Why Deep Learning Is More Efficient than Support Vector Machines, and How It Is Related to Sparsity Techniques in Signal Processing

Laxman Bokati[1], Olga Kosheleva[1], Vladik Kreinovich[1], and Anibal Sosa[2]
[1]University of Texas at El Paso
El Paso, Texas 79968, USA
lbokati@miners.utep.edu, olgak@utep.edu, vladik@utep.edu
[2]Facultad de Ingienería
Universidad Icesi
Calle 18 No. 122-135 Pance
Cali, Colombia, hannibals76@gmail.com

Abstract

Several decades ago, traditional neural networks were the most efficient machine learning technique. Then it turned out that, in general, a different technique called support vector machines is more efficient. Reasonably recently, a new technique called deep learning has been shown to be the most efficient one. These are empirical observations, but how we explain them – thus making the corresponding conclusions more reliable? In this paper, we provide a possible theoretical explanation for the above-described empirical comparisons. This explanation enables us to explain yet another empirical fact – that sparsity techniques turned out to be very efficient in signal processing.

1 Formulation of the Problem

Main objectives of science and engineering. We want to make our lives better, we want to select actions and designs that will make us happier, we want to improve the world so as to increase our happiness level. To do that, we need to know what is the current state of the world, and what changes will occur if we perform different actions. Crudely speaking, learning the state of the world and learning what changes will happen is the main objective of science, while using this knowledge to come up with the best actions and best designs is the main objective of engineering.

Need for machine learning. In some cases, we already know how the world

operates: e.g., we know that the movement of the celestial bodies is well described by Newton's equations – it is described so well that we can predict, e.g., Solar eclipses centuries ahead. In many other cases, however, we do not have such a good knowledge, we need to extract the corresponding laws of nature from the observations.

In general, prediction means that we can predict the future value y of the physical quantity of interest based on the current and past values $x_1, \ldots, x_n$ of related quantities. To be able to do that, we need to have an algorithm that, given the values $x_1, \ldots, x_n$, computes a reasonable estimate for the desired future value y.

In the past, designing such algorithms was done by geniuses – Newton described how to predict the motion of celestial bodies, Einstein provided more accurate algorithms, Schroedinger, in effect, described how to predict probabilities of different states of the quantum system, etc. This still largely remains the domain of geniuses, Nobel prizes are awarded every year for these discoveries. However, now that the computers has become very efficient, they are often used to help. This use of computers is known as *machine learning*: when we know, in several cases $c = 1, \ldots, C$, which values $y^{(c)}$ corresponded to appropriate values $x_1^{(c)}, \ldots, x_n^{(c)}$, and we want to find an algorithm $f(x_1, \ldots, x_n)$ for which, for all these cases c, we have $y^{(c)} \approx f(x_1^{(c)}, \ldots, x_n^{(c)})$.

The value y may be tomorrow's temperature in a given area, it may be a binary (0-1) variable deciding whether a given email is legitimate or a spam (or whether, e.g., the given image is an image of a cat).

Machine learning: a brief history. One of the first successful general machine learning techniques was the technique of *neural networks*; see, e.g., [3]. In this technique, we look for algorithms of the type

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

for some non-linear function $s(z)$ called an *activation function*, and for some values w_{ki} and W_k knows as *weights*. As the function $s(z)$, researchers usually selected the so-called *sigmoid function* $s(z) = \dfrac{1}{1 + \exp(-z)}$.

This algorithm emulates a 3-layer network of biological neurons – the main cells providing data processing in our brains. In the first layer, we have input neurons that read the inputs $x_1, \ldots, x_n$. In the second layer – called a *hidden layer* – we have K neurons each of which first generates a linear combination

$$z_k = \sum_{i=1}^{n} w_{ki} \cdot x_i - w_{k0}$$

of the input signals, and the applies an appropriate nonlinear function $s(z)$ to this combination, resulting in a signal $y_k = s(z_k)$. The processing by biological neurons is well described by the sigmoid activation function – this is the reason

why this function was selected for artificial neural networks in the first place. After that, in the final output layer, the signals y_k from the neurons in the hidden layer are combined into a linear combination $\sum\limits_{k=1}^{K} W_k \cdot y_k - W_0$ which is returned as the output.

