juliacon

CompositionalNetworks.jl: a scaling glass-box neural network to learn combinatorial functions

Jean-François BAFFIER[1], Khalil CHRIT[2], Florian RICHOUX[34], Pedro PATINHO[2], and Salvador ABREU[2]

[1]IIJ, Japan
[2]NOVA-LINCS, University of Évora, Portugal
[3]AIST, Japan
[4]JFLI, CNRS, Japan

ABSTRACT

Interpretable Compositional Networks (ICNs) are a neural network variant for combinatorial function learning that allows the user to obtain interpretable results, unlike ordinary artificial neural networks. An ICN outputs a composition of functions that scale with the size of the input, allowing a learning phase on relatively small spaces. CompositionalNetworks.jl is a pure Julia package that exploits the language's meta-programming, parallelism and multiple dispatch features to produce learned compositions in mathematical and programming languages such as Julia, C or C++.

Keywords

Julia Language, Constraint Programming, Local Search, Metaheuristics, Neural Network, Metaprogramming, Scalable Machine Learning, Glass-Box Algorithm

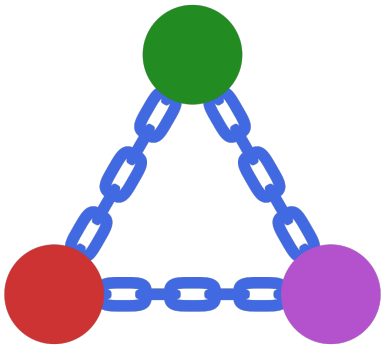


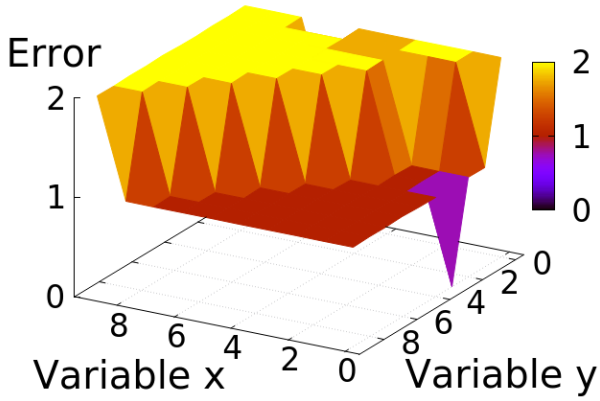
Fig. 1: Logo of the JuliaConstraints organization on GitHub that hosts, among other things, the CompositionalNetworks.jl package.

1. Introduction

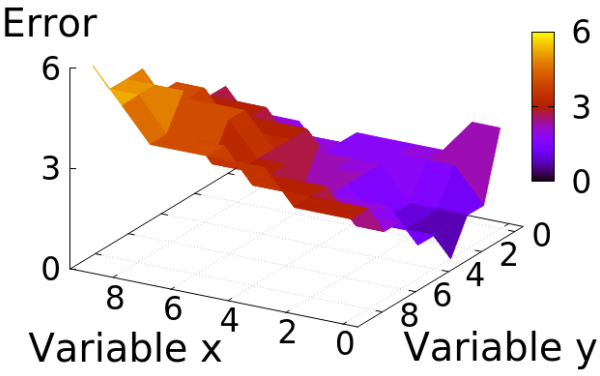
The discipline of combinatorial optimization consists in finding an optimal configuration of elements within a finite set. Such a set is usually subject to constraints that can be represented by functions that are often highly combinatorial. These constraints are mostly formulated as concepts, boolean functions indicating whether each constraint is respected or not. A solution (sometimes called a satisfactory solution, depending on the domain) is a configuration that respects all the constraints.
Different domains such as operational research, constraint programming, metaheuristics, propose methods to find (satisfactory or optimal) solutions. Among these methods, some can or could benefit from a finer granularity in the evaluation of the impact of each constraint on a given configuration.
In [5], we introduced Interpretable Compositional Networks (ICNs), a neural network variant for combinatorial function learning that allows the user to obtain interpretable results, unlike ordinary artificial neural networks. An ICN outputs a composition of functions that scale with the size of the input, allowing a learning phase on relatively small spaces.
This work, consists in a library that implements ICNs in the Julia programming language [2]. Although we present a direct application of ICNs in the following subsection, this neural network framework is simple to extend to learn other combinatorial and non-combinatorial functions.

