Intelligent Information Systems 2008
ISBN 978-83-60434-44-4, pages 191–200

Data-driven Constructive Induction in the Learnable Evolution Model

Janusz Wojtusiak

Machine Learning and Inference Laboratory
Department of Health Administration and Policy
George Mason University, Fairfax, VA, USA

Abstract

The *learnable evolution model* (LEM) is a non-Darwinian evolutionary computation method which applies symbolic machine learning to guide the evolutionary optimization process. This paper investigates application of data-driven constructive induction to automatically improve representation spaces in LEM. This includes investigation of methods for modifying representation spaces and methods for creating new candidate solutions from hypotheses learned in the modified spaces. Experimental results indicate that LEM equipped with constructive induction outperforms LEM working only in the original representation spaces.

Keywords: constructive induction, evolutionary computation, learnable evolution model, machine learning

1 Introduction

This research investigates and extends an evolutionary optimization method, called the *learnable evolution model* (LEM), that applies machine learning to search very complex problem solution spaces. By applying machine learning, LEM hypothesizes why some candidate solutions (e.g. designs, complex parameter settings) perform better than others, and uses that knowledge to create new candidate solutions (Michalski, 1998, 2000; Wojtusiak and Michalski, 2006). In particular, this research investigates how to automatically improve representation spaces in which solutions are sought in order to improve and speed-up the optimization process.

The original representation of candidate solutions may not be adequate for the optimization problem, thus its improvement may lead to finding better solutions or finding solutions more efficiently. The process of designing representation spaces is often complicated and requires substantial domain knowledge. Moreover, different representations may be needed at different stages of the optimization process. Because of that, there is a need for automated methods that improve the representation of solutions. In evolutionary computation several methods for automatically improving representation spaces have been investigated. The methods are usually based on based on evolving representations in parallel to finding solutions (Angeline and Jordan, 1994; Reisinger and Miikkulainen, 2007), thus they are different

```
Create an initial population of candidate solutions
  Evaluate candidate solutions in the initial population
  Loop while stop criteria are not satisfied
    Create new candidate solutions by machine learning:
      Identify groups of high- and low-performing candidate solutions
      Apply machine learning to distinguish between the groups
      Instantiate the learned hypothesis
    Evaluate fitness of the new candidate solutions
    Select a new population
```

FIGURE 1: Pseudocode of a general LEM algorithm.

from those discussed in this research. Here, we adopt selected constructive induction (CI) methods used in machine learning for improving representation spaces.

Section 2 of this paper briefly describes the learnable evolution model and its LEM3 implementation, Section 3 discusses constructive induction and its application in LEM, and Section 4 presents results of experimental evaluation.

2 The Learnable Evolution Model

Research on non-Darwinian evolutionary computation is concerned with developing algorithms in which the creation of new candidate solutions in the population is guided by an "intelligent agent," rather than done merely by random or semi-random change operators, such as mutation and/or crossover, employed in the "Darwinian-type" evolutionary methods. The selection of candidate solutions for the new generation from those generated by the intelligent agent is done according to standard methods of selection, or can also be done by employing advanced reasoning. The learnable evolution model (LEM) employs a machine learning program to direct the evolutionary process (Michalski, 1998, 2000; Wojtusiak and Michalski, 2006). Specifically, the program creates general hypotheses indicating regions in the search space that likely contain optimal solutions and then instantiates these hypotheses to generate new candidate solutions.

The learnable evolution model follows a general evolutionary computation schema. Specifically, in LEM, whose algorithm is presented in the Figure 1, creation of new candidate solutions is done by applying hypothesis learning and instantiation.

