series of experiments: one series of experiments on a standard dataset about Iris flowers [3], and a second series of experiments with a real-life data set that is used to assess how the air pollution affects on children's Peak Expiratory Flow. We did runs with and without the "competing conventions" constraints, put constraints on the weights, and used several non-differentiable error criteria. The results show that the addition of the "competing conventions"

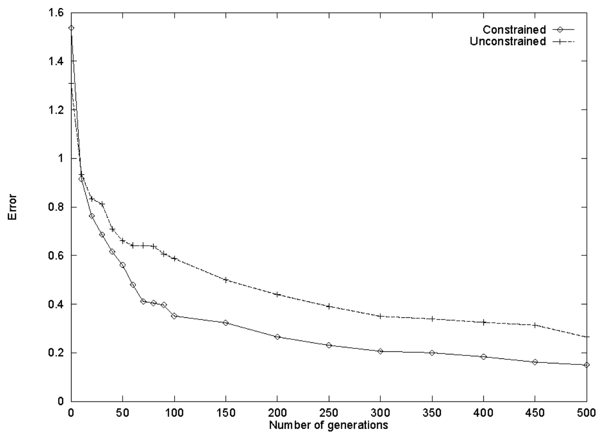

Figure 4: Iris: typical run w.r.t. squared sum of errors

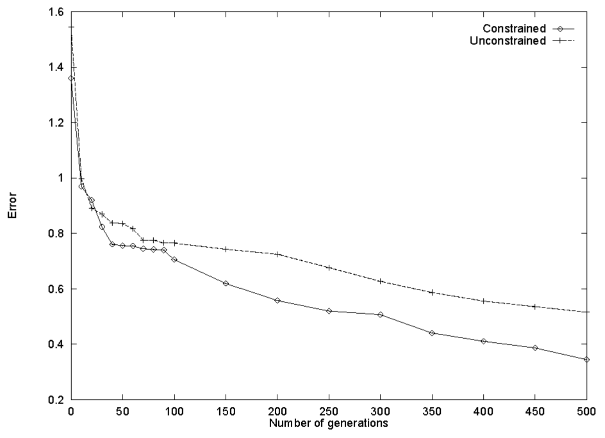

Figure 5: Iris: typical run w.r.t. the sum over the absolute values of errors

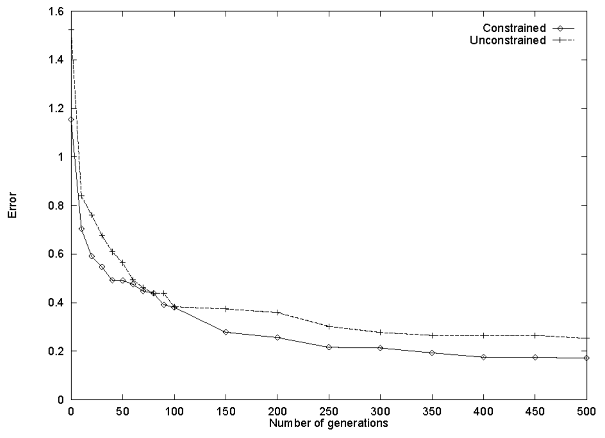

Figure 6: Iris: typical run w.r.t. the sum over the absolute values of the cube of the errors

constraints helps in the convergence. It is difficult to compare results with other methods: our method works for cases where standard learning with back-propagation does not work.

Next we describe our datasets and experiments in more detail. The Iris problem is a classic pattern recognition problem. We are given four parameters describing an Iris flower: the sepal length, the sepal width, the petal length and the petal width.

Each flower is in one of the following three classes: Iris Setosa, Iris Versicolour, and Iris Virginica. The dataset contains 150 examples, 50 for each class (it can be obtained via ftp at ftp://ics.uci.edu/pub/machine-learning-databases/iris/). We used a feedforward neural network with four input units, four hidden units and three output units. The inputs encode the four parameters, and for the output gives one of the three classes. These classes are coded by a so-called one-out-of-three coding scheme. Each output node represents one class, and for a particular input all the three output nodes are required to be zero, except for the one with the right associated class, which has to be one.

