2-2018

From Traditional Neural Networks to Deep Learning: Towards Mathematical Foundations of Empirical Successes

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

Comments:
Technical Report: UTEP-CS-18-16

From Traditional Neural Networks to Deep Learning: Towards Mathematical Foundations of Empirical Successes

Vladik Kreinovich
Department of Computer Science
University of Texas at El Paso
500 W. University
El Paso, Texas 79968, USA
vladik@utep.edu
http://www.cs.utep.edu/vladik

***Abstract*—How do we make computers think? To make machines that fly, it is reasonable to look at the creatures that know how to fly: the birds. To make computers think, it is reasonable to analyze how we think – this is the main origin of neural networks. At first, one of the main motivations was speed – since even with slow biological neurons, we often process information fast. The need for speed motivated traditional 3-layer neural networks. At present, computer speed is rarely a problem, but accuracy is – this motivated deep learning. In this paper, we concentrate on the need to provide mathematical foundations for the empirical success of deep learning.**

I. Traditional Neural Networks: A Brief History

Why traditional neural networks: (sanitized) history. How do we make computers think?

- To make machines that fly, it is reasonable to look at the creatures that know how to fly: the birds.
- To make computers think, it is reasonable to analyze how we humans think.

On the biological level, our brain processes information via special cells called *neurons*.

Somewhat surprisingly, in the brain, signals are electric – just as in the computer. The main difference is that in a neural network, signals are sequence of identical pulses. The intensity of a signal is described by the frequency of pulses.

A neuron has many inputs (up to 10^4). All the inputs $x_1, \ldots, x_n$ are combined, with some loss, into a frequency

$$\sum_{i=1}^{n} w_i \cdot x_i.$$

Low inputs do not active the neuron at all, high inputs lead to largest activation.The output signal is a non-linear function

$$y = s_0\left(\sum_{i=1}^{n} w_i \cdot x_i - w_0\right).$$

In biological neurons,

$$s_0(x) = \frac{1}{1+\exp(-x)}.$$

Traditional neural networks emulate such biological neurons; see, e.g., [2].

Why traditional neural networks: real history. At first, researchers ignored non-linearity and only used linear neurons. They got good results and made many promises.

The euphoria ended in the late 1960s when MIT's Marvin Minsky and Seymour Papert published a book [10]. Their main result was that a composition of linear functions is linear (I am not kidding). This ended the hopes of original schemes.

For some time, neural networks became a bad word. Then, smart researchers came us with a genius idea: let's make neurons non-linear. This revived the field.

Traditional neural networks: main motivation. One of the main motivations for neural networks was that computers were slow. Although human neurons are much slower than CPU, the human processing was often faster. So, the main motivation was to make data processing faster. The idea was that since we are the result of billion years of ever improving evolution, our biological mechanics should be optimal (or close to optimal).

To make processing faster, we need to have many fast processing units working in parallel. The fewer layers, the smaller overall processing time.

In nature, there are many fast linear processes – e.g., combining electric signals. As a result, linear processing (L) is faster than non-linear one.

For non-linear processing, the more inputs, the longer it takes. So, the fastest non-linear processing (NL) units process just one input. It turns out that two layers are not enough to approximate any function.

Why one or two layers are not enough. With one linear (L) layer, we only get linear functions. With one nonlinear (NL) layer, we only get functions of one variable.

With L→NL layers, we get

$$s_0\left(\sum_{i=1}^{n} w_i \cdot x_i - w_0\right).$$

For these functions, the level sets

$$f(x_1, \ldots, x_n) = \text{const}$$

are planes

$$\sum_{i=1}^{n} w_i \cdot x_i = c.$$

Thus, they cannot approximate, e.g., $f(x_1, x_2) = x_1 \cdot x_2$ for which the level set is a hyperbola.

