11-1-2021

Why Model Order Reduction

Salvador Robles
The University of Texas at El Paso, sroblesher1@miners.utep.edu

Martine Ceberio
The University of Texas at El Paso, mceberio@utep.edu

Vladik Kreinovich
The University of Texas at El Paso, vladik@utep.edu

Comments:
Technical Report: UTEP-CS-21-97

Why Model Order Reduction

Salvador Robles, Martine Ceberio, and Vladik Kreinovich

Abstract Reasonably recently, a new efficient method appeared for solving complex non-linear differential equations (and systems of differential equations). In this method – known as Model Order Reduction (MOR) – we select several solutions, and approximate a general solution by a linear combination of the selected solutions. In this paper, we use the known explanation for efficiency of neural networks to explain the efficiency of MOR techniques.

1 Formulation of the Problem

We need to solve systems of differential equations. In physics, in engineering, in many areas of biology, the corresponding phenomena are described by systems of differential equations. Thus, to make predictions about these phenomena, we need to solve such systems.

Solving systems of differential equations is difficult. In general, systems of differential equations are difficult to solve. This difficulty is easy to explain:

- In general, when we solve a system of N equations with N unknowns, the more unknowns we have, the more difficult it is to solve this system.
- In systems of differential equations, the unknowns are the functions $s(x)$. To exactly describe a general function, we need to describe infinitely many different numerical values – e.g., the values $s(x_i)$ of this function at all possible points x_i.

The more accurately we want to represent a function, the more parameters we will need. To get a good approximation to the desired function, we therefore need to

Salvador Robles, Martine Ceberio, and Vladik Kreinovich
Department of Computer Science, University of Texas at El Paso
El Paso, Texas 79968, USA
e-mail: sroblesher1@miners.utep.edu, mceberio@utep.edu, vladik@utep.edu

solve a system with a large number of unknowns – which requires a lot of computational efforts.

Model Order Reduction. Reasonably recently, a new method appeared – known as Model Order Reduction (MOR, for short) that helps to solve systems of differential equations; see, e.g., [1]. In this method, once we have found several different solutions $s_1(x), \ldots, s_n(x)$, we then look for approximate solutions $s(x)$ which are linear combinations of the known solutions, i.e., that have the form

$$s(x) = c_1 \cdot s_1(x) + \ldots + c_n \cdot s_n(x)$$

for some coefficient c_i.

In this approximation, we have only n unknowns, so when n is reasonably small, we have a relatively easy-to-solve system of equations.

This method works, but why? The main idea of this method comes from linear systems, where, once you have several solutions, any linear combination of these solutions is also a solution. Many real-life systems are, however, non-linear. Interestingly, MOR method works very well for many non-linear systems as well.

Why it works is not clear. In this paper, we provide a possible explanation for this empirical success. This explanation is related to the explanation of another empirical success phenomena – an explanation of why neural networks (see, e.g., [2, 3, 4]) work well in many situations.

2 From Neural Networks to Model Order Reduction: Our Explanation

Why neural networks: a reminder. One of the main original motivations for neural networks came from the need to speed up computations – and from the observation of how biological neural networks process data.

Computers can now perform many tasks that humans do: e.g., they can recognize faces, control cars, etc. However, computers perform these tasks by using super-fast processing units that perform billions of operations per seconds, while we humans perform the same tasks by using neurons the fastest of which can perform at most 100 operations per second. The reason why a human brain can make important decisions in a short period of time is that in the brain, there are billions of neurons that work in parallel. As a result, during the time when one neuron processes data, all involved neurons perform billions of computational steps.

What is the fastest way to set up such parallel computations? In parallel computations, first, all the processors perform some operations, then they perform some other operations, etc. Computations are the fastest when each of these operations requires the smallest amount of computation time, and when the number of such consequent operations is the smallest possible.

Which operations are the fastest? In a deterministic computer, the result of each operation is uniquely determined by its inputs, i.e., in mathematical terms, is a function of these inputs. Out of all possible functions, linear functions are the fastest to compute. However, we cannot use only linear functions: if all the processors were computing linear functions of their inputs, then all we could compute are compositions of linear functions – which are also linear, while many real-life processes are non-linear. Thus, in addition to linear functions, we should also compute some non-linear functions.

In general, the more inputs we have, the longer it takes to perform the corresponding computations. Thus, the fastest is to compute non-linear functions with the smallest number of inputs – i.e., non-linear functions $s(x)$ with only one input x. So, to make computations faster, on each computation stage, we either compute a linear function or a non-linear function of one variable.

To make computations fast, a linear stage cannot be followed by a linear stage. Indeed, if after computing a linear function, we again compute a linear function of the first stage's output, we will still be computing a linear function of the original inputs – and this can be done in a single stage. Similarly, if we first compute a function $y = s(x)$ of one variable, and then compute another function $z = t(y)$ of one variable, then, in effect, we compute a composition $z = t(s(x))$ of these functions, and this can also be done in a single stage. Thus, in fast computations, a linear stage must be followed by a non-linear stage, and a non-linear stage must be followed by a linear stage.