A special efficient algorithm – known as *backpropagation* – was developed to *train* the corresponding neural network, i.e., to find the values of the weights that provide the best fit for the observation results $x_1^{(c)}, \ldots, x_n^{(c)}, y^{(c)}$.

Support Vector Machines: a brief description. Later, in many practical problem, a different technique became more efficient: the technique of *Support Vector Machines*; see, e.g., [29] and references therein. Let us explain this technique on the example of a *binary classification* problem, i.e., a problem in which we need to classify all objects (or events) into one of two classes, based on the values $x_1, \ldots, x_n$ of the corresponding parameters – i.e., in which the desired output y has only two possible values.

In general, if, based on the values $x_1, \ldots, x_n$ we can uniquely determine to which of the two classes this object belongs, this means that the set of all possible values of the tuple $x = (x_1, \ldots, x_n)$ is divided into two non-intersecting sets S_1 and S_2 corresponding to each of the two classes.

We can therefore come up with a continuous function $f(x_1, \ldots, x_n)$ such that $f(x) \geq 0$ for $x \in S_1$ and $f(x) \leq 0$ for $x \in S_2$. As an example of such a function, we can take $f(x) = d(x, S_2) - d(x, S_1)$, where the distance $d(x, S)$ between a point x and the set S is defined as the distance from x to the closest point of S, i.e., as $\inf\limits_{s \in S} d(x, s)$. Clearly, if $x \in S$, then $d(x, s) = 0$ for $s = x$ thus $d(x, S) = 0$.

- For points $x \in S_1$, we have $d(x, S_1) = 0$ but usually $d(x, S_2) > 0$, thus $f(x) = d(x, S_2) - d(x, S_1) > 0$.
- On the other hand, for points $x \in S_2$, we have $d(x, S_2) = 0$ while, in general, $d(x, S_1) > 0$, thus $f(x) = d(x, S_2) - d(x, S_1) < 0$.

In some simple cases, there exists a linear function

$$f(x_1, \ldots, x_n) = a_0 + \sum_{i=1}^{n} a_i \cdot x_i$$

that separates the two classes. In this case, there exist efficient algorithms for finding the corresponding coefficients a_i – for example, we can use linear programming (see, e.g., [9, 31]) to find the values a_i for which:

- $a_0 + \sum\limits_{i=1}^{n} a_i \cdot x_i > 0$ for all known tuples $x \in S_1$, and
- $a_0 + \sum\limits_{i=1}^{n} a_i \cdot x_i < 0$ for all known tuples $x \in S_2$.

In many practical situations, however, such a linear separation is not possible. In such situations, we can take into account the known fact that any continuous function on a bounded domain (and for practical problems, there are always bounds on the values of all the quantities) can be approximated, with any given accuracy, by a polynomial. Thus, with any given accuracy, we can separate the two classes by checking whether the f-approximating polynomial

$$P_f(x) = a_0 + \sum_{i=1}^{n} a_i \cdot x_i + \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij} \cdot x_i \cdot x_j + \ldots$$

is positive or negative.

In other words, if we perform a non-linear mapping of each original n-dimensional point $x = (x_1, \ldots, x_n)$ into a higher-dimensional point

$$X = (X_1, \ldots, X_n, X_{11}, X_{12}, \ldots, X_{nn}, \ldots) = (x_1, \ldots, x_n, x_1^2, x_1 \cdot x_2, \ldots, x_n^2, \ldots),$$

then in this higher-dimensional space, the separating function becomes linear:

$$P_f(X) = a_0 + \sum_{i=1}^{n} a_i \cdot X_i + \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij} \cdot X_{ij} + \ldots,$$

and we know how to effectively find a linear separation.

