Defining Neural Networks using constraint programming

Noric Couderc

March 6, 2016

Contents

1 Background

Neural networks

A network is a graph, with nodes, and edges.

In a neural network, the nodes are neurons, and the edges are synapses. Computer scientists use an abstraction of these neurons to perform various kinds of computations. These abstracted neurons are composed of several parts: A number of entry points, with a weight for each entry point, a summation unit that computes the weighted sum of the inputs, and a (user defined) function that transforms the output of the summation unit. Usually, this function is similar to a threshold.

In other words, one can see a neuron as a function $f : \mathbb{R}^n \to \mathbb{R}^m$. The function f is composed of a dot product by a weight vector, composed with a threshold-like function, applied to each component of the input vector: $t(x, y, z) = (t'(x), t'(y), t'(z))$. The result of a neuron corresponds to a function f with $f(\mathbf{v}) = t(\mathbf{v} \cdot \mathbf{w})$. The threshold function t' is either a usual threshold function or a smoother function, like $\tanh$, a sigmoid function: where $S(x) = \frac{1}{1+\exp(-x)}$, or a rectifier $r(x) = \max(0, x)$.

When it comes to the structure of the network itself, we are going to consider the most general case. In this case, the input of the network is a number of values – an input vector – and all inputs are connected to all the nodes of a first layer of neurons. The first layer has an output connected to all the inputs of the following layer, etc. This means each layer is considered a transformation $\ell : \mathbb{R}^n \to \mathbb{R}^m$. In the current setup, the input vector is multiplied by a matrix before application of the threshold function.

The process of machine learning is the ability to derive, from a ***relation*** – i.e. a set $R \subseteq \mathbb{R}^n \times \mathbb{R}^m$ – a computational device that can produce a new element of the relation, while keeping the semantic structure of the relation.

Motivation

The process of finding an easy way to map input data to output data is called *modeling*. The model here is the neural network itself, however, the parameters of the neural network – the weights on all connections – are unknown.

Finding these parameters, considering the constraints of input-output pairs is a constraint satisfaction problem. This article presents an alternative method for training neural networks by using interval arithmetic and constraint propagation. Indeed, constraint propagation allows dropping entire uninteresting

sections of the parameter space, therefore, the region describing the solution will be found *no matter its shape.*

Use cases and API

Before defining the model of neural networks, it might be interesting to define first how the networks are going to be defined and how they are going to be used.

First, the main task of the neural networks is to find a match between two data sets, or more specifically, a number of pairs of input vectors and output vectors.

On the other hand, one of the most important features to have is to be able to easily *change* the network structure in case the matching process did not work.

Moreover, the last task is to be able to export the representation of the neural network to something one could use outside of the constraint programming paradigm.

Enumerating network configurations

In case a matching did not succeed, one possible solution would be to change the network structure. Is it possible to do so in an easy way? And is it possible to make sure *some* configuration will work?

In the current setup, only a specific kind of neural networks is considered: networks which have an input layer, and output layer, and hidden layers in between, so that each layer is connected to the next.

Therefore, the smallest possible network for a function $f : \mathbb{R}^m \to \mathbb{R}^n$ has only one layer, with m inputs and n outputs. This network could be represented by a pair $[(m, n)]$.

In the same fashion, a possible network for a function $f : \mathbb{R}^m \to \mathbb{R}^n$ could be represented by any list $[(m, d_1), (d_1, d_2), (d_2, d_3), \cdots (d_j, n)]$. Removing the duplicates, a network structure can be represented by a list $[m, d_1, d_2, \cdots, d_j, n]$, although this representation is less explicit about the functional aspect of the layers, it show clearly the number of neurons a layer contains.

By fixing limits to the size of the list and the number of values – here the maximum number of dimensions a layer can output – it is possible to enumerate all possible network configurations.

Increasing flexibility

In many cases, it is not possible to find a perfect match between all the input-output pairs. In other words, it is often impossible to have a neural network with 0% error. In constraint programming, allowing error in a system is done *explicitly.* This is done by inserting *slack variables* that are going to represent the error of the system. Then, the problem becomes a constraint optimization problem, where one wants to minimize the error. However, the advantage of using constraint programming in the current setup is that one can *define* the maximum error of the network, and the system will work "backwards" to try to find a network that satisfies what the user wants.

In order to be able to deal with such approximation, the expected output of the network is separated from the actual output of the network. For a dataset of size S, the ith expected output is called $\mathbf{y}_i$, the ith actual output is called $\mathbf{o}_i$, and there is an error vector $\boldsymbol{\delta}_i$ so that:

$$\forall i \in [1, S] : \mathbf{y}_i = \mathbf{o}_i + \boldsymbol{\delta}_i$$

In this context, defining an error of 80% means defining another constraint on $\boldsymbol{\delta}_i$.

$$|\boldsymbol{\delta}_i| = 0.8 \times |\mathbf{o}_i|$$

Exporting networks

TODO

2 A model of neural networks

Parameters

The relation to be modeled: A set $R \subseteq \mathbb{R}^{d_i} \times \mathbb{R}^{d_o}$, representing the function to be modeled. Here d_i will denote the number of dimensions in input data and d_o the number of dimensions in output data.

The input and output data: The relation R is represented by two matrices X and Y of respective sizes $|R| \times d_i$ and $|R| \times d_o$.

Layer specification: To describe the structure of the network, several parameters are needed: L denotes the number of layers of the network. The rest of the structure is represented as a list $[d_i, d_1, d_2 \dots d_{L-1}, d_o]$. In other words, specifying the layer structure amounts to specify a composition – or a sequence – of transformations from one space to another, with the only constraint being that the output of a transformation corresponds to the input of the following transformation.

Threshold function(s): The threshold function, called t, for each neuron is also provided by the user.

Decision Variables

Each layer ℓ has:

- A $|R| \times n$ input matrix I_ℓ.
- A $n \times m$ weight matrix M_ℓ.
- A $|R| \times m$ ***raw*** output matrix R_ℓ, it represents the result of $I_\ell M_\ell$ before applying the threshold function.
- A $|R| \times m$ ***processed*** output matrix O_ℓ, which represents the result of mapping the threshold function on R_ℓ.

All decision variables defined above have the same domain, namely the largest range between the range of the input data and the output data.

For each input-output vector couple, the vector $\boldsymbol{\delta}$ represents the difference between the expected output and the actual output of the network.

- For each couple $(\mathbf{v}_i, \mathbf{w}_i)$ where $i \in [1, |R|[$ there exists an error vector: $\boldsymbol{\delta}_i$

Constraints

The constraints of the system are the mathematical definitions of matrix multiplication and application of a function.

Matrix multiplication: Given an $m \times n$ matrix A and a $n \times p$ matrix B:

$$C = AB \leftrightarrow \forall i \in [1, m[: \forall j \in [1, p[: C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

Layer matrix transformation: For each layer ℓ,

$$R_\ell = I_\ell M_\ell$$

Map with the threshold function: For each layer ℓ,

$$\forall i \in [1, |R|[: \forall j \in [1, m[: O_{\ell,ij} = t(R_{\ell,ij})$$

The allowed error of the network is expressed as a possible difference – denoted Δ – between the output matrix of the last layer O_L and the expected output matrix W.

Error:

$$O_L = Y + \Delta$$

Process

The data and the constraints are handled in two steps:

1. Create storage for intermediate results: Create the structure that will hold the results of each layer of the network, in the form of matrices.
2. Build the layers, that connect the various matrices: Given the sizes of the matrices, and considering the rules of matrix multiplication. The intermediate structure of the layer is built and constrained.