Error Function Learning with Interpretable Compositional Networks for Constraint-Based Local Search

Florian Richoux
AIST, Japan
florian.richoux@aist.go.jp

Jean-François Baffier
The University of Tokyo, Japan
jf@baffier.fr

***Abstract*—In Constraint Programming, some Constraint-Based Local Search algorithms exploit error functions, a finer representation of constraints than the usual one. However, this makes problem modeling significantly harder, since providing a set of error functions is not always easy. Here, we propose a method to automatically learn an error function corresponding to a constraint. Our method learns error functions upon a variant of neural networks we named Interpretable Compositional Networks, allowing us to get interpretable results, unlike regular artificial neural networks. Experiments show that our method can learn, over small-dimensional spaces or incomplete spaces, functions that scale on high-dimensional spaces.**

I. Learning ICN models

The main result of this work is to propose a method to automatically learn an error function representing a constraint, making easier the modeling of Error Function Satisfaction/Optimization Problems [RB21]. We are tackling a regression problem since the goal is to find a function that outputs a target value. For this work, the target value is the Hamming distance between a given variable assignment and its closest solution within the constraint assignment space.

To model error functions, we propose a variant of neural networks retaking two principles from Compositional Pattern-Producing Networks [Sta07]: 1., having neurons containing one activation function among many possible ones, and 2., being able to handle inputs in a size-independent fashion. Due to their interpretable nature, we named our variant **Interpretable Compositional Networks** (ICN). For this work, ICNs are composed of four layers (see Figure 1), each of them having a specific purpose and themselves composed of neurons applying a unique activation function each, called elementary operation. All neurons from a layer are linked to all neurons from the next layer. The weight on each link is purely binary: its value is either 0 or 1. This restriction is crucial to obtain interpretable functions. A weight between neurons n_1 and n_2 with the value 1 means that the neuron n_2 from layer $l+1$ takes as input the output of the neuron n_1 from layer l. Weight with the value 0 means that n_2 discards the output of n_1.

Here is our method workflow in 4 points:

1. Users provide a regular constraint network $\langle V, D, C\rangle$ where V is the set of variables, D their domain, and C a set of concepts representing constraints.

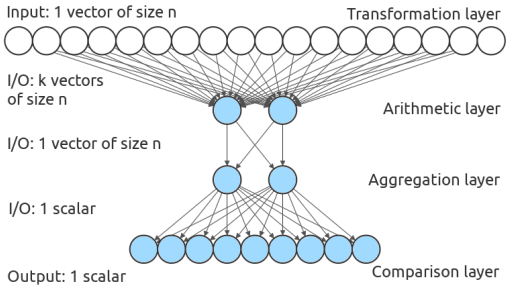

Figure 1: Our 4-layer network. Layers with blue neurons have mutually exclusive operations.

2. We generate for each constraint concept c its ICN input space X, which is either a complete or incomplete constraint assignment space. Those input spaces are our training sets. If the space is complete, then the Hamming distance of each assignment can be pre-computed before learning our ICN model. Otherwise, we randomly draw assignments to build an incomplete space and pre-compute an approximation of the Hamming distance for each drawn assignment.

3. We learn the weights of our ICN model in a supervised fashion, with the following loss function:

$$\text{loss} = \sum_{\vec{x}\in X} (|ICN(\vec{x}) - Hamming(\vec{x})|) + R(ICN) \quad (1)$$

where X is the constraint assignment space, ICN $(\vec{x})$ the output of the ICN model giving $\vec{x} \in X$ as an input, Hamming$(\vec{x})$ the pre-computed Hamming distance of $\vec{x}$ (only approximated if X is incomplete), and R(ICN) is a regularization between 0 and 0.9 to favor short ICNs, *i.e.*, with as few elementary operations as possible, such that $R(ICN) = 0.9 \times \frac{\text{Number of selected elementary operations}}{\text{Maximal number of elementary operations}}$.

4. We have hold-out test sets of assignments from larger dimensions to evaluate the quality of our learned error functions.

Like any neural network, learning an error function through an ICN boils down to learning the value of its weights. Many of our elementary operations are discrete, therefore are not differentiable. Then, we cannot use a back-propagation

algorithm to learn the ICN's weights. This is why we use a genetic algorithm for this task.

Since our weights are binary, we represent individuals of our genetic algorithm by a binary vector of size 29, each bit corresponding to one operation in the four layers. Since arithmetic and aggregation layers contain only two mutually exclusive operations, these operations are represented by one bit for each layer. For the transformation and comparison layers, the i-th bit set to 1 means their i-th operation is selected to be part of the error function.

We randomly generate an initial population of 160 individuals, check and fix them if they do not satisfy mutually exclusive constraints. Then, we run the genetic algorithm to produce at most 800 generations before outputting its best individual according to our fitness function.