Instead of polynomials, we can use another basis $e_1(x)$, $e_2(x)$, $\ldots$, to approximate a general separating function as $a_1 \cdot e_1(x) + a_2 \cdot e_2(x) + \ldots$

The name of this technique comes from the fact that when solving the corresponding linear programming problem, we can safely ignore many of the samples and concentrate only on the vectors X which are close to the boundary between the two sets – if we get linear separation for such *support vectors*, we will automatically get separation for other vectors X as well.

This possibility to decrease the number of iterations enables us to come up with algorithms for the SVM approach which are more efficient than general linear programming algorithms – and many other ideas an tricks help make the resulting algorithms even faster.

Deep learning: a brief description. Lately, the most efficient machine learning tool is *deep learning*; see, e.g., [19]. Deep learning is a version of a neural network, but the main difference is that instead of a large number of neurons in a hidden layer, we have multiple layers with a relatively small number of neurons in each of them.

Similarly to the traditional neural networks, we start with the inputs x_1, $\ldots$, x_n. These inputs serve as inputs $x_i^{(0)}$ to the neurons in the first later. On each layer k, each neuron takes, as inputs, outputs $x_i^{(k-1)}$ from the previous layer and returns the value $x_j^{(k)} = s_k\left(\sum_i w_{ij}^{(k)} \cdot x_i^{(k-1)}\right) - w_{0j}^{(k)}$. For most layers, instead of the sigmoid, it turns out to be more efficient to use a piece-wise linear function $s_k(x) = \max(x, 0)$ which is 0 for $x < 0$ and equal to x for $x > 0$. In the last layer, sometimes, the sigmoid is used.

There are also layers in which inputs are divided into groups, and we combine inputs from each group into a single value – e.g., by taking the maximum of the corresponding values.

In addition to the general backpropagation idea, several other techniques are used to speed up the corresponding computations – e.g., instead of using *all* the neurons in training, one of the techniques is to only use, on each iteration, *some* of the neurons and then combine the results by applying an appropriate combination functions (which turns out to be geometric mean).

Natural questions. So far, we have described what happened: support vector machines turned out to be more efficient in machine learning, and deep learning is, in general, more efficient than support vector machines. A natural question is: why? How can we theoretically explain these empirical facts – thus increasing our trust in the corresponding conclusions?

What we do in this paper. In our previous papers, we explained why deep learning is more efficient than the traditional neural networks; see, e.g., [2, 20, 21]. (We also explained the selection of piece-wise linear activation functions [17], why some combination functions are more efficient [18], and several other features of deep learning [20].)

In this paper, we extend these explanations to the comparison between support vector machines and neural networks.

The resulting explanation will help us understand yet another empirical fact – the empirical efficiency of sparse techniques in signal processing.

2 Why Support Vector Machines Are, in General, More Efficient than Traditional Neural Networks: An Explanation

This empirical comparison is the easiest to explain. Indeed, to train a traditional neural network on the given cases $x_1^{(c)}, \ldots, x_n^{(c)}, y^{(c)}$, we need to find the weights W_k and w_{ki} for which

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

Here, the activation function $s(z)$ is non-linear, so we have a system of non-linear equations for finding the corresponding weights W_k and w_{ki}. In general, solving a system of nonlinear equations is NP-hard even for quadratic equations; see, e.g., [23, 27].

In contrast, for support vector machines, to find the corresponding coefficients a_i, it is sufficient to solve a linear programming problem – and this can be done in feasible time. This explains why support vector machines are more efficient than traditional neural networks.

3 Why Deep Learning Is, in General, More Efficient than Support Vector Machines: An Explanation

At first glance, the above explanation should work for the comparison between support vector machines and deep networks: in the first case, we have a feasible algorithm, while in the second case, we have an NP-hard problem that may require very long (exponential) time.

However, this is only at first glance. Namely, the above comparison assumes that all the inputs $x_1, \ldots, x_n$ are independent – in the sense of functional dependency, i.e., that none of them can be described in terms of one another. In reality, most inputs are dependent in this sense. This is especially clear in many engineering and scientific applications, where we use the results of measuring appropriate quantities at different moments of time as inputs for prediction, and we know that these quantities are usually *not* independent – they satisfy some differential equations. As a result, we do not need to use all n inputs – if there are $m \ll n$ independent ones, this means that it is sufficient to use only m of the inputs – or, alternatively, m different combinations of inputs, as long as they combinations are independent (and, in general, they are); see, e.g., [22].