Figure 4 shows a typical run. We see that the addition of the "competing conventions"-constraints improves the convergence. Figures 5, 6 show runs with different error-criteria. Instead of the squared sum of errors (Fig. 4), we took the sum over the absolute values of errors (Fig. 5), and the sum over the absolute values of the cube of the errors (Fig. 6).

The second dataset is used for a study of possible correlations between air pollution and children's breathing. It consists of 497 samples, and is available by contacting the authors of the paper. In the experiment we used a feedforward network with 26 inputs, 15 hidden nodes and one output node. The features given as input to the network are data concerning the air pollution estimated by recording ambient levels of pollution and meteorological conditions at a fixed site on a given day (of a four month winter period) and children's pulmonary function measured with their Peak Expiratory Flow (PEF). After a training period, the network should predict the PEF of the evening of a a given day, given the morning, afternoon and evening PEF of the two days before, the morning and afternoon PEF and the data about the air pollution of that day. A typical run of the system is shown in Figure 7. We also see here that the "competing conventions"-constraints improve the convergence.

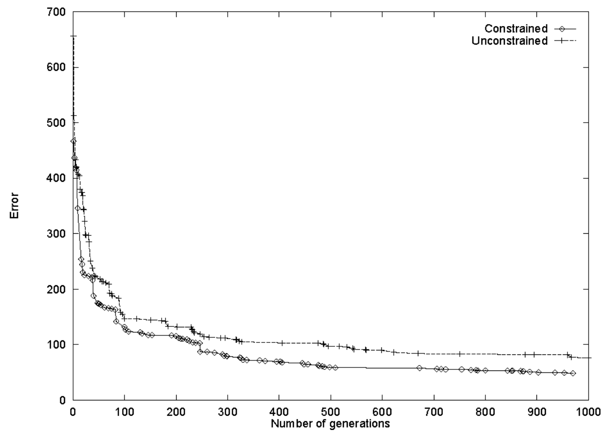

Figure 7: The Peak Expiratory Flow example

5 Related Work

We have illustrated in the previous sections how the proposed integration performs on our specific application. We discuss in the follow some related work that integrates constraint logic programming and computational intelligence.

We are aware of three related papers. A study on the possibility of integrating novel search techniques like genetic algorithms in CLP was done in the CHIC Esprit project [2]. In [11], the author has considered the effect of the combination of local search, genetic algorithms and finite domain solver. In particular, constrained genetic algorithms are proposed, where the creation of a new chromosome is supervised by a constraint solver. Experiments on various instances of the traveling salesman problems have been done, using Prolog and Chip's finite domains. This approach is orthogonal to our, because there CLP is used inside the genetic algorithm, while we use GENOCOP to deal also with constraints, and use a CLP system to produce these constraints.

Another paper on the integration of CLP and GA is [7]. Their approach is a kind of dual of the one in [11], since they use a genetic algorithm for labeling in CLP over finite integer domain. They design a genetic algorithm where variables are labeled with an integer domain, hence a chromosome encodes an area of the search space, which can contain none or many solutions. As a consequence, the genetic operators they define are rather complex, because they apply to data structures instead of values. Moreover, they report the lack of a self adaptive parameter tuning feature, which is essential to achieve a self contained optimization predicate for constraint logic programming over finite integer domains.

Finally, in [12] a programming language called PROCLANN that integrates CLP and neural networks is proposed. Their language is a committed-choice logic programming language with a stochastic constraint solver. The GENET system [18] is used as constraint solver, which translates dynamically a binary constraint problem over finite domain into a suitable neural network, and simulates the network convergence procedure.