The assignment of high- and low-performing candidate solutions into the H-group and L-group allows these sets to be used as examples for a concept learning program. The concept learning program generates a hypothesis determining why candidate solutions in the H-group perform better than those in L-group. Such a hypothesis is then instantiated to generate new candidate solutions which are likely to be high-performing because they satisfy the hypothesis description. LEM3, the newest implementation of the learnable evolution model, uses the AQ21 rule learning program to induce hypotheses differentiating H- and L-groups. In particular, AQ21 learns rules in the form (1), where $PREMISE$ and $CONSEQUENT$ are conjunctions of attributional conditions (Wojtusiak et al., 2007). In LEM

the $CONSEQUENT$ is always [Group=H], indicating that descriptions of high-performing candidate solutions are learned.

$$CONSEQUENT \Leftarrow PREMISE \tag{1}$$

Instantiation of rules in the form (1) is done by sequentially assigning values of their conditions. Because the rules may not include all attributes used to define the optimization problem, values of the not included attributes are taken from selected high-performing candidate solutions, or from the entire attributes' domains.

3 Constructive Induction

The problem *representation space*, also known as *search space*, is the set of all possible problem solutions. Designing suitable representation space for a given optimization or learning problem is one of the most important and challenging tasks. This section proposes an automated method for improving representation spaces in the learnable evolution model. This method is built upon those previously developed in the field of machine learning, in which constructive induction has been introduced.

The original representation space provided to a machine learning, data mining, or evolutionary computation system may be inadequate for performing the desired task for concept learning, pattern discovery, optimization, etc. Constructive induction (CI) methods automatically create new representation spaces based on the original representations. The new representations allow the determination of relationships that cannot be represented in the original spaces. New representations are created by removing attributes irrelevant to the considered problem, by modifying domains of attributes (for example by discretizing numeric attributes), and by creating new attributes. In this study we concentrate on creation of new attributes, because automated discretization and attribute selection were previously studied in LEM (Michalski, 2000; Wojtusiak and Michalski, 2006). The constructive induction process can be characterized by the function:

$$\Psi : E \rightarrow EC \tag{2}$$

where E is the original representation space and EC is the modified representation space. When considering automatic improvement of representation space in the learnable evolution model, two interrelated problems need to be investigated:

- how to modify representation spaces, i.e., find the function Ψ; and
- how to create new candidate solutions from hypotheses learned in the modified spaces, i.e., find or approximate the function Ψ^{-1}.

Each candidate solution needs to be stored in the original space in order to evaluate its fitness and constraints, thus there must exist an inverse transformation Ψ^{-1}. The transformation, however, do not need to be explicit and can be realized by the instantiation operator. Application of learning and instantiation operators equipped with constructive induction is done following four steps:

- Identify groups of high- and low-performing candidate solutions

- Find the best representation space for distinguishing the groups
- Apply machine learning to distinguish between the groups
- Instantiate the learned hypothesis into the original space.

The first step is identical to one in LEM not equipped with constructive induction. The third step applies machine learning in already modified representation spaces, thus no additional changes are needed. The key parts of using constructive induction in the learnable model are in the steps two and four. They are based on the idea that all changes to representation spaces should be such, that there is an efficient method for creating new candidate solutions from hypotheses learned in the modified spaces.

3.1 Improving Representation Spaces

The problem of searching for the best representation space has been investigated in the field of machine learning by numerous researchers, e.g., (Markovich and Rosenstein, 2002; Krawiec, 2002; Muharram and Smith, 2005). This study adapts *data-driven constructive induction* (DCI) to search for the best representation space by analyzing the data and the current representation (Bloedorn and Michalski, 1998). This is possible because machine learning is used to induce hypotheses discriminating between high- and low-performing candidate solutions.

The most important feature of constructive induction is its ability to create new attributes. These new attributes are designed to more adequately capture high-performing candidate solutions, and help in distinguishing high- and low-performing solutions with simple well performing hypotheses. These new attributes can be in the form of equations involving numeric attributes, and special forms involving symbolic attributes (e.g., count, equality). Unlike in concept learning, where the goal is to learn simple and well-performing hypotheses, in LEM an additional requirement needs to be satisfied. The requirement is that the learned hypotheses must be in such a form that the instantiation process is efficient. The following paragraphs describe an algorithm for constructing new attributes in a special instantiable form (3). The form is defined by the existence of a sequence of attributes, $ATT_1, \ldots ATT_n$ such that the expression $EXPR_1$ does not include $ATT_1, \ldots ATT_n$, $EXPR_2$ does not include $ATT_2, \ldots ATT_n$, but may include ATT_1, and so on. Finally, $EXPR_n$ does not include ATT_n, but may include $ATT_1, \ldots ATT_{n-1}$. Such a sequence guarantees that the conditions can be efficiently evaluated during the instantiation as discussed in the next section.

