Deep Learning (Partly) Demystified

Vladik Kreinovich and Olga Kosheleva
University of Texas at El Paso
500 W. University
El Paso, TX 79968
vladik@utep.edu, olgak@utep.edu

Abstract

Successes of deep learning are partly due to appropriate selection of activation function, pooling functions, etc. Most of these choices have been made based on empirical comparison and heuristic ideas. In this paper, we show that many of these choices – and the surprising success of deep learning in the first place – can be explained by reasonably simple and natural mathematics.

1 Traditional Neural Networks: A Brief Reminder

In other to explain deep neural networks, let us first briefly recall the motivations behind (and the formulas of) traditional ones.

In the old days, computers were much slower, this was a big limitation that prevented us from solving many important practical problems. As a result, researchers started looking for ways to speed up computations.

If a person has a task which takes too long for one person, a natural idea is to ask for help, so that several people can work on this task in parallel – and thus, get the result faster. Similarly, if a computation task takes too long, a natural idea is to have several processing units working in parallel. In this case, the overall computation time is just to time that is needed for each of the processing unit to finish its sub-task.

To minimize the overall time, it is therefore necessary to make these sub-tasks as simple (and thus, as easy to compute) as possible. In data processing, the simplest possible functions to compute are linear functions. However, if we only have processing units that compute linear functions, we will only compute linear functions – since a composition of linear functions is always linear. Thus, we need to supplement these units with some nonlinear units. In general, the more inputs, the more complex (and thus longer) the resulting computations. So, the fastest possible nonlinear units are the ones that compute functions of one variable.

So, our ideal computational device should consist of linear (L) units and nonlinear units (NL) that compute functions of one variable. These units should work in parallel: first, all the units from one layer will work, then all units from another layer, etc. The fewer layers, the faster the resulting computations. One can prove that 1-layer schemes and even 2-layer schemes (both L followed by NL and NL followed by L) do not provide a universal approximation property, i.e., do not compute any given continuous function. One can also prove that 3-layer neurons already have this property. Among two possible 3-layer schemes: L-NL-L and NL-L-NL, the first one is faster, since it uses slower nonlinear units only once.

In this scheme, first, each unit from the first layer applies a linear transformation to the inputs $x_1, \ldots, x_n$, resulting in the values $z_k = \sum_{i=1}^{n} w_{ki} \cdot x_i - w_{k0}$, for some values w_{ki} (known as *weights*). In the next NL layer, these values are transformed into $y_k = s_k(y_k)$, for some nonlinear functions $s_k(z)$. Finally, in the last L layers, the values y_k are linearly combined into the final result

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

This is exactly the formula that describes the traditional neural network; see e.g., [2].

In the traditional neural network, usually, all the NL neurons compute the same function – sigmoid: $s_k(z) = \dfrac{1}{1+\exp(-z)}$.

2 Why Do We Need to Go Beyond Traditional Neural Networks

Traditional neural networks were invented when computers were reasonably slow, which prevented them from solving important practical problems. For these computers, computation speed was the main objective. As we have just shown, this need led to what we know as traditional neural networks.

Nowadays, computers are much faster. In most practical applications, speed is no longer the main problem. But now a new problem emerged: that the traditional neural networks, while fast, have limited accuracy of their predictions. It is therefore desirable to come up with devices that would achieve better prediction *accuracy* – even if to achieve this accuracy, they will be slower than the traditional neural networks. This is, in a nutshell, the main motivation behind what is now known as deep learning; see, e.g., [1, 5, 6, 7].

3 The More Models We Have, the More Accurately We Can Approximate

As a result of training a neural network – or any other machine learning tool – we get the values of some parameters for which the corresponding models provides the best approximation to the actual data. There are usually finitely many parameters, each of which – when represented in a computer – consists of finitely many bits. We can write down all these bits into a single binary sequence.

Let us denote the length of this sequence by N. If we have a parameters each of which consists of b bits, then $N = a \cdot b$.

In these terms, different models that can be obtained from training can be described by different N-bit sequences.

There are two possible values of each bit. For each of these values, we can have two different 2-but extensions depending on whether we extend the original bit with 0 or 1, so we have $2 \cdot 2 = 2^2$ possible sequences of length 2. For each of these 2-bit sequences, we have two possible 3-bits extensions, so the overall number of 3-bit sequences is $2 \cdot 2^2 = 2^3$. In general, for b bits, we have 2^b possible b-bit sequences.

In particular, if we use N bits to store all the information about all the tuned parameters, then we can have 2^N possible binary sequences and thus, 2^N possible models. In these terms, training simply means selecting one of these 2^N possible models.