How many stages do we need? If we use only one stage, then all we can compute are either linear functions or functions of one variable, and many real-life quantities depend non-linearly on several variables. So, we need to have at least two layers. This is exactly how a neural network works: each of its processing units (neurons):

- first computes a linear combination $y = w_1 \cdot x_1 + \ldots + w_n \cdot x_n + w_0$ of its inputs $x_1, \ldots, x_n$, and
- then applies a non-linear function $z = s(y)$ – known as an *activation function* – to the resulting value y.

As a result, each neuron computes the value

$$z = s(w_1 \cdot x_1 + \ldots + w_n \cdot x_n + w_0).$$

From general to specific computational problems. Neural networks are used for machine learning, when:

- we have no prior information about the dependence between the quantities, and
- we want to determine this dependence based on observation results.

In this case, it makes sense to require that a neural network be able to approximate any possible dependence. So, the activation functions are selected to make sure that the corresponding neural networks are *universal approximators* – i.e., that they can approximate any reasonable function with any given accuracy.

For a general neural network, in principle, in addition to activation functions (that need to be computed every time), we can also use functions that have already been computed before. Using these functions will not add computation time – since these functions have already been computed before. However, since a neural network is intended to compute all possible functions from all possible domains, having a pre-computed function from, e.g., biology will probably not help in solving the next problem which may be from geosciences.

In contrast, when we solve a given system of differential equations, we are interested in very specific functions – solutions to this system of equations. Many of these solutions – e.g., corresponding to similar initial conditions – are similar, so it is reasonable to expect that knowing a solution to a similar problem can help in solving the current problem. Thus, for solving systems of differential equations, it makes sense to consider, in addition to activation functions (of one variable), also use pre-computed functions $s_1(x), \ldots, s_n(x)$, possibly of several variables.

What can we compute this way if we use the fastest (two-stage) computations? Since a linear layer cannot be followed by a linear one and a non-linear stage cannot be followed by a non-linear one, we have two options:

- we can have a linear stage followed by a non-linear stage; we will denote this option by L-NL, and
- we can have a non-linear stage followed by a linear stage; we will denote this option by NL-L.

L-NL option. In this option, first, we compute some linear combinations $T(x)$ of the inputs, and then apply an appropriate non-linear function s_i, resulting in $s_i(T(x))$.

The problem is that in this option, we have n different families of functions corresponding to using n different pre-computed functions $s_i(x)$. There is no continuous transition between these families. In this sense, we have a union of n disconnected families of functions. However, what we want to approximate is the family of all solutions, which continuously depend on initial conditions and parameters of the system. In other words, in this option, there is a discrepancy between:

- the class of functions that we want to approximate – namely, the class of all solutions corresponding to different initial conditions and different values of the parameters, and
- the class of functions that we use for approximation – in this option, the class of functions $s_i(T(x))$ corresponding to $i = 1, \ldots, n$.

This leaves us with the need to consider the second option.

NL-L option. In this case, first, we apply non-linear functions, i.e., compute the values $y_1 = s_1(x), \ldots, y_n = s_n(x)$, and then we compute a linear combination of these values, i.e., an expression

$$c_1 \cdot y_1 + \ldots + c_n \cdot y_n + c_0 = c_1 \cdot s_1(x) + \ldots + c_n \cdot s_n(x) + c_0.$$

In this options, different solutions correspond to different values of c_i, so they can be easily smoothly transformed into one another.

Modulo a constant term c_0, what we get in this option is exactly the approximation used in Model Order Reduction (MOR). Thus, we have indeed explained the empirical success of the MOR techniques: they naturally appear if we are looking for the fastest-to-compute approximations.

Acknowledgments

This work was supported in part by the National Science Foundation grants:

- 1623190 (A Model of Change for Preparing a New Generation for Professional Practice in Computer Science), and
- HRD-1834620 and HRD-2034030 (CAHSI Includes).

It was also supported:

- by the AT&T Fellowship in Information Technology, and
- by the program of the development of the Scientific-Educational Mathematical Center of Volga Federal District No. 075-02-2020-1478.

References

1. P. Benner, S. Grivet-Talosia, A. Quarteroni, G. Rozza, W. Schilders, and L. M. Silveira (eds.), *Model Order Reduction*, de Gruyter, Berlin, 2020.
2. C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, New York, 2006.
3. I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, Cambridge, Massachusetts, 2016.
4. V. Kreinovich and O. Kosheleva, "Optimization under uncertainty explains empirical success of deep learning heuristics", In: P. Pardalos, V. Rasskazova, and M. N. Vrahatis (eds.), *Black Box Optimization, Machine Learning and No-Free Lunch Theorems*, Springer, Cham, Switzerland, 2021, pp. 195–220.