5-2020

Towards Fast and Understandable Computations: Which "And"- and "Or"-Operations Can Be Represented by the Fastest (i.e., 1-Layer) Neural Networks? Which Activations Functions Allow Such Representations?

Kevin Alvarez
The University of Texas at El Paso, kalvarez9@miners.utep.edu

Julio Urenda
The University of Texas at El Paso, jcurenda@utep.edu

Orsoly Csiszár
Óbuda University, orsolya.csiszar@nik.uni-obuda.hu

Gábor Csiszár
University of Stuttgart, gabor.csiszar@mp.imw.uni-stuttgart.de

József Dombi
University of Szeged, dombi@inf.u-szeged.hu

Comments:
Technical Report: UTEP-CS-20-42

Towards Fast and Understandable Computations: Which "And"- and "Or"-Operations Can Be Represented by the Fastest (i.e., 1-Layer) Neural Networks? Which Activations Functions Allow Such Representations?

Kevin Alvarez[1], Julio C. Urenda[1,2], Orsolya Csiszár[3,4], Gábor Csiszár[5], József Dombi[6], György Eigner[4], and Vladik Kreinovich[1]
[1]Department of Computer Science
[2]Department of Mathematical Sciences
University of Texas at El Paso
El Paso, TX 79928, USA
kalvarez9@miners.utep.edu, jcurenda@utep.edu, vladik@utep.edu
[3]Faculty of Basic Sciences, University of Applied Sciences Esslingen
Esslingen, Germany
[4]Institute of Applied Mathematics, Óbuda University
Budapest, Hungary, orsolya.csiszar@nik.uni-obuda.hu
eigner.gyorgy@nik.uni-obuda.hu
[5]Institute of Materials Physics, University of Stuttgart
Stuttgart, Germany
gabor.csiszar@mp.imw.uni-stuttgart.de
[6]Institute of Informatics, University of Szeged
Szeged, Hungary, dombi@inf.u-szeged.hu

Abstract

We want computations to be fast, and we want them to be understandable. As we show, the need for computations to be fast naturally leads to neural networks, with 1-layer networks being the fastest, and the need to be understandable naturally leads to fuzzy logic and to the corresponding "and"- and "or"-operations. Since we want our computations to be both fast and understandable, a natural question is: which "and"- and "or"-operations of fuzzy logic can be represented by the fastest (i.e., 1-layer) neural network? And a related question is: which activation functions allow such a representation? In this paper, we provide an answer to both

questions: the only "and"- and "or"-operations that can be thus represented are $\max(0, a + b - 1)$ and $\min(a + b, 1)$, and the only activations functions allowing such a representation are equivalent to the rectified linear function – the one used in deep learning. This result provides an additional explanation of why rectified linear neurons are so successful. With also show that with full 2-layer networks, we can compute practically any "and"- and "or"-operation.

1 Formulation of the Problem

Computations are needed. In many application areas, we need to process data. Because of this need, computers are ubiquitous. What do we want from the computation results? First of all, we want them to be correct:

- if we are predicting weather, we want these predictions to be mostly successful,
- if we are deciding whether to give a loan to a bank's customer, we want to be sure that customers who get the loans have a high chance of repaying them, and that most customers to whom the program decided not to give the loan will not become very successful – and thus will not present our missed opportunities.

Coming up with such an algorithm is not easy, this is the main challenge. But once we have this algorithm, there are two other important challenges.

Two important challenges: computation speed and understandability. First, in most practical problems, we need to process a large amount of data – and we need to make a decision reasonably fast:

- if we predict weather, we need to take into account all the results of today's measurements of temperature, wind speed and direction, etc., in a given geographic areas, satellite images, historical data – and get the prediction of tomorrow's weather the same day: otherwise, our prediction will be useless;
- if we decide whether to give a person a loan, we need to take into account this person's financial history, financial history of similar customers, general economic situation in the region, etc. – and get the result fast, otherwise the customer may lose the business opportunity for which he/she is seeking this loan.

So, we need all the computations to be as fast as possible.

We also ideally want the computations to be understandable.

- When a weatherperson on the TV predict's tomorrow's weather, it is much more convincing if this person explains why we should expect strong winds, or, vice versa, perfect weather. These explanations may not be quantitative, usually, qualitative explanations are good enough.