Our genetic algorithm is rather simple: The **fitness function** is the loss function of our supervised learning depicted by Equation 1. **Selection** is made by a tournament selection between 2 individuals. **Variation** is done by a one-point crossover operation and a one-flip mutation operation, both crafted to always produce new individuals verifying the mutually exclusive constraint of the comparison layer. The crossover rate is fixed at 0.4, and exactly one bit is mutated for each selected individual with a mutation rate of 1. **Replacement** is done by an elitist merge, keeping 17% of the best individuals from the old generation into the new one, and a deterministic tournament truncates the new population to 160 individuals. The algorithm stops before reaching 800 generations if no improvements have been done in the last 50 generations. We use the framework EVOLVING OBJECTS [KMRS02] to code our genetic algorithm.

Our hyperparameters, *i.e.*, the population size, the maximal number of generations, the number of steady generations before early stop, the crossover, mutation and replacement rates, and the size of tournaments have been chosen using ParamILS [HHLBS09], trained one week on one CPU over a large range of values for each hyperparameter.

II. EXPERIMENTS

To show the versatility of our method, we tested it on five very different constraints: AllDifferent, Ordered, LinearSum, NoOverlap1D, and Minimum. According to XCSP specifications (Boussemart et al. [BLAP16], see also http://xcsp.org/specifications), those global constraints belong to four different families: Comparison (AllDifferent and Ordered), Counting/Summing (LinearSum), Packing/Scheduling (NoOverlap1D) and Connection (Minimum). Again according to XCSP specifications, these five constraints are among the twenty most popular and common constraints.

The source code and experimental setups are accessible on GitHub.[1]

Table I presents the mean error and the normalized mean error of the most frequently learned error function for each constraint type, over test sets of 20,000 assignments sampled from huge spaces of 10^{100} assignments. Its second column shows test errors for error functions learned over small complete spaces (about $500 \sim 625$ assignments) and the third one for error functions learned over incomplete spaces (20,000 drawn assignments in spaces of size about 10^{12}).

Constraints	complete		incomplete	
	mean	norm.	mean	norm.
all_different-100-100	0	0	5.2821	0.0528
linear_sum-100-100-5279	0.0379	0.0003	0.0379	0.0003
minimum-100-100-30	0	0	0	0
no_overlap-10-35-3	2.6863	0.2686	2.0257	0.2025
ordered-12-18	1.2745	0.1062	0.6054	0.0504

Table I: Mean test error over 20,000 assignments in high dimensions of most frequently learned error functions.

The mean error is the total error on a test set divided by the size of the test set (20,000 for each test set). Therefore, a mean error of 5 for instance means that, on average, the error function computes a Hamming distance off by 5 variables regarding the expected Hamming distance. However, it is not the same thing to be off by 5 variables on instances with 10 variables or with 100 ones. Thus, Table I also contains a normalized mean error corresponding to the mean error divided by the number of variables in the instance.

The perfect score of 0 for AllDifferent and Minimum shows that our system has been able to learn the exact Hamming distance over a small constraint assignment space of 625 assignments. For LinearSum, the error function only has a total error of 758 over 20,000 assignments, giving a mean error of 0.0379 over each assignment. Since our test set instance for LinearSum is over 100 variables, the normalized mean error is 3.79×10^{-4}.

Ordered and NoOverlap1D do not show such good results. For Ordered, a mean error of 1.2745 on assignments with 12 variables is still honorable: it means that on average, the difference between the expected and estimated Hamming distance over 10 variables is about one variable. Put differently, there is a mean error of 0.1062 per variable in the test instance.

However, the mean error of 2.6863 for NoOverlap1D, considering the constraint instance has 10 variables, is not so good: this leads to a normalized mean error of 0.2686, which starts to be significative (about one error every 4 variables). NoOverlap1D is certainly the most intrinsically combinatorial over our 5 constraints, partly explaining why it is harder to learn a correct error function for it.

[1] https://github.com/richoux/LearningErrorFunctions/releases/tag/1.0

REFERENCES

[BLAP16] Frederic Boussemart, Christophe Lecoutre, Gilles Audemard, and Cédric Piette. XCSP3: An Integrated Format for Benchmarking Combinatorial Constrained Problems. *arXiv e-prints*, abs/1611.03398:1–238, 2016.

[HHLBS09] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

[KMRS02] Maarten Keijzer, J. J. Merelo, G. Romero, and M. Schoenauer. Evolving Objects: A General Purpose Evolutionary Computation Library. *Artificial Evolution*, 2310:829–888, 2002.

[RB21] Florian Richoux and Jean-François Baffier. Error function learning with interpretable compositional networks for constraint-based local search. *arXiv e-prints*, abs/2002.09811:1–11, 2021.

[Sta07] Kenneth O. Stanley. Compositional Pattern Producing Networks: A Novel Abstraction of Development. *Genetic Programming and Evolvable Machines*, 8(2):131–162, 2007.