Training Neural Networks on Interval Data: Unexpected Results and Their Explanation

Edwin Tomy George, Luc Jaulin, Vladik Kreinovich,
Christoph Lauter, and Martine Ceberio

Abstract In many practically useful numerical computations, training-and-then-using a neural network turned out to be a much faster alternative than running the original computations. When we applied a similar idea to take into account interval uncertainty, we encountered two unexpected results: (1) that while for numerical computations, it is usually better to represent an interval by its midpoint and half-width, for neural networks, it is more efficient to represent an interval by its endpoints, and (2) that while usually, it is better to train a neural network on the whole data processing algorithm, in our problems, it turned out to be more efficient to train several subnetworks on subtasks and then combine their results. In this paper, we provide a theoretical explanation for these unexpected results.

1 Formulation of the Problem

We need faster computations. Modern computers are very fast, but for some practical problems, they are still not fast enough. For example, modern high-performance computers provide a reasonable accurate prediction of tomorrow's weather – a few hours on a high performance computer, and we have a good understanding of the next day's weather.

In principle, similar algorithms can also predict extreme weather – e.g., they can predict in what direction a tornado will move in the next 15 minutes. However, a tornado is a very fast process, what usually takes a day to change for the case of regular weather happens in 15 minutes for a tornado. As a result, we need the same

Edwin Tomy George, Vladik Kreinovich, Christoph Lauter, and Martine Ceberio
Department of Computer Science, University of Texas at El Paso, 500 W. University
El Paso, Texas 79968, USA
e-mail: etomygeorg@miners.utep.edu, vladik@utep.edu, cqlauter@utep.edu, mceberio@utep.edu

Luc Jaulin
Lab-STICC, ENSTA-Bretagne, Brest, 29200, Finisterre, France, e-mail: lucjaulin@gmail.com

few hours on a high performance computer to predict where a tornado will turn in the next 15 minutes – as to predict next day;s weather. For weather, this prediction makes sense, we still get the result before the next day. However, for a tornado, such "predictions" make no sense: in 15 minutes, way before the computations finish, we will know where the tornado went :-(

There are many practical problems like that, that show that we need faster computations.

A natural idea how to speed up computations. In many engineering situations, a good idea is to look how nature has solved the corresponding problem. After all, what we see in life creatures is the result of billions of years of optimizing evolution. If there was a better solution, in billions of years nature would have found it – and the corresponding creature would have out-competed everyone else.

So, to see how we can speed up data processing, a natural idea is to look how living creatures process data. In living creatures, data is usually processed by neurons. The results are not bad. For many practical problems like face recognition, we human are often as good as (or even better than) modern computers. This may not sound that unexpected until we realize that a computer consists of components that perform several billion operations per second, while the fastest neurons (data processing cells in the human brain) can perform at most 100 (and usually much fewer) operations per second.

The reason why human brain, with its relatively very slow components, can compete with computers (whose components are millions times faster) is that data processing in a human brain is heavily parallelized, with up to billions of neurons – each of which performs a very simple operation – working in parallel.

So, a natural idea to speed up computations is to simulate how the brain works. This is exactly what artificial neural networks – usually called just neural networks – do; see, e.g., [1]. In the last decades, neural networks achieved enormous successes. Training them requires a lot of time – this is npt surprising, since training an infant to recognize faces and to perform other tasks takes several years. However, once the network is trained, it is usually very fast.

In many case, neural networks are used for situations of machine learning, when we do not know the data processing algorithm a priori, so we need to reconstruct this algorithm based on several input-output examples. However, neural networks have also been very helpful in solving numerical problems for which algorithms are known. In such cases, we train a neural network on many input-output examples generated by the known algorithm. This training takes a significant amount of time, but once the network is trained, it produced the results very fast, usually much faster then the original numerical algorithm.

Need for uncertainty quantification. In data processing, we inputs some values $x_1, \ldots, x_n$ into an algorithm f to compute the desired output

$$y = f(x_1, \ldots, x_n).$$

At first glance, it may seem that this is all we need: e.g., we input the values x_i of today's temperature, humidity, wind speed at different locations, and we get an

estimate y for tomorrow's weather. But what we also need to know is how accurate is our estimate.