- When we explain, to the person, why he/she is not getting a loan while his/her friends are, we need to have some reasonable explanations – at least to avoid lawsuits claiming gender-based, age-based, or race-based bias.

How can we achieve these two goals?

Need for fast computations leads to neural networks. A natural way to speed up computations is to perform them in parallel. In the past, only high-performance super-computers had several processors working in parallel, but nowadays, parallelism is ubiquitous: even the cheapest computers have up to four processors working in parallel. In parallel computations, all that matters is how fast computations can be performed on one of the processors – since computations on other processors are performed at the same time.

Which computations are fast? In general, computers process numbers, so, in general, any computation takes numerical inputs $x_1, \ldots, x_n$ – e.g., measurement results – and converts them into one or more numerical values y. In mathematics, a situation when to each input $x = (x_1, \ldots, x_n)$ there corresponds the result is known as a *function*, so we can say that each processor computes some function $y = f(x_1, \ldots, x_n)$.

Which functions are the easier to compute? Functions can be linear or non-linear. In general, linear functions, i.e., functions of the type

$$f(x_1, \ldots, x_n) = w_0 + w_1 \cdot x_1 + \ldots + w_n \cdot x_n$$

are the easiest to compute, so let us keep them in our list of easiest-to-compute functions. However, we cannot just limit ourselves to linear functions, because otherwise, if we only apply linear transformations, you will only get linear functions, but in real life, many dependencies are nonlinear. So, we need some nonlinear functions as well.

Which nonlinear functions are the easiest to compute? In general, the more inputs the function has, the longer it takes to process all these inputs. Thus, the easiest to compute are functions of one variable $y = s(z)$.

So, we arrive at the following computation scheme:

- first, each processor applies the fastest – linear – transformation to the data, i.e., computes the value $z = w_0 + w_1 \cdot x_1 + \ldots + w_n \cdot x_n$;
- if this is not enough, we apply the fastest non-linear transformation and compute $y = s(z)$; as a result, we get the value

$$y = s(w_0 + w_1 \cdot x_1 + \ldots + w_n \cdot x_n); \tag{1}$$

- then, if needed, we apply another linear transformation, then another nonlinear one, etc.

As a result, we get a layered computation scheme in which on each layer, each pair of processors computes the values (1), and then the results from these pairs become inputs to another layer, etc.

This scheme is what is usually known as a *neural network*; see, e.g., [2, 3, 5]. A two-part component computing the expression (1) is known as a *neuron*, and the non-linear function $s(z)$ is known as the *activation function*. So, the need for fast computations has indeed led us to neural networks. The fewer layers, the faster computations: 1-layer networks are the fastest, 2-layer networks are second fastest.

Neural networks have been very successful in practical applications, especially the currently popular *deep* neural networks. Deep neural networks use more layers, they also use a different activation function: traditionally, neural networks used *sigmoid* activation functions $s(z) = 1/(1 + \exp(-z))$, but lately, with the switch to deep neural networks, different activation functions are mostly used: *rectified linear* functions $s(z) = \max(0, z)$.

Need for understandability leads to fuzzy techniques. Understandability means that we should be able to describe the computations by using words from natural language. One of the main challenges in coming up with such a description is that natural language is imprecise (fuzzy), so it is difficult to find the relation between imprecise words from natural language and precise algorithms. In solving this challenge, it is natural to use the experience of researchers who came up with such a relationship from the other side of it: by trying to translate natural-language knowledge into precise terms.

This experience led to the design on fuzzy logic by Lotfi Zadeh; see, e.g., [1, 4, 6, 8, 9, 10]. Lotfi Zadeh, a specialist in control and an author of a successful textbook on control, noticed, in the early 1960s, a puzzling phenomenon: that human-led control often leads to much better results than even the optimal automatic control. The answer to this puzzle was clear: humans use additional knowledge which was not taken into account when the automatic controllers were designed. The reason why this additional knowledge was not taken into account is that this knowledge is not described in precise terms, it is described by using imprecise words from natural language. For example, an operator may say: if the pressure drops a little bit, increase a little bit the flow of the chemical into the chamber; here, "a little bit" does not have a precise meaning. Zadeh invented a methodology for translating this "fuzzy" knowledge into precise terms, a methodology that he called *fuzzy logic*, or, more generally, *fuzzy techniques*.