For NL→L layers, we get

$$f(x_1, \ldots, x_n) = \sum_{i=1}^{n} s_i(x_i).$$

For all these functions,

$$d \stackrel{\text{def}}{=} \frac{\partial^2 f}{\partial x_1 \partial x_2} = 0,$$

so we also cannot approximate

$$f(x_1, x_2) = x_1 \cdot x_2$$

with $d = 1 \neq 0$.

Why three layers are sufficient: Newton's prism and Fourier transform. In principle, we can have two 3-layer configurations: L→NL→L and NL→L→NL. Since L is faster than NL, the fastest is L→NL→L:

$$y = \sum_{k=1}^{K} W_k \cdot s_k\left(\sum_{i=1}^{n} w_{ki} \cdot x_i - w_{k0}\right) - W_0.$$

Newton showed that a prism decomposes while light (or any light) into elementary colors. In precise terms, elementary colors are sinusoids

$$A \cdot \sin(w \cdot t) + B \cdot \cos(w \cdot t).$$

Thus, every function can be approximated, with any accuracy, as a linear combination of sinusoids:

$$f(x_1) \approx \sum_k (A_k \cdot \sin(w_k \cdot x_1) + B_k \cdot \cos(w_k \cdot x_1)).$$

This result was theoretically proven later by Fourier.

For $f(x_1, x_2)$, we get a similar expression for each x_2, with $A_k(x_2)$ and $B_k(x_2)$. We can similarly represent $A_k(x_2)$ and $B_k(x_2)$, thus getting products of sines, and it is known that, e.g.:

$$\cos(a) \cdot \cos(b) = \frac{1}{2} \cdot (\cos(a+b) + \cos(a-b)).$$

Thus, we get an approximation of the desired form with $s_k = \sin$ or $s_k = \cos$:

$$y = \sum_{k=1}^{K} W_k \cdot s_k\left(\sum_{i=1}^{n} w_{ki} \cdot x_i - w_{k0}\right).$$

Which activation functions $s_k(z)$ should we choose. A general 3-layer NN has the form:

$$y = \sum_{k=1}^{K} W_k \cdot s_k\left(\sum_{i=1}^{n} w_{ki} \cdot x_i - w_{k0}\right) - W_0.$$

Biological neurons use

$$s_0(z) = \frac{1}{1 + \exp(-z)},$$

but shall we simulate it?

Simulations are not always efficient. E.g., airplanes have wings like birds but they do not flap them.

Let us analyze this problem theoretically. There is always some noise c in the communication channel. So, we can consider either the original signals x_i or denoised ones $x_i - c$. The results should not change if we perform a full or partial denoising $z \to z' = z - c$.

Denoising means replacing $y = s_0(z)$ with $y' = s_0(z - c)$. So, $s_0(z)$ should not change under shift $z \to z - c$.

Of course, $s_0(z)$ cannot remain the same: if

$$s_0(z) = s_0(z - c)$$

for all c, then $s_0(z) = \text{const}$. The idea is that once we re-scale x, we should get the same formula after we apply a "natural" y-re-scaling T_c:

$$s_0(x - c) = T_c(s_0(x)).$$

But which re-scalings are natural?

Which transformations are natural? Linear re-scalings are natural: they corresponding to changing units and starting points (like C to F).

An inverse T_c^{-1} to a natural re-scaling T_c should also be natural. A composition $y \to T_c(T_{c'}(y))$ of two natural re-scalings T_c and $T_{c'}$ should also be natural.

In mathematical terms, natural re-scalings form a *group*.

For practical purposes, we should only consider re-scaling determined by finitely many parameters. So, we look for a finite-parametric group containing all linear transformations.

A somewhat unexpected approach. N. Wiener, in his famous book *Cybernetics* [13], noticed that when we approach an object, we have distinct phases:

- first, we see a blob (the image is invariant under all transformations);
- then, we start distinguishing angles from smooth but not sizes (projective transformations);
- after that, we detect parallel lines (affine transformations);
- then, we detect relative sizes (similarities);
- finally, we see the exact shapes and sizes.

Are there other transformation groups?