1.1 Application to Constraint Programming

Constraint Satisfaction Problem (CSP) and Constrained Optimization Problem (COP) are constraint-based problems where constraints can be seen as predicates (*concepts*) allowing or forbidding some combinations of variable assignments. Such a formulation corresponds well to the so-called complete solution methods which guarantee an exploration of all solutions. Unfortunately, due to the highly combinatorial nature of some problems, these methods cannot always converge in a reasonable time.
On the other hand, Constraint-Based Local Search (CBLS) is a family of metaheuristics in which the neighborhood is constructed on the basis of an *error function*, itself a quantitative representation of how far the current configuration is from an admissible solution. We refer to the corresponding problems as Error Function Satisfaction Problem (EFSP) and Error Function Optimization Problem (EFOP). In the case of CBLS, this is computed as a function of the constraints which are not currently satisfied. Error functions may be derived from the constraint satisfaction problem structure, hand-coded, automatically acquired by some machine learning process, or constructed as a combination of these methods. It should be noted that, the best performing systems resort to hand-tuned error functions, as witness [4].
As illustrated by Figure 2, a well-designed error function helps in converging faster towards better-quality solutions. It does, however, introduce an additional layer of complexity to the user.



(a) CSP landscape



(b) EFSP landscape

Fig. 2: Comparison of a Constraint Satisfaction Problem (CSP) and an Error Function Satisfaction Problem (EFSP) landscapes. The finer scale in the error heuristic of the EFSP leads to a better convergence rate.

1.2 Interpretable Compositional Networks (ICN)

In [5], we proposed a neural network inspired by Compositional Pattern-Producing Networks (CPPN) to learn (highly) combinatorial functions as non-linear combinations of elementary operations. CPPNs [7] are themselves a variant of artificial neural networks. While neurons in regular neural networks usually contain sigmoid-like functions only (such as ReLU, *i.e.* Rectified Linear Unit), CPPN's neurons can contain many other kinds of functions: sigmoids, Gaussians, trigonometric functions, and linear functions among others. CPPNs are often used to generate 2D or 3D images by applying the function modeled by a CPPN giving each pixel individually as input, instead of considering all pixels at once. This simple trick allows the learned CPPN model to produce images of any resolution.
We propose our variant by taking these two principles from CPPN: having neurons containing one operation among many possible ones, and handling inputs in a size-independent fashion. Due to their interpretable nature, we named our variant **Interpretable Compositional Networks** (ICN).
Although ICNs are not limited to learning function for EFSP/EFOP, the original structure was designed to learn compositions weighted in accordance to the hamming metric [5]. The hamming distance between a configuration and its closest solution is a meaningful indicator of the number of variables needed to be changed. Manhattan and other variants of minkowski comparison are other example of possible metrics.

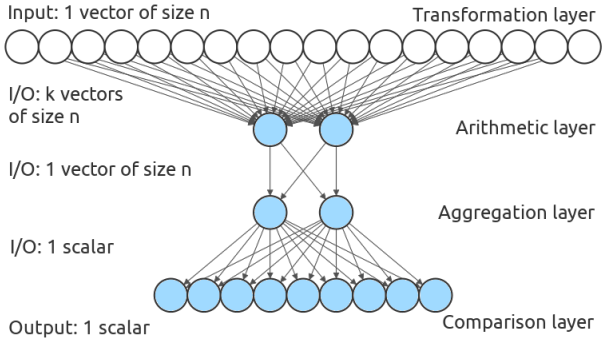


Fig. 3: Scheme of the basic 4-layers ICN model used in EFSP/EFOP. Layers with blue neurons have mutually exclusive operations. the *transformation* layer increases the input dimension, the *arithmetic* and *aggregation* layers reduces it, and the *comparison* layer leaves it untouched. The figure is taken from [5].