His main point is that in contrast to exact statements like "pressure is below 1.2 atmospheres" – which is always either true or false – about the statements that include natural-language words – like "the drop from 1.3 to 1.2 means that the pressure dropped a little bit" – experts are not sure. The smaller the drop, the larger the expert's degree of confidence that this statement is true. For each value of the corresponding quantity (e.g., pressure), we can gauge the expert's degree of confidence in the corresponding statement by asking the expert to mark it on a scale, e.g., from 0 to 10. The resulting mark depends on what scale we use: from 0 to 5 or from 0 to 10 or form 0 to any other number. To make these estimates uniform, a reasonable idea is to divide the mark by the largest number on the scale, so that, e.g., 7 on a scale from 0 to 10 becomes

$7/10 = 0.7$. In this new scale, 1 means that the expert is absolutely confident that this statement is true, 0 means that the expert is absolutely confident that the statement is false, and values between 0 and 1 correspond to intermediate degrees of confidence.

The reason why this methodology is called fuzzy *logic* is that in addition to simple statements – like the ones above – expert knowledge often contains statements that include *logical connectives* like "and" and "or". For example, an expert can recommend a certain action if the pressure dropped a little bit *and* the temperature increased somewhat. How can we gauge our degree of certainty in such composite statements? It would be great if we could similarly ask the expert to estimate his/her degree of confidence for all possible pairs of values (pressure, temperature). If we have a composite statement combining three or four different statements, we would need to consider all possible triples or quadruples. Even if we consider a reasonable number 20-30 of possible values of each quantity, it makes sense to ask the expert about all 30 values, but asking about all $30^4 = 810000$ possible quadruples is not realistic. Since we cannot directly elicit the degree of confidence in all such composite statements directly from the expert, we need to be able to estimate this degree based on whatever information we can elicit – i.e., based on the expert's degrees of confidence in the component statements.

In precise terms, we need a procedure that would take, as input, the degrees of confidence a and b in two statements A and B and return an estimate for the expert's degree of confidence in a composite statement $A \,\&\, B$. We will denote this estimate by $f_{\&}(a, b)$. The corresponding function $f_{\&}$ is known as an *"and"-operation*, or, for historical reason, a *t-norm*.

Since the statements "A and B" and "B and A" mean the same thing, it is reasonable to require that for these two statements, we have the same degree of confidence, i.e., that $f_{\&}(a, b) = f_{\&}(b, a)$. In other words, an "and"-operation must be commutative.

When A is false, clearly $A \,\&\, B$ is false too, so we must have $f_{\&}(0, b) = 0$ for all b. When A is true, our degree of confidence in $A \,\&\, B$ is the same as our degree of confidence in B, i.e., we must have $f_{\&}(1, b) = b$.

Similarly, we need a procedure that would take, as input, the degrees of confidence a and b in two statements A and B and return an estimate for the expert's degree of confidence in a composite statement $A \vee B$. We will denote this estimate by $f_{\vee}(a, b)$. The corresponding function $f_{\vee}$ is known as an *"or"-operation*, or, for historical reason, a *t-conorm*.

Since the statements "A or B" and "B or A" mean the same thing, it is reasonable to require that for these two statements, we have the same degree of confidence, i.e., that $f_{\vee}(a, b) = f_{\vee}(b, a)$. In other words, an "or"-operation must be commutative.

When A is true, clearly $A \vee B$ is true too, so we must have $f_{\vee}(1, b) = 1$ for all b. When A is false, our degree of confidence in $A \vee B$ is the same as our degree of confidence in B, i.e., we must have $f_{\vee}(0, b) = b$.

Natural questions. As we have mentioned earlier, we want our computations

to be both fast and understandable. Understandable means that we have to use some "and"- and "or'-operations. We thus want these operations to be fast. The fastest possible computations are computations on a 1-layer neural network, in which thus "and"-operation is computed by a single neuron, and in which the "or"-operation can also be computed by a single neuron. So, natural questions are:

- which "and"- and "or"-operations can be computed by a 1-layer neural network, and
- what activation functions allow computing "and"- and "or"-operations by such neural networks.

What we do in this paper. In this paper, we provide answers to both questions, namely:

- we show that the only "and"- and "or"-operations which can be computed by a 1-layer neural network are $\max(0, a + b - 1)$ and $\min(a + b, 1)$, and
- we show that the only activation function allowing such fast computations are equivalent to *rectified linear neurons* – which probably provides some explanations for the current success of such activation functions.