If we have only one model to represent the actual dependence, this model will be a very lousy description. If we can have two models, we can have more accurate approximations. If we can have 100 different models, we can have an even more accurate one. In general, the more models we have, the more accurate representation we can have.

We can illustrate this idea on the example of approximating real numbers from the interval $[0, 1]$. If we have only one model – e.g., the value $x = 0.5$, then we approximate every other number with accuracy 0.5. If we can have 10 models, then we can take 10 values 0.05, 0.15, ..., 0.95. The first value approximates all the numbers from the interval $[0, 0.1]$ with accuracy 0.05. The second value approximates all the numbers from the interval $[0, 1, 0.2]$ with the same accuracy, etc. As a result, by selecting one these values, we can approximate any number from the interval $[0, 1]$ with accuracy 0.05.

If we can have 100 models, then can similarly take the values 0.005, 0.015, ..., all the way to 0.995 and this approximate any number from the interval $[0, 1]$ with accuracy 0.005.

4 How Many Models Can We Represent with a Traditional Neural Network

Let us consider a traditional neural network with K neurons. Each neuron k is characterized by several weights W_k and w_{ki}. Let b denote the number of bits needed to describe all the weights corresponding to a single neuron. Then, overall, to describe all possible bit sequences resulted from training, we need $N = K \cdot b$ bits.

As we mentioned, we can have 2^N different binary sequences of length N. So, at first glance, one may think that, in line with what we have discussed earlier, we can thus represent 2^N different models. However, the actual number of models is much smaller. The reason for this is that if we swap two neurons – i.e., swap all the bits corresponding to these two neurons – the resulting functions

$$f(x_1, \ldots, x_n) = \sum_{k=1}^{K} W_k \cdot s\left(\sum_{i=1}^{n} w_{ki} \cdot x_i - w_{k0}\right) - W_0$$

will not change – since the sum does not change if we swap two of added numbers.

For example, if we have two neurons, the first one is described by a binary sequence 010 and the second one by 110, then this whole configuration is described by the sequence 010110. If we swap the neurons, we get a different binary sequence 110010 which, however, represents the same dependence – i.e., the same model.

Similarly, if instead of swapping two neurons, we apply any permutation, we get the exact same model.

For K neurons, there are $K!$ possible permutations.

Thus, $K!$ different binary sequences represent the same model. So, by using N neurons, instead of 2^N possible models, we can only have $\dfrac{2^N}{K!}$ possible models.

5 How Can We Achieve Better Accuracy: The Main Idea Behind Deep Learning

As we have mentioned, the more models we can represent, the more accurate will be the resulting approximation. When the overall number of bits is fixed – e.g., by the ability of our computers – the only way to increase the number of models is to decrease $K!$, i.e., to decrease K; see, e.g., [1, 6, 7].

In the traditional neural networks, all the neurons are, in effect, in one layer – known as the hidden layer. The only way to decrease K is to make the number of neurons in each layer much smaller. This means that instead of placing the neurons into a single layer, we place then in many layers. Each of these layers is *narrow* – in the sense that the number of neurons in each later is small. We now have several layers – the construction is *deep* – so that the inputs go to the

first layer, then the outputs of the first layer serve as inputs of the second layer, etc. – until we reach the final layer, which produces the resulting value y.

6 Which Activation Function Should We Use for Deep Learning?

To answer this question, we need to recall that usually, we process the values of physical quantities, but the numerical values of physical quantities depend on what measuring unit we use and – for some quantities like temperature or time – what starting point we select for the measurement.

If we change a measuring unit to a one which is λ times smaller – e.g., replace feet with inches which are 12 times smaller – then all numerical values get multiplied by λ – e.g., 2.5 feet becomes 30 inches. So, instead of the original numerical value x, we get a new numerical value $x' = \lambda \cdot x$.

Similarly, if we replace the original starting point with the new point which is x_0 units before, then each numerical value x is replaced by a new numerical value $x' = x + x_0$. For example, if instead of measuring years from our current Year 0 (supposedly the year when Jesus was born, although many historians disagree), we use Year -3761 (supposedly the year when the world was created, although here no historians agree), then current year 2019 becomes $2019 + 3761 = 5780$, which is our year in the traditional Jewish calendar.

We want to select an activation function $s(x)$ that would not depend on the choice of a measuring unit. In other words, we want to make sure that if $y = s(x)$ and we select a new measuring unit, i.e., switch to new numerical values $x' = \lambda \cdot x$ and $y' = \lambda \cdot y$, then for these new values x' and y', we will have the exact same dependence: $y' = s(x')$.