And this is exactly what is happening in a deep neural network. Indeed, in the traditional neural network, in which we have many neurons in the processing (hidden) layer – we can have as many as inputs or even more. In contrast, in the deep neural networks, the number of neurons in each layer is limited. In particular, the number of neurons in the first processing layer is, in general, much smaller than the number of inputs. And all the resulting computations are based *only* on the outputs $x_k^{(1)}$ of the neurons from this first layer. Thus, in effect, the desired quantity y is computed not based on all n inputs, but based only on m combinations – where m is the number of neurons in the first processing layer.

The empirical fact – that, in spite of this limitation, deep neural networks seem to provide a universal approximation to all kinds of actual dependencies – is an indication that, inputs are usually dependent on each other.

This dependence explains why, empirically, deep neural networks work better than support vector machines – deep networks implicitly take into account this dependency, while support vector machines do not take any advantage of this dependency. As a result, deep networks need fewer parameters than would be needed if they would consider n functionally independent inputs. Hence, during the same time, they can perform more processing and thus, get more accurate predictions.

4 Sparsity Techniques: an Explanation of Their Efficiency

What are sparsity techniques. The above explanations help us explain another empirical fact: that in many applications of signal and image processing, sparsity techniques has been very effective. Specifically, usually, in signal processing, we represent the signal $x(t)$ by the coefficients a_i of its expansion in the appropriate basis $e_1(t)$, $e_2(t)$, etc.: $x(t) \approx \sum\limits_{i=1}^{n} a_i \cdot e_i(t)$. In Fourier analysis, we use the basic of sines and cosines, in wavelet analysis, we use wavelets as the basis, etc. Similarly, in image processing, we represent an image $I(x)$ by the coefficients of its expansion over some basis.

It turns out that in many practical problems, we can select the basis $e_i(t)$ in such a way that for most actual signals, the corresponding representation becomes *sparse* in the sense that most of the corresponding coefficients a_i are zeros. This phenomenon leads to very efficient algorithms for signal and image processing; see, e.g., [1, 4, 5, 6, 7, 10, 11, 13, 14, 15, 16, 24, 25, 26, 28, 30, 32]. However, while empirically successful, from the theoretical viewpoint, this phenomenon largely remains a mystery: why can we find such a basis? Some preliminary explanations were provided in our previous papers [8, 12], but additional explanations are definitely desirable.

Our new explanation. The shape of the actual signal $x(t)$ depends on many different phenomena. So, in general, we can say that $x(t) = F(t, c_1, \ldots, c_N)$ for some function F, where $c_1, \ldots, c_N$ are numerical values characterizing all these phenomena.

Usual signal processing algorithms implicitly assume that we can have all possible combinations of these values c_i. However, as we have mentioned, in reality, the corresponding phenomena are dependent on each other. As a result, there is a functional dependence between the corresponding values c_i. Only few of them $m \ll N$ are truly independent, others can be determined based on the these few ones.

If we denote the corresponding m independent values by $b_1, \ldots, b_m$, then the above description takes the form $x_i(t) = G(t, b_1, \ldots, b_m)$ for an appropriate function G.

It is known that any continuous function – in particular, our function G – can be approximated by piecewise linear functions. If we use this approximation instead of the original function G, then we conclude that the domain of possible values of the tuples $(b_1, \ldots, b_m)$ is divided into a small number of sub-domains $D_1, \ldots, D_p$ on each of which D_j the dependence of $x_i(t)$ on the values b_i is linear, i.e., has the form

$$x_i(t) = \sum_{k=1}^{m} b_k \cdot e_{jk}(t),$$

for some functions $e_{jk}(t)$.

So, if we take all $m \cdot p$ the functions $e_{jk}(t)$ corresponding to different subdomains as the basis, we conclude that on each subdomain, each signal can be described by no more than $m \ll p \cdot m$ non-zero coefficients – this is exactly the phenomenon that we observe and utilize in sparsity techniques.