We also show that if we allow linear pre-processing after a single neuron, then we also represent $\min(a, b)$ and $\max(a, b)$. If we allow several neurons in a 2-layer network, then, in effect, we can compute any "and"- and "or"-operations.

2 Definitions and the Main Results

Definition 1. *By an* "and"-operation*, we mean a function*

$$f_{\&} : [0, 1] \times [0, 1] \to [0, 1]$$

for which the following properties are satisfied:

- $f_{\&}(a, b) = f_{\&}(b, a)$ *for all* a *and* b*,*
- $f_{\&}(0, b) = 0$ *and* $f_{\&}(1, b) = b$ *for all* b*.*

Definition 2. *By an* "or"-operation*, we mean a function*

$$f_{\vee} : [0, 1] \times [0, 1] \to [0, 1]$$

for which the following properties are satisfied:

- $f_{\vee}(a, b) = f_{\vee}(b, a)$ *for all* a *and* b*,*
- $f_{\vee}(0, b) = b$ *and* $f_{\vee}(1, b) = 1$ *for all* b*.*

Comment. Usually, for both "and"- and "or"-operations, other properties are required as well – namely, continuity, monotonicity, and associativity – but for our main results, we do not need these additional properties.

Definition 3. *We say that a function $f(x_1, \ldots, x_n)$ can be represented by a 1-layer neural network if this function can be represented in the form*

$$f(x_1, \ldots, x_n) = s(w_0 + w_1 \cdot x_1 + \ldots + w_n \cdot x_n)$$

for some function $s(z)$ and for some values w_i. The corresponding function $s(z)$ is called an activation function*.*

Definition 4. *By a* rectified linear *function, we mean a function*

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

Definition 5. *We say that two activation functions $s_1(z)$ and $s_2(z)$ are* equivalent *if for some constants a_{ij} and b_{ij}, we have*

$$s_1(z) = a_{10} + a_{12} \cdot s_2(b_{10} + b_{11} \cdot z) + a_{1z} \cdot z$$

and

$$s_2(z) = a_{20} + a_{21} \cdot s_1(b_{20} + b_{21} \cdot z) + a_{2z} \cdot z$$

for all z.

Comment. This way, the corresponding multi-layer neural networks represent, in effect, the same class of functions, since each non-linear layer is equivalent to adding extra linear transformations before and after the non-linear layer representing another activation function.

Proposition 1. *The only "and"-operation that can be represented by a 1-layer neural network is $\max(0, a + b - 1)$, and all activation functions allowing such a representation are equivalent to the rectified linear function.*

Proposition 2. *The only "or"-operation that can be represented by a 1-layer neural network is $\min(a + b, 1)$, and all activation functions allowing such a representation are equivalent to the rectified linear function.*

Comment. These results provide another explanation for why rectified linear activation functions are so successful in deep neural networks.

Proof of Proposition 1. Let us consider an "and"-operation $f_{\&}(a, b)$ which can be represented by a 1-layer neural network. By definition of such a representation, this means that $f_{\&}(a, b) = s(w_0 + w_a \cdot a + w_b \cdot b)$ for some function $s(z)$ and for some coefficients w_i.

By definition of an "and"-operation, we have $f_{\&}(a, b) = f_{\&}(b, a)$ for all a and b. Thus, the expression $s(w_0 + w_a \cdot a + w_b \cdot b)$ should not change if we swap

a and b: $s(w_0 + w_a \cdot a + w_b \cdot b) = s(w_0 + w_a \cdot b + w_b \cdot a)$. Therefore, we must have $w_a = w_b$, i.e., $f_{\&}(a, b) = s(w_0 + w_a \cdot a + w_a \cdot b)$, and thus,

$$f_{\&}(a, b) = s(w_0 + w_a \cdot (a + b)). \tag{1}$$

Let us introduce an auxiliary function $t(z) \stackrel{\text{def}}{=} s(w_0 + w_a \cdot z)$. This function is, by the definition of equivalence, equivalent to $s(z)$. In terms of this auxiliary function, the formula (1) takes the following simplified form:

$$f_{\&}(a, b) = t(a + b). \tag{2}$$

For $a = 0$, by definition of an "and"-operation, we have $f_{\&}(0, b) = 0$ for all $b \in [0, 1]$, thus $t(z) = 0$ for all $z \in [0, 1]$.