Substituting the expressions $x' = \lambda \cdot x$ and $y' = \lambda \cdot y$ into this formula, we conclude that $\lambda \cdot y = s(\lambda \cdot x)$. Here, $y = s(x)$, so we conclude that

$$s(\lambda \cdot x) = \lambda \cdot s(x)$$

for all possible x and $\lambda > 0$.

For $x = 1$, we conclude that $s(\lambda) = \lambda \cdot s(1)$. If we denote $s(1)$ by c_+, and rename λ into z, we conclude that for all $z > 0$, we get $s(z) = c_+ \cdot z$.

For $x = -1$, we conclude that $s(-\lambda) = \lambda \cdot s(-1)$. If we denote $-s(-1)$ by c_- (so that $s(-1) = -c_-$) and denote $-\lambda$ by z (so that $\lambda = -z$), we conclude that for all negative values z, we have $s(z) = (-c_-) \cdot (-z) = c_- \cdot z$.

Thus, we conclude that the activation function $s(z)$ should have the following *piecewise linear* form (see, e.g., [3]):

- for $z > 0$, we have $s(z) = c_+ \cdot z$;
- for $z < 0$, we have $s(z) = c_- \cdot z$.

Comment. We must have $c_+ \neq c_-$, since otherwise, the function $s(z)$ would be linear, and we know that with linear functions, we can only describe linear dependencies.

7 What Activation Function Is Actually Used in Deep Learning? Why

To uniquely determine a piecewise linear function, we need to select two real numbers: c_+ and c_-. The simplest possible real numbers is 0 and 1. Thus, the simplest possible piecewise linear function has the form:

- for $z > 0$, we have $s(z) = z$;
- for $z < 0$, we have $s(z) = 0$.

In other words, $s(z) = \max(z, 0)$. This function is known as *rectified linear function*, it is actually used in deep learning; see, e.g., [5].

8 It Does Not Matter Which Piecewise Linear Activation Function to Use

Why not use some other piecewise linear function? Because it does not matter much. Indeed, for all the layers – expect for the last one – the output of a neuron is linearly combined with other signals anyway. And any piecewise linear function can be represented as a linear combination of the rectified linear function $\max(z, 0)$ and the identity function z:

$$s(z) = c_- \cdot z + (c_+ - c_-) \cdot \max(z, 0).$$

Indeed:

- for $z > 0$, the right-hand side is equal to $c_- \cdot z + c_+ \cdot 0 = c_- \cdot z$, while
- for $z < 0$, the right-hand side is equal to

$$c_- \cdot z + (c_+ - c_-) \cdot z = (c_- + (c_+ - c_-)) \cdot z = c_+ \cdot z.$$

9 Why Cannot We Require Shift-Invariance Instead of Scale-Invariance?

We mentioned that the numerical value of a physical quantity changes when we change the measuring unit *and* when we change the starting point. However, when we looked for the activation function, we only considered invariance with respect to changing the unit (it is known as *scale-invariance*). Why not consider invariance with respect to changing the starting point (which is called *shift-invariance*)?

Let us give it a try. Similarly to the case of scale-invariance, we want to make sure that when $y = s(x)$ then for $x' = x + x_0$ and $y' = y + x_0$, we will

have $y' = s(x')$. Substituting the expressions for x' and y' into the formula $y' = s(x')$, we get $y + x_0 = s(x + x_0)$. Here, $s(x) = y$, so we have

$$s(x + x_0) = s(x) + x_0$$

for all possible values x and x_0.

In particular, for $x = 0$, we get $s(x_0) = s(0) + x_0$. Renaming $s(0)$ as a and x_0 as z, we conclude that $s(z) = z + a$. This is a linear function – thus, such neurons cannot describe any non-linear process.

10 Need for Pooling

Often, we have a lot of data points to process. For example, an image is usually represented in a computer by the intensities at each pixel. Even for a not very good 1000 by 1000 picture, we have 1,000,000 pixels – so to process such an image, we need to process 1,000,000 numbers.

In a traditional neural network, we could use as many neurons as needed, but in a deep neural network, there are only a few neurons in the first layer. Thus, before we start processing, we need to combined several input values into one (a similar procedure can also be applied at a later stage). This operation of combining several values into one is known as *pooling*; see, e.g., [5].