There are several reasons why our estimate is different from the actual value of the desired quantity. In many case, the algorithm itself $y = f(x_1, \ldots, x_n)$ is approximate. However, even when the algorithm is exactly describing the relation between the actual values of the corresponding physical quantities x_i and y, the result of data processing is still only approximate – because:

- what we process are not the actual values x_i of the physical quantities, but the results $\widetilde{x}_i$ of measuring these quantities, and
- these results are, in general, somewhat different from the actual values; see, e.g., [8].

Thus, the value $\widetilde{y} = f(\widetilde{x}_1, \ldots, \widetilde{x}_n)$ obtained by processing the measurement results $\widetilde{x}_i$ is, in general, different from the ideal value $y = f(x_1, \ldots, x_n)$ that we would have got if we knew the exact values x_i.

It is therefore desirable to analyse how the measurement errors $\Delta x_i \stackrel{\text{def}}{=} \widetilde{x}_i - x_i$ affect the result of data processing, i.e., what can we say about the resulting uncertainty $\Delta y \stackrel{\text{def}}{=} \widetilde{y} - y$. This analysis is known as *uncertainty quantification*.

Need for interval computations. In some cases, we know the range of possible values of each measurement error Δx_i, and we know how frequent are different values from this range – i.e., we know the probability distribution of each measurement error. In this case, in principle, we can perform uncertainty quantification by Monte-Carlo techniques – i.e., by simulating all these probability distributions. However, there are two important cases when we do not know these probability distributions.

The first such case is state-of-the-art measurements. Indeed, usually, we get the probability distribution of the measurement error by comparing the results of measuring some quantities by our measuring instrument and by another, much more accurate measuring instrument – whose measurement error are much smaller than of our instrument and can, thus, be safely ignored. This process is known as the *calibration* of the measureing instrument. However, for the state-of-the art measurements, there is no more accurate instrument – the one we have is the best. In this case, we cannot determine the probability distribution. The best we can do is find the upper bound Δ_i on the absolute value of the measurement error: $|\Delta x_i| \le \Delta_i$.

The second case is routine measurements on the shop floor. In principle, we could calibrate every sensor, every measurement instrument. However, sensors are now cheap – e.g., school kids use cheap distance-measuring sensors to design robots – while calibration requires the use of expensive high-accuracy measurement instruments and is, thus, much more expensive than the sensor itself. Because of this cost, most measuring instruments are not fully calibrated. The best we can do is use the upper bound Δ_i provided by the instrument's manufacturer. And the manufacturer has to supply some such bound – otherwise, if there is no guaranteed bounds on the measurement error, this means that any actual value if possible, so this cannot be called a measuring instrument.

In both cases, if all we know about the measurement error is the upper bound Δx_i, then after we get the measurement result $\widetilde{x}_i$, the only thing we can conclude about the actual value x_i of the corresponding quantity is that it is contained in the interval $[\underline{x}_i, \overline{x}_i]$, where $\underline{x}_i = \widetilde{x}_i - \Delta_i$ and $\overline{x}_i = \widetilde{x}_i + \Delta_i$. In this case, all we know about $y = f(x_1, \ldots, x_n)$ is that y belongs to the interval

$$[\underline{y}, \overline{y}] \stackrel{\text{def}}{=} \{f(x_1, \ldots, x_n) : x_i \in [\underline{x}_i, \overline{x}_i]\}.$$

Computing this interval $[\underline{y}, \overline{y}]$ is one of the main tasks of *interval computations*; see, e.g., [2, 4, 6, 7].

Need for interval computations adds complexity to data processing. It is known – see, e.g., [3] – that in many cases, interval computations are more computationally complex than the corresponding data processing. For example, for quadratic functions $f(x_1, \ldots, x_n)$, computation is straightforward, but the corresponding interval computation problem is NP-hard; see, e.g., [3]. Thus, for interval-related problems, there is even more need to speed up computations than for data processing in general.

One way to speed up computations is to take into account that the more bits we process, the more time we need for this processing, Since the measurement error is usually much smaller than the measurement result, we therefore need fewer bits to represent the bound Δ_i than to represent measurement values. Thus, usually, in interval computations, an interval is represented by its midpoint $\widetilde{x}_i$ and its half-width Δ_i – rather than by its endpoints.

The use the midpoint-half-width speeds up computations somewhat. However, in many practical situations, the resulting speed up is not sufficient, so using a neural network to further speed up interval computations is a reasonable idea.