$$[ATT_1 \pm EXPR_1] \wedge [ATT_2 \pm EXPR_2] \wedge \ldots \wedge [ATT_n \pm EXPR_n] \qquad (3)$$

The algorithm for creating attributes in the form (3) is presented in Figure 2. It starts with attributes from the current representation space. The representation may be original, or it may be already modified. New attributes are built from expressions by adding available existing attributes to expressions using standard arithmetic operators ($+$, $-$, $*$, $/$), and by applying user-specified functions such as $sin(X)$, $cos(X)$, $sqrt(X)$, etc. For symbolic attributes, program may construct *count*, and *equality* attributes. Each newly created attribute is evaluated and if its

```
Attributes = Current attributes
For depth = 1 to maxdepth
  For each attribute Att in Attributes
    Add Att to the list of used attributes
    For each attribute Att1 in Attributes and not in the used attributes
      Create attributes using operators +, -, *, / on Att and Att1
      Evaluate quality of new attributes
    Create attributes using requested functions on Att
    Evaluate quality of new attributes and add to Attributes list if
      the quality is above a given threshold
Remove all attributes with quality below threshold
Remove worst quality attributes so only k best are kept
```

FIGURE 2: Pseudocode of the algorithm for constructing instantiable attributes.

quality is above a given threshold, it is added to the list of attributes. Finally, new representation space is assembled using at most the k highest quality attributes, where $k << P$ is a user-defined parameter significantly smaller than the number of high-performing examples in the current population (P).

The constructed attributes are used by the AQ21 learning module as standard numeric attributes, as their values can be computed for each candidate solution prior to execution of the learning module. Because of the imposed form of these attributes, there are transformations that cannot be represented. For example (4) cannot be created by the algorithm.

$$X^2 + X + Y^2 + Y \tag{4}$$

The algorithm presented above tends to be inefficient or even impossible to apply for representation spaces with very large numbers of attributes. For example, if there are 100 original attributes and the program seeks only combinations of pairs of attributes, there are nearly $40,000$ potential new attributes for which the quality measure needs to be computed. Even if the method implements additional heuristics, such as checking monotonicity of the attributes, checking duplicate attributes, etc., the number of possible combinations is still very large. In concept learning in which exploration of these possibilities is usually done only once, it still may be possible to do. In LEM, however, such a process is repeated many times at different stages of evolution, thus it may be computationally too expensive for practical applications. One possibility of solving this problem may be to combine only high quality attributes. Such attributes are more likely to produce new high quality attributes than combinations of low quality ones.

3.2 Instantiation in Modified Representation Spaces

Creation of new candidate solutions in LEM's learning mode is realized by instantiation of learned hypotheses. Newly created candidate solutions need to be stored in the original representation space, in order to allow evaluation of their fitness and constraints. The process of creating solutions that satisfy a given rule

```
For each rule
  Compute the number of candidate solutions to be generated
  New Solutions = NULL
  Compute order of attributes based on their interdependency
  While size of New Solutions < number of solutions to be generated
    For each attribute in the computed order
      Compute all expressions with the attribute on the left side
      Compute condition for the attribute
      If condition is empty
        If the total number of tries exceeds a threshold
          Stop instantiating the current rule
        Jump to one of attributes before based on backtracking
      Else
        Instantiate the condition
    Add the candidate solution to New Solutions
```

FIGURE 3: Top level instantiation algorithm for constrained rules.

may be very time consuming, thus, the approach taken in this research is to allow constructed attributes only in the form (3) that can be efficiently evaluated.

The pseudocode in Figure 3 describes the algorithm for generating new candidate solutions from a given hypothesis learned in a modified representation space. It is assumed that all constructed attributes are in the form (3), meaning that they can be sequentially evaluated while generating a new candidate solution. Whenever a condition cannot be satisfied because of previous selections of attribute values, a simple backtracking algorithm is used.