11 Which Pooling Operation Shall We Use?

Let us consider the case when we pool two values a and b into a single value c. Let us denote the resulting value c by $p(a, b)$. Of course, the pooling should not depend on the order, i.e., we should have $p(a, b) = p(b, a)$. In other words, the pooling operation should be *commutative*.

Similarly to the above discussions, it is reasonable to require that the result of pooling will not change if we simply change the measuring unit or change the starting point for measurement. So, we make two requirements:

- if $c = p(a, b)$, then $c' = p(a', b')$, where $a' = \lambda \cdot a$, $b' = \lambda \cdot b$, and $c' = \lambda \cdot c$;
- if $c = p(a, b)$, then $c' = p(a', b')$, where $a' = a + a_0$, $b' = b + a_0$, and $c' = c + a_0$.

From the first requirement, substituting the expressions $a' = \lambda \cdot a$, $b' = \lambda \cdot b$, and $c' = \lambda \cdot c$ into the formula $c' = p(a', b')$, we conclude that $\lambda \cdot c = p(\lambda \cdot a, \lambda \cdot b)$. Here, $c = p(a, b)$, so we conclude that

$$p(\lambda \cdot a, \lambda \cdot b) = \lambda \cdot p(a, b).$$

From the second requirement, substituting the expressions $a' = a + a_0$, $b' = b + a_0$, and $c' = c + a_0$ into the formula $c' = p(a', b')$, we conclude that $c + a_0 = p(a + a_0, b + a_0)$. Here, $c = p(a, b)$, so we conclude that

$$p(a + a_0, b + a_0) = p(a, b) + a_0.$$

Let us use the resulting formulas to find the value $p(x, y)$ for all possible pairs (x, y). Without losing generality, we can assume that $x < y$. Then, substituting $a = 0$, $a_0 = x$, and $b = y - x$ into the formula $p(a + a_0, b + a_0) = p(a, b) + a_0$, we conclude that

$$p(x, y) = p(0, y - x) + x.$$

Substituting $\lambda = y - x$, $a = 0$, and $b = 1$ into the formula $p(\lambda \cdot a, \lambda \cdot b) = \lambda \cdot p(a, b)$, we conclude that $p(0, y - x) = (y - x) \cdot p(0, 1)$. Substituting this expression into the formula $p(x, y) = p(0, y - x) + x$ and denoting $p(0, 1)$ by α, we conclude that

$$p(x, y) = x + \alpha \cdot (y - x) = \alpha \cdot y + (1 - \alpha) \cdot x.$$

Once we learn how to pool two values, we can pool four values easily:

- divide the four values into two pairs,
- pool results within each pair, and then
- pool the two pooling results into a single value.

It is reasonable to require that the result should not depend on how exactly we divide the four original values into pairs. Let us consider the values 0, 1, 1, and 2.

First, we combine 0 with 1 and 1 with 2. Pooling 0 and 1 results in

$$\alpha \cdot 1 + (1 - \alpha) \cdot 0 = \alpha.$$

Pooling 1 and 2 results in

$$\alpha \cdot 2 + (1 - \alpha) \cdot 1 = 2\alpha + 1 - \alpha = 1 + \alpha.$$

Here always $1 + \alpha$ is larger than α, so combining the results α and $1 + \alpha$ leads to

$$\alpha \cdot (1 + \alpha) + (1 - \alpha) \cdot \alpha = \alpha + \alpha^2 + \alpha - \alpha^2 = 2\alpha.$$

What if we instead combine 1 with 1 and 0 with 2? Combining 1 with 1 results in

$$\alpha \cdot 1 + (1 - \alpha) \cdot 1 = 1.$$

Pooling 0 with 2 results in

$$\alpha \cdot 2 + (1 - \alpha) \cdot 0 = 2\alpha.$$

The resulting of pooling the resulting too values 1 and 2α depends on which of the two values is larger.

- If $2\alpha \geq 1$, i.e., if $\alpha \geq 0.5$, then we get

$$\alpha \cdot (2\alpha) + (1 - \alpha) \cdot 1 = 2\alpha^2 + 1 - \alpha.$$

In this case, the desired equality is

$$2\alpha^2 + 1 - \alpha = 2\alpha,$$

i.e.,

$$2\alpha^2 - 3\alpha + 1 = 0.$$

One can easily check that this quadratic equation has two solutions: $\alpha = 0.5$ and $\alpha = 1$.