What we did and what were the unexpected results. We did train a neural network to solve an interval computation problem – namely, a problem of locating a robot based on distance and angle measurements; see, e.g., [10, 11]. Overall, the results are good, but some results were unexpected.

First:

- while usually in interval computations, a representation of an interval by its midpoint and half-width leads to better results,
- for neural networks, we got better results when we represented an interval by its endpoints.

Second:

- while usually in machine learning, one gets better results training the neural network on the whole algorithm,
- in our case, we got better result when we trained neural networks on some modules, and then combined the results.

What we do in this paper. In this paper, we provide a theoretical explanation for both unexpected results.

2 Why for Neural Networks, Endpoint Representation Works Better

Why is there a difference in the first place: question. At first glance, it is not clear why there is a difference between the two possible interval representations at all. Indeed, in a usual neural network, the only neurons that process the measurement results are neurons from the first layer – all other neurons process the output signals of the neurons from the previous layer.

In general, in a neural network, a neuron that inputs the input values $v_1, \ldots, v_n$ produces an output signal $z = s(a_0 + a_1 \cdot v_1 + \ldots + a_N \cdot v_N)$, for some non-linear function $s(t)$ – which is called an *activation function*. So, if we input endpoints of n intervals, we get

$$s(a_0 + a_1 \cdot \underline{x}_1 + b_1 \cdot \overline{x}_1 + \ldots + a_n \cdot \underline{x}_n + b_n \cdot \overline{x}_n).$$

If we input midpoint and half-width of each interval, we get a similar expression:

$$s(a_0 + c_1 \cdot \widetilde{x}_1 + d_1 \cdot \Delta_1 + \ldots + c_n \cdot \widetilde{x}_n + d_n \cdot \Delta_n).$$

Since $\underline{x}_i$ and $\overline{x}_i$ are linear combinations of the values $\widetilde{x}_i$ and Δ_i, the corresponding classes of functions are the same:

- When we know a_i and b_i, then we get

  $$a_i \cdot \underline{x}_i + b_i \cdot \overline{x}_i = a_i \cdot (\widetilde{x}_i - \Delta_i) + b_i \cdot (\widetilde{x}_i + \Delta_i) = (a_i + b_i) \cdot \widetilde{x}_i + (b_i - a_i) \cdot \Delta_i,$$

  i.e., we have the equivalent second expression with $c_i = a_i + b_i$ and $d_i = b_i - a_i$.
- Vice versa, from $\underline{x}_i = \widetilde{x}_i - \Delta_i$ and $\overline{x}_i = \widetilde{x}_i + \Delta_i$, we get

  $$\widetilde{x}_i = \frac{\underline{x}_i + \overline{x}_i}{2} \text{ and } \Delta_i = \frac{\overline{x}_i - \underline{x}_i}{2}$$

  and thus,

  $$c_i \cdot \widetilde{x}_i + d_i \cdot \Delta_i = c_i \cdot \frac{\underline{x}_i + \overline{x}_i}{2} + d_i \cdot \frac{\overline{x}_i - \underline{x}_i}{2} = \frac{c_i - d_i}{2} \cdot \widetilde{x}_i + \frac{c_i + d_i}{2} \cdot \Delta_i,$$

  i.e., we have the equivalent first expression with

  $$a_i = \frac{c_i - d_i}{2} \text{ and } b_i = \frac{c_i + d_i}{2}.$$

Since we have the same class of approximating functions in both case, why is there a difference in the training results?

Why is there a difference: explanation. The formulas are the same, so the training iterations are, in effect, equivalent, but what is different is the initial values of the corresponding weights. Usually, the initial weights are selected randomly. For example, we start with the same probability distribution for each weight – usually, a

normal distribution with 0 mean and standard deviation σ_0 – and select each weight independently. This way, there *is* a difference between the two representations:

- If we select a_i and b_i as independent normally distributed random variables with mean 0 and standard deviation σ_0, then, as one can easily check, the combinations $c_i = a_i + b_i$ and $d_i = a_i - b_i$ are also independent, normally distributed with mean 0 and standard deviation $\sqrt{2} \cdot \sigma_0$.
- Similarly, if we select c_i and d_i as independent normally distributed random variables with mean 0 and standard deviation σ_0, then, as one can easily check, the combinations
$$a_i = \frac{c_i - d_i}{2} \text{ and } b_i = \frac{c_i + d_i}{2}$$
are also independent, normally distributed with mean 0 and standard deviation
$$\frac{1}{\sqrt{2}} \cdot \sigma_0.$$

When standard deviation is smaller, the resulting random number is, on average, smaller, so we need fewer bits to represent this number. Thus, as we have mentioned earlier, we need smaller computation time.