For $a = 1$, by definition of an "and"-operation, we have $f_{\&}(1, b) = b$ for all $b \in [0, 1]$, thus $t(1 + b) = b$ for all $b \in [0, 1]$. For $z = 1 + b$, we have $z \in [1, 2]$ and $b = z - 1$, thus $t(z) = z - 1$ for all $z \in [1, 2]$. So, we have:

- $t(z) = 0$ for $z \in [0, 1]$, and
- $t(z) = z - 1$ for $z \in [1, 2]$.

These two cases can be combined into a single formula

$$t(z) = \max(0, z - 1). \tag{3}$$

Substituting this expression for $t(z)$ into the formula (2), we conclude that $f_{\&}(a, b) = \max(0, a + b - 1)$. So, this "and"-operation is indeed the only one that can be represented by a 1-layer neural network.

Which activation functions can be used for this representation? From the formula (3), we can see that $t(z)$ is indeed equivalent to the rectified linear activation function. Since the original function $s(z)$ is equivalent to $t(z)$, we can conclude that $s(z)$ is also equivalent to the rectified linear activation function. Thus, the 1-layer representation of an "and"-operation is only possible if we use rectified linear neurons.

The proposition is proven.

Proof of Proposition 2. Let us now consider an "or"-operation $f_{\vee}(a, b)$ which can be represented by a 1-layer neural network. By definition of such a representation, this means that $f_{\vee}(a, b) = s(w_0 + w_a \cdot a + w_b \cdot b)$ for some function $s(z)$ and for some coefficients w_i.

By definition of an "or"-operation, we have $f_{\vee}(a, b) = f_{\vee}(b, a)$ for all a and b. Thus, the expression $s(w_0 + w_a \cdot a + w_b \cdot b)$ should not change if we swap a and b: $s(w_0 + w_a \cdot a + w_b \cdot b) = s(w_0 + w_a \cdot b + w_b \cdot a)$. Therefore, we must have $w_a = w_b$, i.e., $f_{\vee}(a, b) = s(w_0 + w_a \cdot a + w_a \cdot b)$, and thus,

$$f_{\vee}(a, b) = s(w_0 + w_a \cdot (a + b)). \tag{4}$$

Similar to the proof of Proposition 1, let us introduce an auxiliary function $t(z) \stackrel{\text{def}}{=} s(w_0 + w_a \cdot z)$. This function is, by the definition of equivalence, equivalent

to $s(z)$. In terms of this auxiliary function, the formula (4) takes the following simplified form:

$$f_\vee(a,b) = t(a+b). \tag{5}$$

For $a = 0$, by definition of an "or"-operation, we have $f_\vee(0,b) = b$ for all $b \in [0,1]$, thus $t(z) = z$ for all $z \in [0,1]$.

For $a = 1$, by definition of an "or"-operation, we have $f_\&(1,b) = 1$ for all $b \in [0,1]$, thus $t(1+b) = 1$ for all $b \in [0,1]$. For $z = 1+b$, we have $z \in [1,2]$ and $b = z-1$, thus $t(z) = 1$ for all $z \in [1,2]$. So, we have:

- $t(z) = z$ for $z \in [0,1]$, and
- $t(z) = 1$ for $z \in [1,2]$.

These two cases can be combined into a single formula

$$t(z) = \min(z, 1). \tag{6}$$

Substituting this expression for $t(z)$ into the formula (5), we conclude that $f_\vee(a,b) = \min(1, a+b)$. So, this "or"-operation is indeed the only one that can be represented by a 1-layer neural network.

Which activation functions can be used for this representation? One can easily see that the expression (6) can be represented in an equivalent form $t(z) = 1 - \max(1-z, 0)$, so $t(z)$ is indeed equivalent to the rectified linear activation function. Since the original function $s(z)$ is equivalent to $t(z)$, we can conclude that $s(z)$ is also equivalent to the rectified linear activation function. Thus, the 1-layer representation of an "or"-operation is only possible if we use rectified linear neurons.

The proposition is proven.

3 Two-Layer Networks and the Auxiliary Result

What about other "and"- and "or"-operations? In this paper, we have shown that only the operations $f_\&(a,b) = \max(0, a+b-1)$ and $f_\vee(a,b) = \min(a+b, 1)$ can be represented by 1-layer neural networks. How many layers do we need to represent general "and"- and "or"-operations?