An ICN is made of several layers of (possibly exclusive) operations such that the first layers accept a vector as input and the last layer returns a single numerical value. Generally, layers alter there input in three possible ways: *increment*, *decrement*, or *conservation* of the dimension of the input.
In the context of EFSP/EFOP, the output should be non-negative if the constraint is violated and equal to 0 otherwise. As shown by Figure 3, our ICNs are composed of four layers, each of them having a specific purpose and themselves composed of neurons applying a unique operation each. 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. We refer the reader to [5] for a more comprehensive definition.

1.3 A first C++ library

In [5] we introduced the concept of ICN and a first implementation for Constraint Programming as a C++ library[1]. The results were evaluated through the GHOST C++ solver [6] and serve as a proof of concept that most of the models give scalable functions, and remain fairly effective using incomplete training sets.
As mentioned in Section 4, CompositionalNetworks.jl generates code for a direct use in Julia, but also exports usable code for both the GHOST(C++) and AdaptiveSearch (C)[2]. The code exported in C++, that is the *operations* of each neuron, is strongly inspired by our C++ library.

2. The example of the JuliaConstraints framework

The CompositionalNetworks.jl package is part of the JuliaConstraints GitHub organization that proposes a first collection of Julia packages for Constraint-Based Local Search (CBLS), a subfield of Constraint Programming.
The main goal of this framework is to provide a set of high level semi-automatic tools which strike a good compromise between efficiency and ease of modeling.

[1] https://github.com/richoux/LearningUtilityFunctions
[2] This feature is a WIP at the moment this article is submitted, but is expected to be completed in the coming weeks.

2.1 History of JuliaConstraints

The LocalSearchSolvers.jl package is a Constraint-Based Local Search framework started in Fall 2020 and inspired by other CBLS solvers such as GHOST (C++) and AdaptiveSearch (C) that allows users to tune their own solver.
During the development of LocalSearchSolvers.jl, we decided to split the code and functionality of the original framework into several independent packages for ease of maintenance and in hopes of providing common tools for other constraint programming packages in Julia.
During 2021, these tools were collected into the JuliaConstraints ecosystem as follows

—ConstraintDomains.jl: creation of discrete, continuous, and arbitrary domain (beta status)
—Constraints.jl: generation of constraints from a boolean concept or an error function. Also list usual constraints and their properties (beta status)
—CompositionalNetworks.jl: stable
—LocalSearchSolvers.jl: A CBLS framework in Julia with built-in parallel and distributed scalability
—CBLS.jl: A MOI/JuMP wrapper for LocalSearchSolvers
—ConstraintModels.jl: list of CP models for CBLS.jl and LocalSearchSolvers.jl

The global state of the Constraint Programming ecosystem is presented in Figure 4. Beside the CBLS framework of JuliaConstraints, other solvers are complete search methods, such as ConstraintSolver.jl[3], CPLEXCP.jl[4], and SeaPearl.jl [3].

2.2 A simple example with `CBLS.jl`

A Magic Square of order n is composed of n^2 distinct values ranging from 1 to n^2 layed out as a square array X. These values need to be arranged such that the sums of each diagonal, row and column be equal to the same value, the magic sum Σ.

$$\sum_{j=1}^{n} X_{ij} = \sum_{i=1}^{n} X_{ij} = \sum_{j=1}^{n} X_{ii} = \sum_{j=1}^{n} X_{i,n-i+1} = \frac{n(n^2+1)}{2} = \Sigma$$

In `ConstraintModels.jl`, we use the following code to generate a magic-square instance.

```
# Import CBLS.jl and JuMP.jl
using CBLS, JuMP

function magic_square(n::Integer)
model = Model(CBLS.Optimizer)
N = n^2
Σ = n * (N + 1) / 2
@variable(model, 1 <= X[1:n, 1:n] <= N, Int)
@constraint(model, vec(X) in AllDifferent())
for i in 1:n
  @constraint(model, X[i,:] in Linear(Σ))
  @constraint(model, X[:,i] in Linear(Σ))
end
@constraint(model,
  [X[i,i] for i in 1:n] in Linear(Σ)
)
@constraint(model,
  [X[i,n+1-i] for i in 1:n] in Linear(Σ)
)
  return model, X
end

# Create a magic square instance of size 4x4
m, X = magic_square(4)
# Optimize
optimize!(m)
# Output values
@info values.(X)
```

[3] https://github.com/Wikunia/ConstraintSolver.jl
[4] https://github.com/dourouc05/CPLEXCP.jl

3. A flexible implementation

CompositionalNetworks.jl extends the work of the first C++ library used in [5]. As mentioned in Section 1.3, that library used for prototyping was limited through several aspects: this, this, and that. Obviously, those limitations could have been lifted in the first library. However, we decided to use some features of the Julia language to ease provisioning flexibility and broader features to ICNs.