Wiener argued that if there were other groups, after billions years of evolutions, we would use them. So he conjectured that there are no other groups.

Wiener was right. Wiener's conjecture was indeed proven in the 1960s. In 1-D case, this means that all our transformations are fractionally linear:

$$s_0(z-c) = \frac{A(c)\cdot s_0(z) + B(c)}{C(c)\cdot s_0(z) + D(c)};$$

see Appendix for technical details.

For $c = 0$, we get $A(0) = D(0) = 1$, $B(0) = C(0) = 0$. Differentiating the above equation by c and taking $c = 0$, we get a differential equation for $s_0(z)$:

$$-\frac{ds_0}{dz} = (A'(0)\cdot s_0(z) + B'(0)) - s_0(z)\cdot(C'(0)\cdot s_0(z) + D'(0)).$$

So,

$$\frac{ds_0}{C'(0)\cdot s_0^2 + (A'(0) - C'(0))\cdot s_0 + B'(0)} = -dz.$$

Integrating, we indeed get

$$s_0(z) = \frac{1}{1+\exp(-z)}.$$

(after an appropriate linear re-scaling of z and $s_0(z)$); see [9], [11], [12].

How to train traditional neural networks: main idea. *Reminder:* a 3-layer neural network has the form:

$$y = \sum_{k=1}^{K} W_k \cdot s_0\left(\sum_{i=1}^{n} w_{ki}\cdot x_i - w_{k0}\right) - W_0.$$

We need to find the weights that best described observations

$$\left(x_1^{(p)}, \ldots, x_n^{(p)}, y^{(p)}\right), \quad 1 \le p \le P.$$

We find the weights that minimize the mean square approximation error

$$E \stackrel{\text{def}}{=} \sum_{p=1}^{P}\left(y^{(p)} - y_{NN}^{(p)}\right)^2,$$

where

$$y^{(p)} = \sum_{k=1}^{K} W_k \cdot s_0\left(\sum_{i=1}^{n} w_{ki}\cdot x_i^{(p)} - w_{k0}\right) - W_0.$$

The simplest minimization algorithm is gradient descent:

$$w_i \to w_i - \lambda\cdot\frac{\partial E}{\partial w_i}.$$

Towards faster differentiation. To achieve high accuracy, we need many neurons. Thus, we need to find many weights. To apply gradient descent, we need to compute all partial derivatives

$$\frac{\partial E}{\partial w_i}.$$

Differentiating a function f is easy:

- the expression f is a sequence of elementary steps,
- so we take into account that $(f\pm g)' = f'\pm g'$, $(f\cdot g)' = f'\cdot g + f\cdot g'$, $(f(g))' = f'(g)\cdot g'$, etc.

For a function that takes T steps to compute, computing f' thus takes $c_0\cdot T$ steps, with $c_0 \le 3$.

However, for a function of n variables, we need to compute n derivatives. This would take time $n\cdot c_0\cdot T \gg T$: this is too long.

Faster differentiation: backpropagation. Idea:

- instead of starting from the variables,
- start from the last step, and compute

$$\frac{\partial E}{\partial v}$$

for all intermediate results v.

For example, if the very last step is $E = a\cdot b$, then

$$\frac{\partial E}{\partial a} = b$$

and

$$\frac{\partial E}{\partial b} = a.$$

At each step y, if we know

$$\frac{\partial E}{\partial v}$$

and $v = a\cdot b$, then

$$\frac{\partial E}{\partial a} = \frac{\partial E}{\partial v}\cdot b$$

and

$$\frac{\partial E}{\partial b} = \frac{\partial E}{\partial v}\cdot a.$$

At the end, we get all n derivatives

$$\frac{\partial E}{\partial w_i}$$

in time

$$c_0\cdot T \ll c_0\cdot T\cdot n.$$

This is known as *backpropagation*; see, e.g., [2].

II. Beyond Traditional Neural Networks (NN)

Need for deep learning. Nowadays, computer speed is no longer a big problem. What *is* a problem is *accuracy:* even after thousands of iterations, the NNs do not learn well. So, instead of computation speed, we would like to maximize learning accuracy.