In the algorithm presented in Figure 3, the number of new candidate solutions generated for each constrained rule is proportional to its positive coverage (number of covered high-performing solutions). At this point the set of already created new candidate solutions is empty. It is important to correctly order conditions (attributes) in the current rule, so all attributes needed to instantiate an attribute are already instantiated. After the list of attributes to be instantiated is prepared, the algorithm creates new candidate solutions until their total number reaches the desired level, or until the total number of unsuccessful conditions' instantiations exceeds a given threshold.

Each new candidate solution is created by selecting values of attributes in the defined order. If for an attribute being considered there is no condition in the rule, the program uses one of methods already known in the learnable evolution model. These include taking a value from a randomly selected high-performing candidate solution that satisfies the rule, or selecting a value from the entire attribute's domain. For attributes included in the rule, the final condition to be instantiated is computed based on previous and currently instantiated conditions. For example, the hypothesis may include a rule with conditions $[Height \geq Width + Length - 2]$ and $[Height \leq 50]$. To select a value of the attribute *Height*, it is necessary to compute conjunction of the two conditions: $[Height \geq Width + Length - 2] \wedge [Height \leq 50]$. Because it is assumed that all constructed attributes are in the form (3), the values of attributes $Length$ and $Width$ are already selected,

and therefore all expressions in the conditions can be evaluated. For example, if $Width = 30$ and $Length = 10$, we have $[Height \geq 38] \wedge [Height \leq 50]$, which can be reformulated as $[Height = 38..50]$. The final condition consists of a range that can be instantiated by selecting a value from between 38 and 50.

An important question arises what should be done in a situation if a condition cannot be satisfied because of the choice of previously selected values of attributes needed to evaluate a condition. To illustrate this situation, suppose that in the previous example the new candidate solution was assigned values $Width = 31$ and $Length = 25$. In such a case, conditions for the attribute *Height* are $[Height \geq 54] \wedge [Height \leq 50]$ which is an empty interval. To solve this problem the program needs to reinstantiate the attribute *Length* and compute the conditions for *Height* again. Because values of attributes that satisfy conditions are assigned randomly (according to some distribution) it may happen that the condition is not satisfied again and the operation needs to be repeated. This process is continued no longer than a specified number of times, when the program tries to reinstantiate a previous attribute (in this case *Width*) also for a specified number of times. This process repeats until the condition is satisfied or a total number of tries (usually very large) is exceeded and the rule is ignored. It is always possible to find a combination of values that satisfy all conditions (Wojtusiak, 2007), but the process may take a large number of trials. However, the backtracking algorithm mentioned above tends to be efficient and in the performed experiments all conditions are satisfied relatively fast.

In addition to the *instantiation* algorithm described above, we implemented also a simple *rejection* method for instantiating rules with constructed attributes. The method creates new candidate solutions by ignoring all conditions with constructed attributes. Then, the newly created candidate solutions are checked against all conditions with constructed attributes. If some of the conditions are violated, such solutions are rejected and the process is repeated until a sufficient number of candidate solutions satisfying all conditions is created (Wojtusiak, 2007).

4 Experimental Results

There are many possible methods of reporting results of testing optimization algorithms. The most common are to report the best result obtained after a given number of fitness evaluations (or generations), or to report the number of fitness evaluations needed to achieve a given solution. Other possibilities include computation time needed to achieve a given solution on a given computer, the number of fitness evaluations needed to achieve a given improvement, and so forth. In the latter method the results can be reported for $\delta - close$ solutions that are characterized by a normalized distance from the optimal solution (Michalski, 2000; Wojtusiak and Michalski, 2006). This method can be used to evaluate performance on test problems to which solutions are known. The δ-close solution, s, is a solution for which function $\delta(s)$, defined as (5) reaches an assumed $\delta - target$ value, where $init$ is the evaluation (fitness value) of the best solution in the initial population, opt is the optimal value, and $v(s)$ is the evaluation of the solution s. Such a measure works for both maximization and minimization problems.