Block about Julia. test

3.1 Layers of operations

A layer is defined by a non-empty collection of (possibly mutually exclusive) operations. The original ICN is composed of four layers: *transformation* (exclusive), *arithmetic*, *aggregation*, and *comparison*.

```
struct Layer
  functions::LittleDict{Symbol, Function}
  exclusive::Bool
end
```

```
# Transformation defined for a vector
tr_identity(x; param=nothing) = identity(x)

# Transformation defined for the index of a vector
tr_count_eq(i, x; param=nothing) =
  count(y -> x[i] == y, x) - 1
tr_count_eq_param(i, x; param) =
  count(y -> y == x[i] + param, x)

# Generating vectorized versions
lazy(tr_count_eq)
lazy_param(tr_count_eq_param)
```

```
function transformation_layer(param=false)
  transformations = LittleDict{Symbol,Function}(
    :identity => tr_identity,
    :count_eq => tr_count_eq,
    # ...
  )
  if param
    tr_param = LittleDict{Symbol, Function}(
      :count_eq_param => tr_count_eq_param,
      # ...
    )
    transformations = LittleDict(
      union(transformations, tr_param)
    )
  end
  return Layer(transformations, false)
end
```

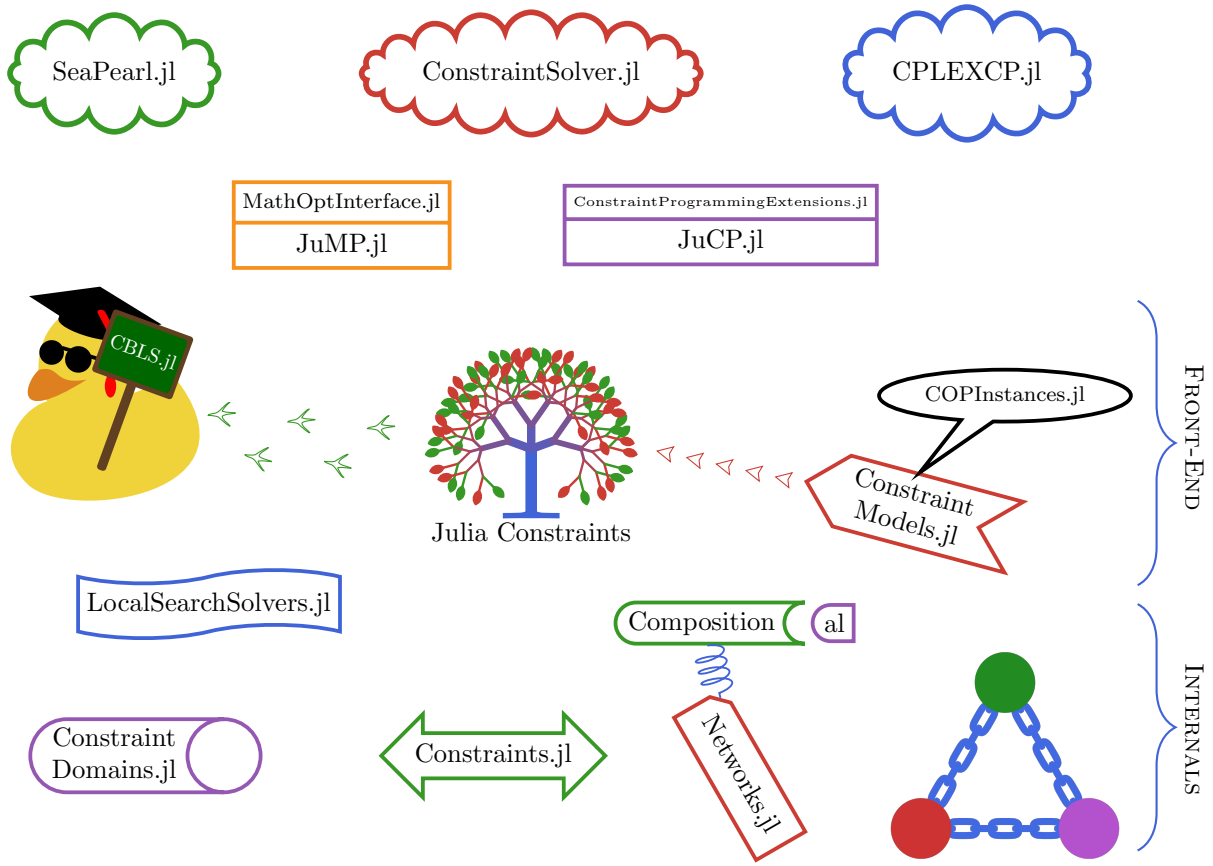


Fig. 4: Overview of the Constraint Programming ecosystem in Julia, including JuliaConstraints. *Front-End* and *Internals* refer to the latest. Note that, at the moment of the writing, `ConstraintProgrammingExtension.jl`, `JuCP.jl`, and `COPInstances.jl` are temporary names.

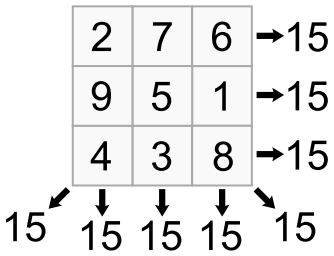


Fig. 5: A solved magic-square of size 3. The *magic sum* is equal to 15. Image from Wikipedia's user PHIDAUEX.

All methods required for the definition of additional layers are all available.

3.2 Composition

We store the composition of an ICN in three forms:

—A collection of `Symbols` per layer of the ICN that allows the generation of code as a string

—The Julia object of type `Function` to apply the composition directly

```