- If $2\alpha \le 1$, i.e., if $\alpha \le 0.5$, then we get

$$\alpha \cdot 1 + (1 - \alpha) \cdot 2\alpha = \alpha + 2\alpha - 2\alpha^2 = 3\alpha - 2\alpha^2.$$

In this case, the desired equality is

$$3\alpha - 2\alpha^2 = 2\alpha,$$

i.e.,

$$2\alpha^2 - \alpha = 0.$$

One can easily check that this quadratic equation has two solutions: $\alpha = 0$ and $\alpha = 0.5$.

So, we have three options: $\alpha = 0$, $\alpha = 0.5$, and $\alpha = 1$.

- If $\alpha = 0$, then the pooling formula takes the form $p(x, y) = 0 \cdot y + 1 \cdot x = x$, i.e., since $x \le y$, the form

$$p(x, y) = \min(x, y).$$

- If $\alpha = 0.5$, then the pooling formula takes the form $p(x, y) = 0.5 \cdot y + 0.5 \cdot x$, i.e., to the arithmetic average

$$p(x, y) = \frac{x + y}{2}.$$

- If $\alpha = 1$, then the pooling formula takes the form $p(x, y) = 1 \cdot y + 0 \cdot x = y$, i.e., since $x \le y$, the form

$$p(x, y) = \max(x, y);$$

see, e.g., [7].

These three operations – minimum, maximum, and arithmetic averge – are indeed the ones which work most successfully in deep learning [5].

12 Sensitivity of Deep Learning: Phenomenon

One of the problems with deep learning is that its results are often too sensitive to minor changes in the inputs. For example, changing a few pixels in a picture of a cat may result in this picture being misclassified as a dog; see, e.g., [5].

This is a very troublesome features, since in practice, signals often come with noise, and it is not good that a small noise can ruin the results.

13 Sensitivity of Deep Leaning: An Explanation

Each neuron is affected by the noise. It can take the original noise level δ and amplify it to a higher level $c \cdot \alpha$ for some $c > 1$.

In deep learning, we have several (L) layers. In the first layer, each neuron amplifies the noise level δ to $c \cdot \delta$. Neurons in the second layer amplify it even more, to $c \cdot (c \cdot \delta) = c^2 \cdot \delta$. After the third layer, we get $c^3 \cdot \delta$, etc., and after all L layers, we get $c^L \cdot \delta$. The exponential function c^L grows very fast with L, so, not surprisingly, we get a much higher noise level than for the traditional neural networks.

14 How to Deal with Sensitivity of Deep Learning

To train a traditional neural network, we feed it with actually observed patterns $\left(x_1^{(p)}, \ldots, x_n^{(p)}, y^{(p)}\right)$, andfind the values of the corresponding weights that match all these patterns. As a result, the trained network usually works well not only for the original patterns, but also for modified versions of these patterns – e.g., it still works well when we add some noise to the original pattern.

For deep learning, we do not have automatic success on noised patterns. So, to achieve such success, it is reasonable to artificially add noise to the patterns and add such simulated-noise modification to the original patterns when training a network. This idea seems to work reasonably well.

Acknowledgments

This work was supported in part by the US National Science Foundation grants 1623190 (A Model of Change for Preparing a New Generation for Professional Practice in Computer Science) and HRD-1242122 (Cyber-ShARE Center of Excellence).

References

[1] C. Baral, O. Fuentes, and V. Kreinovich, "Why Deep Neural Networks: A Possible Theoretical Explanation", In: M. Ceberio and V. Kreinovich (eds.), *Constraint Programming and Decision Making: Theory and Applications*, Springer Verlag, Berlin, Heidelberg, 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, 2019, to appear.

[4] A. Gholamy, J. Parra, V. Kreinovich, O. Fuentes, and E. Anthony, "How to best apply deep neural networks in geosciences: towards optimal 'averaging' in dropout training", In: J. Watada, Shing Chieng Tan, P. Vasant, E. Padmanabhan, and L. C. Jain (eds.), *Smart Unconventional/ Modelling, Simulation and Optimization for Geosciences and Petroleum Engineering*, Springer Verlag, 2019, pp. 15–26.

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

[6] V. Kreinovich, "From traditional neural networks to deep learning: towards mathematical foundations of empirical successes", In: S. N. Shahbazova, J. Kacprzyk, V. E. Balas, and V. Kreinovich (eds.) *Proceedings of the World Conference on Soft Computing*, Baku, Azerbaijan, May 29–31, 2018.

[7] V. Kreinovich and O. Kosheleva, *Optimization under Uncertainty Explains Empirical Success of Deep Learning Heuristics*, University of Texas at El Paso, Department of Computer Science, Technical Report UTEP-CS-19-49, 2019, http://www.cs.utep.edu/vladik/2019/tr19-49.pdf