$$\delta(s) = \frac{|opt - v(s)|}{|opt - init|} \tag{5}$$

This definition of $\delta - close$ solution suggests two possible ways of analyzing performance of evolutionary computation methods. First, one may consider the problem of how many fitness function evaluations are needed to achieve a given $\delta = k$ by the best candidate solution in the population, denoted as $FE(\delta = k)$, where k is a number between 0 and 1. Secondly, one may consider the problem of finding $\delta(s)$ after given number of fitness evaluations or generations, where s is the best candidate solution found after a given number of fitness function evaluations or generations. The latter is the primary way of reporting results in this study. For example, if the fitness value of the best candidate solution in the initial population is 100 and during the process of minimization the program achieved value 0.1, and the optimal value is 0 then $\delta = 0.001$, indicating that program found a solution within 0.1% distance from the optimal solution, normalized by the fitness value of the best candidate solution in the initial population.

To compare performance of the learnable evolution model with and without constructive induction, it has been applied to optimizing well known benchmark problems, namely the *Rastrigin*, *Griewangk*, *Rosenbrock*, *Sphere*, and *Step* functions. For descriptions of the problems see, for example, (Wojtusiak, 2007). Because of the very large number of performed experiments (about 700 combinations of tested parameter settings, each repeated 10 times), the presented results are averaged. They are grouped by numbers of attributes, optimization problems, and methods for improving representation spaces. The results are presented in terms of the average $\delta(s)$ values obtained after 100 generations of LEM3's evolutions. The results grouped by the number of attributes are presented in Table 1, and the results grouped by optimization problem are presented in Table 2.

LEM3 equipped with automated improvement of representation spaces by constructive induction gave on average better results than LEM3 without constructive induction. The results are very strong for problems with 2 and 4 attributes. However, it can be observed from results in the Table 1 that the advantage of LEM3 equipped with constructive induction tends to diminish with increasing numbers of attributes. The average result for 100 variables is better for LEM3 without constructive induction than for LEM3 that improves representation spaces. This fact can be attributed to the method of improving the representation space, which is unable to create new attributes that include many attributes from the original spaces. Thus, the method is unable to capture complex numerical relationships involving many attributes. Similarly, Table 2 suggests that the automated improvement of representation spaces significantly improves results for most tested problems, but may not be appropriate for others. For the *step* function LEM3 without constructive induction gives better results than one improving representation space. This can be explained by the fact that to optimize this function it is sufficient to optimize all attributes independently, and construction of new attributes only adds additional complexity to the problem. Further investigation of characteristics of problems to which LEM with constructive induction should be applied is needed to confirm this hypothesis.

TABLE 1: Mean and standard deviation $\delta(s)$ values after 100 generations for different numbers of attributes averaged for Griewangk, Rastrigin, Rosenbrock, and Sphere functions.

Number of Attributes	**No CI**	**CI with Instantiation**	**CI with Rejection**
2	0.021942 ± 0.01	0.01 ± 0.01	0.011148 ± 0.007
4	0.01 ± 0.004	0.009134 ± 0.005	0.010078 ± 0.003
10	0.050666 ± 0.017	0.048895 ± 0.026	0.058637 ± 0.02
50	0.213508 ± 0.08	0.208315 ± 0.087	0.215043 ± 0.089
100	0.277329 ± 0.085	0.295008 ± 0.074	0.280424 ± 0.083
average	0.115112 ± 0.0399	0.116198 ± 0.04	0.115066 ± 0.041

TABLE 2: Mean and standard deviation $\delta(s)$ values after 100 generations for the Griewangk, Rastrigin, Rosenbrock, Sphere, and Step functions averaged for 2, 4, 10, 50 and 100 attributes.