struct Composition{F<:Function}
  code::Dict{Symbol,String}
  f::F
  symbols::Vector{Vector{Symbol}}
end
```

The result output by an ICN is a composition, a mathematical function composed of several basic operations. The `code` function returns the definition of a composition in either a mathematical or programming language. From *v1.0* of CompositionalNetworks.jl, `code` accepts `:maths, :Julia, :C, :Cpp` as values for the *lang* key argument.

```
function code(
  c::Composition, lang=:maths;
  name="composition"
)
  return get!(
    c.code, lang, generate(c, name, Val(lang))
  )
end
```

4. Explore, learn, and compose

It is far from practical to apply the internals of CompositionalNetworks.jl step-by-step. We provide a user-friendly sequence of higher-level actions, *explore*, *learn*, and *compose* that fits most of expected uses of this package.
Each of those actions is semi-automated through default parameters and machine learning over a large collection of benchmarks [1].

4.1 Exploration

```
function explore(
  domains,
  concept,
  param=nothing;
  search=:flexible,
  complete_search_limit=10^6,
  max_samplings=sum(domain_size, domains),
  solutions_limit=floor(Int, sqrt(max_samplings)),
)
  if search == :flexible
    search = sum(domain_size, domains) <
      complete_search_limit ? :complete : :partial
  end
  return explore(Val(search), domains, concept,
    param, solutions_limit, max_samplings
  )
end
```

5. Future challenges

—Add new operations in existing layers
—Add new layers
—Use reinforcement learning
—Cover more metrics
—Apply ICN to other field than EFN for CBLS solvers

6. Problems, constraints, and compositions zoo

The problems modelled, and the compositions extracted in this section are subject to future changes and improvements of the JuMP/JuCP packages. However, the keys ideas are presented here and examples will be updated accordingly within JuliaConstraints in the future.