It is known – see, e.g., [7] – that for every continuous "and"- (or "or"-) operation $f(a,b)$ and for every $\varepsilon > 0$, then exists a function $F(z)$ for which an "and"- (or, respectively, "or"-) operation

$$g(a,b) = F^{-1}(F(a) + F(b)) \tag{7}$$

satisfies the property $|f(a,b) - g(a,b)| \le \varepsilon$ for all a and b. (Of course, for this result to be true, it is not sufficient to have the above simplified definitions of "and"- and "or"-operations: we also need to assume associativity and monotonicity.)

For very small ε, the operations $f(a,b)$ and $g(a,b)$ are practically indistinguishable. So, from practical viewpoint, every "and"-operation and every

"or"-operation can be represented in the form (7). Every function of this form can be computed by a 2-layer neural network:

- in the first layer, we use the inputs a and b to compute the values $a' = F(a)$ and $b' = F(b)$;
- then, in the second layer, we compute the value $F^{-1}(a' + b')$, which is exactly the desired value $F^{-1}(F(a) + F(b))$.

So, from the practical viewpoint, every "and"-operation and every "or"-operation can be computed by a 2-layer neural network.

For example, a widely used "and"-operation $f_{\&}(a, b) = a \cdot b$ can be computed as $\exp(\ln(a) + \ln(b))$, with $F(z) = \ln(z)$ and the inverse function $F^{-1}(z) = \exp(z)$. Similarly, a widely used "or"-operation $f_{\vee}(a, b) = a + b - a \cdot b$ can be computed in the form (7) with $F(z) = \ln(1 - z)$ and $F^{-1}(z) = 1 - \exp(z)$.

When is it sufficient to have a single neuron with linear post-processing? We have shown that, from the practical viewpoint, all "and"- and "or"-operations can be represented by a 2-layer neural network. Interestingly, some "and"- and "or"-operations $f(a, b)$ can be represented by a single neuron if we allow an additional linear post-processing. For example, one can easily see that $\min(a, b) = b - \max(0, b - a)$ and $\max(a, b) = a + \max(0, b - a)$.

It turns out that these are the only "and"- and "or"-operations which can be thus represented.

Definition 6. *We say that a continuous monotonic associative "and"-operation* $f_{\&}(a, b)$ can be computed by a single neuron with linear post-processing *if we have*

$$f_{\&}(a, b) = c_0 + c_a \cdot a + c_b \cdot b + s(w_0 + w_a \cdot a + w_b \cdot b). \tag{8}$$

Definition 7. *We say that a continuous monotonic associative "or"-operation* $f_{\vee}(a, b)$ can be computed by a single neuron with linear post-processing *if we have*

$$f_{\vee}(a, b) = c_0 + c_a \cdot a + c_b \cdot b + s(w_0 + w_a \cdot a + w_b \cdot b). \tag{9}$$

Proposition 3. *The only "and"-operations that can be computed by a single neuron with linear post-processing are* $\max(0, a + b - 1)$ *and* $\min(a, b)$. *All activation functions allowing such a computation are equivalent to the rectified linear function.*

Proposition 4. *The only "or"-operations that can be computed by a single neuron with linear post-processing are* $\min(a + b, 1)$ *and* $\max(a, b)$. *All activation functions allowing such a computation are equivalent to the rectified linear function.*

Proof of Propositions 3 and 4. First of all, let us somewhat simplify the expressions (8) and (9) for the corresponding operation $f(a, b)$.

We cannot have $w_a = w_b = 0$ because then, the function $f(a,b)$ would be linear, and it is easy to show that no linear function can satisfy all the requirements of an "and"-operation or of an "or"-operation. Thus, either $w_a \neq 0$ or $w_b \neq 0$ (or both).

If $w_a = 0$, then, due to commutativity of $f(a,b)$, we can swap a and b and get an expression with $w_a \neq 0$. Thus, without losing generality, we can assume that $w_a \neq 0$.

We can thus introduce an auxiliary function $t(z) = c_0 + s(w_0 + w_a \cdot z)$. In terms of this auxiliary function, formulas (8) and (9) take the form

$$f(a,b) = c_a \cdot a + c_b \cdot b + t(a + k \cdot b), \tag{10}$$

where $k \stackrel{\text{def}}{=} w_b/w_a$.