We can still consider L and NL elements. For the same number of variables w_i, we want to get more accurate approximations. For given number of variables, and given accuracy, we get N possible combinations. If all combinations correspond to different functions, we can implement N functions. However, if some combinations lead to the same function, we implement fewer different functions.

For a traditional NN with K neurons, each of $K!$ permutations of neurons retains the resulting function; see, e.g., [6]. Thus, instead of N functions, we only implement

$$\frac{N}{K!} \ll N \text{ functions.}$$

Thus, to increase accuracy, we need to minimize the number K of neurons in each layer.

To get a good accuracy, we need many parameters, thus many neurons. Since each layer is small, we thus need many layers. This is the *main idea* behind *deep learning* [1], [5].

III. Empirical Formulas Behind Deep Learning Successes and How They Can Be Justified

Formulation of the problem. While the general idea of deep learning is natural, many specific formulas that lead to deep learning successes are purely empirical, and need to be explained. In this section, we list such formulas, and briefly mention how the corresponding formulas can be explained.

Rectified linear neurons. Instead of complex nonlinear neurons used in traditional neural networks, deep networks utilize *rectified linear neurons* for which the activation function $s_0(z)$ has the form

$$s_0(z) = \max(0, z).$$

Our explanation [3] is that:

- this activation function is invariant under re-scaling (changing of the measuring unit) $z \to \lambda \cdot x$;
- moreover, it is, in effect, the only activation function which is this invariant, and
- it is, in effect, the only activation which is optimal with respect to any scale-invariant optimality criterion.

Combining several results. To speed up the training, the current deep learning algorithms use dropout techniques:

- they train several sub-networks on different portions of data, and then
- "average" the results.

A natural idea is to use arithmetic mean for this "averaging", but empirically, geometric mean works much better. In [4], we provide a theoretical explanation for the empirical efficiency of selecting geometric mean as the "averaging" in dropout training: namely, it turns out that

- this choice is scale-invariant – and,
- in effect, it is the only scale-invariant choice.

Softmax. In deep learning, instead of selecting an alternative for which the objective function $f(x)$ is the largest possible, we use so-called *softmax* – i.e., select each alternative x with probability proportional to $\exp(\alpha \cdot f(x))$, for some $\alpha > 0$.

In general, we could select any increasing function $F(z)$ and select probabilities proportional to $F(f(x))$, so why exponential function is the most successful? It turns out [8], [11] that:

- when we use this function, the resulting probabilities do not change if we simply shift all the values $f(x)$, i.e., change them to $f(x) + c$ for some c – which does not change the original optimization problem;
- moreover, exponential functions are the only ones which lead to such shift-invariant selection, and,
- in effect, the exponential functions are only ones which can be optimal under a shift-invariant optimality criterion.

Replacing Least Squares with a more complex expression. In deep learning, instead of the Least Squares, we often optimize the Kullback-Leibler (KL) divergence

$$-\sum_{k=1}^{K}\left[y^{(k)} \cdot \log\left(y_{\rm NN}^{(k)}\right) + \left(1 - y^{(k)}\right) \cdot \log\left(1 - y_{\rm NN}^{(k)}\right)\right].$$

A possible explanation for the empirical success of this formula is given in [7].

Acknowledgments

This work was supported in part by the US National Science Foundation grant HRD-1242122.

References

[1] C. Baral, O. Fuentes, and V. Kreinovich, "Why deep neural networks: a possible theoretical explanation", In: M. Ceberio et al. (eds.), *Constraint Programming and Decision Making: Theory and Applications*, Springer Verlag, 2018, pp. 1–6.

[2] C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, New York, 2006.