Function	**No CI**	**CI with Instantiation**	**CI with Rejection**
Griewangk	0.260521 ± 0.0798	0.270987 ± 0.076	0.260965 ± 0.087
Rastrigin	0.134349 ± 0.0428	0.130729 ± 0.0466	0.133836 ± 0.038
Rosenbrock	0.019625 ± 0.013	0.018622 ± 0.014	0.018889 ± 0.013
Sphere	0.045952 ± 0.024	0.044452 ± 0.026	0.046573 ± 0.026
Step	0.09915 ± 0.037	0.104438 ± 0.053	0.11328 ± 0.0427

5 Conclusion

This paper briefly introduced the idea of using data-driven constructive induction to improve representation spaces in the learnable evolution model. An important result is that the presented method for modifying representation spaces consistently improved LEM3's performance. This improvement, however, tends to diminish with growing numbers of attributes. This may be due to use of a DCI-based method for automated improvement of representation space that is unable to construct attributes involving many original attributes - the search space is too large. Two potential solutions to this problem include using a different method for modifying representations, and using background knowledge to narrow the attribute search space. A method for modifying the representation space should be able to construct new attributes that include many original attributes. For example, principal component analysis, which is widely used in statistics (Gentle, 2002), constructs new dimensions that may include all original attributes at the same time. Another possible solution is to use background knowledge by suggesting possible transformations of the representation space in the form of advices in order to narrow the search for new attributes. Given such advices the program will be able to immediately construct correct attributes without trying all possible combinations.

6 Acknowledgments

The author would like to thank Professor Ryszard S. Michalski who supervised work on the presented research. The development of the LEM3 and AQ21 systems was supported in part by the National Science Foundation grants IIS 9906858 and IIS 0097476. The findings and opinions expressed here are those of the author, and do not necessarily reflect those of the above sponsoring organization.

References

Peter J. Angeline and Jordan B. Pollack (1994), Coevolving High-Level Representations, In *Proceedings of the Third Workshop on Artificial Life*, pp 55-71.

Eric Bloedorn and Ryszard S. Michalski (1998), Data-Driven Constructive Induction, IEEE Intelligent Systems, Special issue on Feature Transformation and Subset Selection, pp. 30-37.

James E. Gentle (2002), Elements of Computational Statistics, Springer.

Krzysztof Krawiec (2002), Genetic Programming-based Construction of Features for Machine Learning and Knowledge Discovery Tasks, Genetic Programming and Evolvable Machines, 3, pp. 329-343.

Shaul Markovich and Dan Rosenstein (2002), Feature Generation Using General Constructor Functions, Machine Learning, 49, pp. 59-98.

Ryszard S. Michalski (1998), Learnable Evolution: Combining Symbolic and Evolutionary Learning, In *Porceedings of the Fourth International Workshop on Multistrategy Learning (MSL'98)*, Desenzano del Garda, Italy, pp. 14-20.

Ryszard S. Michalski (2000), LEARNABLE EVOLUTION MODEL: Evolutionary Processes Guided by Machine Learning, *Machine Learning*, 38, pp. 9-40.

Mohammed Muharram and George D. Smith (2005), Evolutionary Constructive Induction, IEEE Transactions on Knowledge and Data Engineering, 17, pp. 1518-1528.

Joseph Reisinger and Risto Miikkulainen (2007), Acquiring Evolvability Through Adaptive Representations, In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, pp. 1045 - 1052.

Janusz Wojtusiak and Ryszard S. Michalski (2006), The LEM3 Implementation of Learnable Evolution Model and Its Testing on Complex Function Optimization Problems, In *Proceedings of Genetic and Evolutionary Computation Conference, GECCO 2006*, Seattle, WA, July 8-12.

Janusz Wojtusiak, Ryszard S. Michalski, Kenneth A. Kaufman, and Jaroslaw Pietrzykowski (2006), The AQ21 Natural Induction Program for Pattern Discovery: Initial Version and its Novel Features, In *Proceedings of The 18th IEEE International Conference on Tools with Artificial Intelligence*, Washington D.C.

Janusz Wojtusiak (2007), Handling Constrained Optimization Problems and Using Constructive Induction to Improve Representation Spaces in Learnable Evolution Model, *Ph.D. Dissertation*, College of Science, George Mason University, Fairfax, VA.