6.1 A few combinatorial models

```
function mincut(graph, source, sink, interdiction)
  m = JuMP.Model(CBLS.Optimizer)
  n = size(graph, 1)
  separator = n + 1

  @variable(m, 0 <= X[1:separator] <= n, Int)

  @constraint(m,
    [X[source], X[separator], X[sink]] in Ordered()
  )
  @constraint(m, X in AllDifferent())

  obj(x...) = o_mincut(graph, x...; interdiction)
  @objective(m, Min, ScalarFunction(obj))

  return m, X
end
```

```
function golomb(n, L)
  m = JuMP.Model(CBLS.Optimizer)

  @variable(m, 0 <= X[1:n] <= L, Int)

  @constraint(m, X in AllDifferent())
  @constraint(m, X in Ordered()) # optional

  # No two pairs have the same length
  for i in 1:(n - 1), j in (i + 1):n
    for k in i:(n - 1), l in (k + 1):n
      (i, j) < (k, l) || continue
      @constraint(m,
        [X[i], X[j], X[k], X[l]] in DistDifferent()
      )
    end
  end

  # Add objective
  @objective(m, Min, ScalarFunction(maximum))

  return m, X
end
```

6.2 Constraints and compositions

Along with the magic-square model in Section 1, the models in this zoo use the following constraints: `:all_different`, `:dist_different`, `:linear`, and `:ordered`.
The *AllDifferent* constraint ensures that all the values of a given configuration are unique.

```
function all_different(x;
  X = zeros(length(x), 1), param=nothing, dom_size
)
  tr_in(Tuple([tr_count_eq_left]), X, x, param)
  for i in 1:length(x)
    X[i,1] = ar_sum(@view X[i,:])
  end
  return ag_count_positive(@view X[:,1]) |> (
      y -> co_identity(
        y; param, dom_size, nvars=length(x)
      )
    )
end
```

DistDifferent is constraint ensuring that $|x[1]-x[2]| \neq |x[3]-x[4]|$ for any vector x of size 4.

```
function dist_different(x)
  return abs(x[1] - x[2]) != abs(x[3] - x[4])
end
```

Linear (also called *Sum* or *LinearSum* in the literature) Global ensures that the sum of the values of x is equal to a given parameter $param$. Note that this version is a simplification of a linear sum of values with some coefficients.

```
function linear(x; param, dom_size)
  return abs(sum(x) - param) / dom_size
end
```

Ordered is a constraint ensuring that all the values of x are ordered (here in a decreasing order).

```
function ordered(x;
  X = zeros(length(x), 1), param=nothing, dom_size
)
  tr_in(
    Tuple([tr_contiguous_vals_minus]), X, x, param
  )
  for i in 1:length(x)
    X[i,1] = ar_sum(@view X[i,:])
  end
  return ag_count_positive(@view X[:,1]) |> (
      y -> co_identity(
        y; param, dom_size, nvars=length(x)
      )
    )
end
```

7. Acknowledgements

We're grateful to Ricardo ROSA and Pranshu MALIK for the logo of JuliaConstraints presented as Figure 1.

8. References

[1] Jean-François Baffier, Khalil Chrit, Florian Richoux, Pedro Patinho, and Salvador Abreu. Interpretable composition networks: A scaling glass-box neural network to learn combinatorial functions. *IJCAI21-DSO Special issue of annals of mathematics and artificial intelligence on Data Science meets Optimization*, 2, 2022.

[2] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[3] Félix Chalumeau, Ilan Coulon, Quentin Cappart, and Louis-Martin Rousseau. Seapearl: A constraint programming solver guided by reinforcement learning. *CoRR*, abs/2102.09193, 2021. 2102.09193.

[4] Philippe Codognet, Danny Munera, Daniel Diaz, and Salvador Abreu. Parallel local search. In Youssef Hamadi and Lakhdar Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 381–417. Springer, 2018. doi:10.1007/978-3-319-63516-3_10.

[5] Florian Richoux and Jean-François Baffier. Automatic cost function learning with interpretable compositional networks, 2020. 2002.09811.

[6] Florian Richoux, Alberto Uriarte, and Jean-François Baffier. Ghost: A combinatorial optimization framework for rts-related problems. *IEEE Transactions on Computational Intelligence and AI in Games*, PP(99):1–1, 2016. doi:10.1109/TCIAIG.2016.2573199.

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