[3] O. Fuentes, J. Parra, E. Anthony, and V. Kreinovich, "Why rectified linear neurons are efficient: a possible theoretical explanations", In: O. Kosheleva, S. Shary, G. Xiang, and R. Zapatrin (eds.), *Beyond Traditional Probabilistic Data Processing Techniques: Interval, Fuzzy, etc. Methods and Their Applications*, Springer, Cham, Switzerland, 2018, to appear.

[4] A. Gholamy, J. Parra, V. Kreinovich, O. Fuentes, and E. Anthony, *How to Best Apply Neural Networks in Geosciences: Towards Optimal "Averaging" in Dropout Training*, University of Texas at El Paso, Department of Computer Science, Technical Report UTEP-CS-17-98, http://www.cs.utep.edu/vladik/2018/tr17-98.pdf

[5] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Leaning*, MIT Press, Cambridge, Massachusetts, 2016.

[6] P. C. Kainen, V. Kurkova, V. Kreinovich, and O. Sirisaengtaksin. "Uniqueness of network parameterization and faster learning", *Neural, Parallel, and Scientific Computations*, 1994, Vol. 2, pp. 459–466.

[7] O. Kosheleva and V. Kreinovich, "Why deep learning methods use kl divergence instead of least squares: a possible pedagogical explanation", *Mathematical Structures and Modeling*, to appear.

[8] V. Kreinovich, "Group-theoretic approach to intractable problems," *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Vol. 417, 1990, pp. 112–121.

[9] V. Kreinovich and C. Quintana. "Neural networks: what non-linearity to choose?," *Proceedings of the 4th University of New Brunswick Artificial Intelligence Workshop*, Fredericton, New Brunswick Canada, 1991, pp. 627–637.

[10] M. Minsky and S. Papert, *Perceptions*, MIT Press, Camdrige, Massachusetts, 1969.

[11] H. T. Nguyen and V. Kreinovich, *Applications of Continuous Mathematics to Computer Science*, Kluwer, Dordrecht, Netherlands, 1997.

[12] O. Sirisaengtaksin, V. Kreinovich, and H. T. Nguyen, "Sigmoid neurons are the safest against additive errors", *Proceedings of the First International Conference on Neural, Parallel, and Scientific Computations*, Atlanta, GA, May 28–31, 1995, Vol. 1, pp. 419–423.

[13] N. Wiener, *Cybernetics: Or Control and Communication in the Animal and the Machine*, MIT Press, Cambridge, Massachisetts, 1948.

Appendix

Every transformation is a composition of infinitesimal ones

$$x \to x + \varepsilon \cdot f(x),$$

for infinitely small ε. So, it's enough to consider infinitesimal transformations.

The class of the corresponding functions $f(x)$ is known as a *Lie algebra* A of the corresponding transformation group. Infinitesimal linear transformations correspond to

$$f(x) = a + b \cdot x,$$

so all linear functions are in A. In particular, $1 \in A$ and $x \in A$.

For any λ, the product $\varepsilon \cdot \lambda$ is also infinitesimal, so we get

$$x \to x + (\varepsilon \cdot \lambda) \cdot f(x) = x \to x + \varepsilon \cdot (\lambda \cdot f(x)).$$

So, if $f(x) \in A$, then $\lambda \cdot f(x) \in A$.

If we first apply $f(x)$, then $g(x)$, we get

$$x \to (x + \varepsilon \cdot f(x)) + \varepsilon \cdot g(x + \varepsilon \cdot f(x)) =$$

$$x + \varepsilon \cdot (f(x) + g(x)) + o(\varepsilon).$$

Thus, if $f(x) \in A$ and $g(x) \in A$, then $f(x) + g(x) \in A$. So, A is a linear space.