This explains why for a neural network, using the endpoints – which corresponds to using coefficients a_i and b_i – leads to slightly faster training for the same accuracy or, equivalently, slightly more accurate training for the same training time. In other words, this explains the first unexpected result.

3 Why Training for Modules Works Better

In general, the simpler the function, the easier is to train a neural network on this function, and the fewer parameters we need to approximate this function with a given accuracy. The complexity of a function is usually measured by the length of the corresponding program; see, e.g., [5]. Thus, a function obtained by applying some operation to two equal-size modules is approximately twice more complex that each of its modules.

For each function, training means finding the values of the weights for which the values produced by the neural network are approximately equal to the desired results. In statistics, finding parameters that match the known input-output pairs is known as *regression*. It is known – see, e.g., [9] – that for the same level of noise, the more parameters we need to determine, the less accurate is our estimation of these parameters. This comes from the fact that in the linear approximation, matching means solving a system of linear equations $Xa = y$, where X is a matrix formed by results of measure x_i, and a are the desired weights.

So we need to compute $a = X^{-1}y$. In the basis formed by eigenvectors of the matrix X, this means that $a_i = y_i/\lambda_i$, where λ_i is the corresponding eigenvalue. For random matrices, the smallest eigenvalue is close to 0, and it tends to 0 as the matrix

size increases. Thus, indeed, the more parameters, the larger the matrix size, the larger the resulting uncertainty in a_i – and thus, the resulting uncertainty in the final estimate.

Since, as we have mentioned, modules require fewer parameters than the algorithm as a whole, their training leads to more accurate results than training for the whole algorithm. If we then combine the two more-accurate module estimates, we thus get a more accurate estimate for the overall result of data processing – this is exactly what we observed.

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), HRD-1834620 and HRD-2034030 (CAHSI Includes), EAR-2225395 (Center for Collective Impact in Earthquake Science C-CIES), and by the AT&T Fellowship in Information Technology.

It was also supported by a grant from the Hungarian National Research, Development and Innovation Office (NRDI), and by the Institute for Risk and Reliability, Leibniz Universitaet Hannover, Germany.

References

1. I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, Cambridge, Massachusetts, 2016.
2. L. Jaulin, M. Kiefer, O. Didrit, and E. Walter, *Applied Interval Analysis, with Examples in Parameter and State Estimation, Robust Control, and Robotics*, Springer, London, 2012.
3. V. Kreinovich, A. Lakeyev, J. Rohn, and P. Kahl, *Computational Complexity and Feasibility of Data Processing and Interval Computations*, Kluwer, Dordrecht, 1998.
4. B. J. Kubica, *Interval Methods for Solving Nonlinear Constraint Satisfaction, Optimization, and Similar Problems: from Inequalities Systems to Game Solutions*, Springer, Cham, Switzerland, 2019.
5. M. Li and P. Vitanyi, *An Introduction to Kolmogorov Complexity and Its Applications*, Springer, Cham, Switzerland, 2019.
6. G. Mayer, *Interval Analysis and Automatic Result Verification*, de Gruyter, Berlin, 2017.
7. R. E. Moore, R. B. Kearfott, and M. J. Cloud, *Introduction to Interval Analysis*, SIAM, Philadelphia, 2009.
8. S. G. Rabinovich, *Measurement Errors and Uncertainty: Theory and Practice*, Springer Verlag, New York, 2005.
9. D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*, Chapman and Hall/CRC, Boca Raton, Florida, 2011.
10. E. Tomy George, *Neural networks for decisions under uncertainty*, Master's Thesis, Department of Computer Science, University of Texas at El Paso, El Paso, Texas, USA, 2024.
11. E. Tomy George, L. Jaulin, V. Kreinovich, C. Lauter, and M. Ceberio, "Localizing robots using neural networks with interval data", *Proceedings of the 10th International Workshop on Reliable Engineering Computing REC'2024*, Beijing, China, October 26–27, 2024.