6 Conclusion

This paper advocates the integration of constraint logic programming systems (CLP) and evolutionary systems (ES), for applications that require a first phase in which a number of constraints need to be generated, and a second phase in which an optimal solution satisfying these constraints is produced. The first phase is carried by the CLP and the second one by the ES. We have presented a specific framework where ECL^{i}PS^e and GENOCOP are integrated in the CoCo framework. We have applied this framework to the training problem for constrained neural networks. To the best of our knowledge, the idea of constraining a neural network by means of a CLP is original. Standard methods for training neural networks based on back-propagation do not apply in the presence of constraints, and alternative methods based on genetic algorithms are problem dependent, while our framework can be applied to neural networks with any set of constraints and any optimization criterion. Moreover, the results show that training neural networks using our constraint-based approach yields significant improvements.

We think that this approach (integration of CLP with ES) is also useful for many other applications with difficult optimization criteria. For our kind of applications we would need a form of Constrained optimization Logic Programming (CoLP), in which the result of a program is not only a set of constraints, but also an optimization criterion. This optimization criterion should be build during the execution of the program, and it can depend for example on what kind of constraints we add to the store.

Acknowledgments

We would like to thank Maarten Lamers for providing the PEF dataset.

References

[1] J. Branke. Evolutionary algorithms for neural network design and training. In *Proc. of the 1st Nordic Workshop on Genetic Algorithms and its Applications.* Vaasa, Finland, 1995.

[2] A. Chamard, A. Fischler, D. Guinaudeau, and A. Guillaud. Chic lessons on CLP methodology. Technical report, ECRC, 1995. Available via ftp at ftp.ecrc.de/pub/ECRC_tech_reports/reports/.

[3] R.A. Fisher. The use of multiple measurements in taxonomic problems. *Annual Eugenics*, 7(II):179–188, 1936.

[4] P.J.B. Hancock. *Coding Strategies for Genetic Algorithms and Neural Nets.* PhD thesis, Dept. of Computing Science and Mathematics, University of Stirling, 1992.

[5] S. Haykin. *Neural Networks, A Comprehensive Foundation.* Macmillan, 1994.

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

[7] A. Ruiz-Andino Illera and J.J. Ruz Ortiz. Labelling in CLP(FD) with evolutionary programming. In M. Alpuente and M.I. Sessa, editors, *Proc. of the GULP-PRODE'95 Joint Conference on Declarative Programming*, pages 569–580. Poligraph Press, 1995.

[8] J. Jaffar and M.J. Maher. Constraint logic programming: A survey. *J. Logic Programming*, 19,20:503–581, 1994.

[9] H. Kitano. Empirical studies on the speed of convergence of neural network training using genetic algorithms. In *Proc. AAAI*, pages 789–795, 1990.

[10] J.N. Kok, E. Marchiori, M. Marchiori, and C. Rossi. Constraining of weights using regularities. In M. Verleysen, editor, *Proc. of the 4th European Symposium on Artificial Neural Networks.* D facto, 1996. To appear.

[11] V. Küchenhoff. Novel search and constraints - an integration. Technical Report IR-LP-92-16i, ECRC, 1992.

[12] J.H.M. Lee and V.W.L. Tam. Towards the integration of artificial neural networks and constraint logic programming. Technical Report CS-TR-94-14, The Chinese University of Hong Kong, 1994.

[13] E. Marchiori, M. Marchiori, and J.N. Kok. From failure to success with forward-tracking. Submitted, 1996.

[14] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer, 1994.

[15] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[16] H.A. Simon. Search and reasoning in artificial intelligence. *Artificial Intelligence*, 21:7–29, 1983.

[17] D. Thierens, J. Suykens, J. Vanderwalle, and B. De Moor. Genetic weight optimization of a feedforward neural network controller. In *Proc. of the Conference on Artificial Neural Nets and Genetic Algorithms*, pages 658–663. Springer-Verlag, 1993.

[18] E.P.K. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[19] B. Yoon, D.J. Holmes, G. Langholz, and A. Kandel. Efficient genetic algorithms for training layered feedforward neural networks. *Information Sciences*, 76:67–85, 1994.