In general, for the composition, we get

$$x \to (x + \varepsilon_1 \cdot f(x)) + \varepsilon_2 \cdot g(x_1 + \varepsilon_1 \cdot f(x)) =$$

$$x + \varepsilon_1 \cdot f(x) + \varepsilon_2 \cdot g(x) + \varepsilon_1 \cdot \varepsilon_2 \cdot g'(x) \cdot f(x) + \text{ quadratic terms.}$$

If we then apply the inverses to $x \to x + \varepsilon_1 \cdot f(x)$ and $x \to x + \varepsilon_2 \cdot g(x)$, the linear terms disappear, we get:

$$x \to x + \varepsilon_1 \cdot \varepsilon_2 \cdot \{f, g\}(x),$$

where

$$\{f, g\} \stackrel{\text{def}}{=} f'(x) \cdot g(x) - f(x) \cdot g'(x).$$

Thus, if $f(x) \in A$ and $g(x) \in A$, then $\{f, g\}(x) \in A$. The expression $\{f, g\}$ is known as the *Poisson bracket*.

Let's expand any function $f(x)$ in Taylor series:

$$f(x) = a_0 + a_1 \cdot x + \ldots$$

If k is the first non-zero term in this expansion, we get

$$f(x) = a_k \cdot x^k + a_{k+1} \cdot x^{k+1} + a_{k+2} \cdot x^{k+2} + \ldots$$

For every λ, the algebra A also contains

$$\lambda^{-k} \cdot f(\lambda \cdot x) = a_k \cdot x^k + \lambda \cdot a_{k+1} \cdot x^{k+1} + \lambda^2 \cdot a_{k+2} \cdot x^{k+2} + \ldots$$

In the limit $\lambda \to 0$, we get $a_k \cdot x^k \in A$, hence $x^k \in A$.

Thus,

$$f(x) - a_k \cdot x^k = a_{k+1} \cdot x^{k+1} + \ldots \in A.$$

We can similarly conclude that A contains all the terms x^n for which $a_n \neq 0$ in the original Taylor expansion.

Since $g(x) = 1 \in A$, for each $f \in A$, we have

$$\{f, 1\} = f'(x) \cdot 1 + f(x) \cdot q' = f'(x) \in A.$$

Thus, for each k, if $x^k \in A$, we have

$$(x^k)' = k \cdot x^{k-1} \in A$$

hence $x^{k-1} \in A$, etc. So, if $x^k \in A$, all smaller power are in A too. In particular, this means that if $x^k \in A$ for some $k \geq 3$, then we have $x^3 \in A$ and $x^2 \in A$; thus:

$$\{x^3, x^2\} = (x^3)' \cdot x^2 - x^3 \cdot (x^2)' = 3 \cdot x^2 \cdot x^2 - x^3 \cdot 2 \cdot x = x^4 \in A.$$

In general, once $x^k \in A$ for $k \geq 3$, we get

$$\{x^k, x^2\} = (x^k)' \cdot x^2 - x^k \cdot (x^2)' = k \cdot x^{k-1} \cdot x^2 - x^k \cdot 2 \cdot x =$$

$$(k-2) \cdot x^{k+1} \in A \text{ hence } x^{k+1} \in A.$$

So, by induction, $x^k \in A$ for all k. Thus, A is infinite-dimensional – which contradicts to our assumption that A is finite-dimensional.

So, we cannot have Taylor terms of power $k \geq 3$; therefore we have:

$$x \to x + \varepsilon \cdot (a_0 + a_1 \cdot x + a_2 \cdot x^2).$$

This corresponds to an infinitesimal fractional-linear transformation

$$x \to \frac{\varepsilon \cdot A + (1 + \varepsilon \cdot B) \cdot x}{1 + \varepsilon \cdot D \cdot x} =$$

$$(\varepsilon \cdot A + (1 + \varepsilon \cdot B) \cdot x) \cdot (1 - \varepsilon \cdot D \cdot x) + o(\varepsilon) =$$

$$x + \varepsilon \cdot (A + (B - D) \cdot x - D \cdot x^2).$$

So, to match, we need

$$A = a_0, \quad D = -a_2, \text{ and } B = a_1 - a_2.$$

We concluded that every infinitesimal transformation is fractionally linear. Every transformation is a composition of infinitesimal ones. Composition of fractional-linear transformations is fractional linear. Thus, all transformations are fractional linear.