If $k = 1$, then the expression $t(a + k \cdot b)$ is symmetric with respect to a and b. Since for both types of operations, the function $f(a,b)$ is commutative, we thus conclude that the difference

$$c_a \cdot a + c_b \cdot b = f(a,b) - t(a+b)$$

is also commutative. Therefore, $c_a = c_b$, hence the whole expression (10) depends only on the sum $a+b$, i.e., has the form $F(a+b)$ for some function $F(z)$. This means that each such function is computable by a 1-layer neural network, and all "and"- and "or"-operations which can be thus represented have been described in Propositions 1 and 2.

To complete the proof, it is therefore necessary to consider the case when $k \neq 1$, i.e., when the lines $a + k \cdot b = \text{const}$ are not parallel to the diagonal $a = b$ of the square $[0,1] \times [0,1]$. Each line $a + k \cdot b = \text{const}$ intersects the borderline of the square at two points. On the borderline – i.e., when one of the values a and b is equal to 0 or to 1 – the value of an "and"- or "or"-operation is uniquely determined by the corresponding Definition (Definition 1 or Definition 2). Since the function $f(a,b)$ is linear on this line, its values for all the points from this line are uniquely determined by the values at these two borderline points. Thus, for each k, we uniquely determine all the values $f(a,b)$ for all the pairs (a,b).

One can check that the only case when the resulting function is commutative and associative is the case $k = -1$, in which case we indeed get $\min(a,b)$ and $\max(a,b)$. We can also easily check that in both case, the activation function $t(z)$ is indeed equivalent to the rectified linear function. The propositions are proven.

Remaining open problems. It is known (see, e.g., [2]) that functions represented as linear combinations of the results of 1-neuron layer are universal approximators – i.e., for each continuous function on a bounded domain and for each accuracy $\varepsilon > 0$, we can find a neural network which computes the given function with the desired accuracy. In general, the more accuracy we require, the more neurons we need. So, to achieve perfect accuracy – i.e., exact computations – we will need potentially infinite number of neurons. However, for some "and"- and "or"-operations, we can have perfect accuracy with a limited

number of neurons: e.g., the operation $a \cdot b$ can be computed by a 2-neuron network, as

$$a \cdot b = \frac{1}{4} \cdot (a+b)^2 - \frac{1}{4} \cdot (a-b)^2.$$

The operation $a + b - a \cdot b$ can be computed by a 3-neuron network:

$$a + b - a \cdot b = (a+b) - \frac{1}{4} \cdot (a+b)^2 - \frac{1}{4} \cdot (a-b)^2.$$

It would be interesting to describe all such "and"- and "or"-operations. Maybe $a \cdot b$ and $a + b - a \cdot b$ are the only such operations?

Acknowledgments

This work was supported in part by the grant TUDFO/47138-1/2019-ITM from the Ministry of Technology and Innovation, Hungary, and 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] R. Belohlavek, J. W. Dauben, and G. J. Klir, *Fuzzy Logic and Mathematics: A Historical Perspective*, Oxford University Press, New York, 2017.

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

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

[4] G. Klir and B. Yuan, *Fuzzy Sets and Fuzzy Logic*, Prentice Hall, Upper Saddle River, New Jersey, 1995.

[5] V. Kreinovich and O. Kosheleva, "Deep learning (partly) demystified", *Proceedings of the 4th International Conference on Intelligent Systems, Metaheuristics & Swarm Intelligence ISMSI'2020*, Thimpu, Bhutan, April 18–19, 2020.

[6] J. M. Mendel, *Uncertain Rule-Based Fuzzy Systems: Introduction and New Directions*, Springer, Cham, Switzerland, 2017.

[7] H. T. Nguyen, V. Kreinovich, and P. Wojciechowski, "Strict Archimedean t-norms and t-conorms as universal approximators", *International Journal of Approximate Reasoning*, 1998, Vol. 18, Nos. 3–4, pp. 239–249.

[8] H. T. Nguyen, C. L. Walker, and E. A. Walker, *A First Course in Fuzzy Logic*, Chapman and Hall/CRC, Boca Raton, Florida, 2019.

[9] V. Novák, I. Perfilieva, and J. Močkoř, *Mathematical Principles of Fuzzy Logic*, Kluwer, Boston, Dordrecht, 1999.

[10] L. A. Zadeh, "Fuzzy sets", *Information and Control*, 1965, Vol. 8, pp. 338–353.