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] B. Amizic, L. Spinoulas, R. Molina, and A. K. Katsaggelos, "Compressive blind image deconvolution", *IEEE Transactions on Image Processing*, 2013, Vol. 22, No. 10, pp. 3994–4006.

[2] 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.

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

[4] E. J. Candès, J. Romberg and T. Tao, "Stable signal recovery from incomplete and inaccurate measurements", *Comm. Pure Appl. Math.*, 2006, Vol. 59, pp. 1207–1223.

[5] E. Candès, J. Romberg, and T. Tao, "Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information", *IEEE Transactions on Information Theory*, 2006, Vol. 52, No. 2, pp. 489–509.

[6] E. J. Candès and T. Tao, "Decoding by linear programming", *IEEE Transactions on Information Theory*, 2005, Vol. 51, No. 12, pp. 4203–4215.

[7] E. J. Candès and M. B. Wakin, "An Introduction to compressive sampling", *IEEE Signal Processing Magazine*, 2008, Vol. 25, No. 2, pp. 21–30.

[8] F. Cervantes, B. Usevitch, L. Valera, and V. Kreinovich, "Why sparse? Fuzzy techniques explain empirical efficiency of sparsity-based data- and image-processing algorithms", In: L. Zadeh, R. R. Yager, S. N. Shahbazova, M. Reformat, and V. Kreinovich (eds.), *Recent Developments and New Direction in Soft Computing: Foundations and Applications*, Springer Verlag, Cham, Switzerland, 2018, pp. 419–430.

[9] Th. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts, 2009.

[10] D. L. Donoho, "Compressed sensing", *IEEE Transactions on Information Theory*, 2005, Vol. 52, No. 4, pp. 1289–1306.

[11] M. F. Duarte, M. A. Davenport, D. Takhar, J. N. Laska, T. Sun, K. F. Kelly, and R. G. Baraniuk, "Single-pixel imaging via compressive sampling," *IEEE Signal Processing Magazine*, 2008, Vol. 25, No. 2, pp. 83–91.

[12] T. Dumrongpokaphan, O. Kosheleva, V. Kreinovich, and A. Belina, "Why sparse?", 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.

[13] T. Edeler, K. Ohliger, S. Hussmann, and A. Mertins, "Super-resolution model for a compressed-sensing measurement setup", *IEEE Transactions on Instrumentation and Measurement*, 2012, Vol. 61, No. 5, pp. 1140–1148.

[14] M. Elad, *Sparse and Redundant Representations*, Springer Verlag, 2010.

[15] Y. C. Eldar and G. Kutyniok (eds.), *Compressed Sensing: Theory and Applications*, Cambridge University Press, New York, 2012.

[16] S. Foucart and H. Rauhut, *A Mathematical Introduction to Compressive Sensing*, Birkhäuser, New York, 2013.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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

[22] V. Kreinovich, O. Kosheleva, and J. Urenda, "Why such a nonlinear process as protein synthesis is well approximated by linear formulas", *Abstracts of the 2019 Southwest and Rocky Mountain Regional Meeting of the American Chemical Society SWRMRM'2019*, El Paso, Texas, November 13–16, 2019.

[23] V. Kreinovich, A. Lakeyev, J. Rohn, and P. Kahl. (1998) *Computational Complexity and Feasibility of Data Processing and Interval Computations*, Kluwer, Dordrecht.

[24] J. Ma and F.-X. Le Dimet, "Deblurring from highly incomplete measurements for remote sensing", *IEEE Transactions on Geosciences Remote Sensing*, 2009, Vol. 47, No. 3, pp. 792–802.

[25] L. McMackin, M. A. Herman, B. Chatterjee, and M. Weldon, "A high-resolution swir camera via compressed sensing", *Proceedings of SPIE*, 2012, Vol. 8353, No. 1, p. 8353-03.

[26] B. K. Natarajan, "Sparse approximate solutions to linear systems", *SIAM Journal on Computing*, 1995, Vol. 24, pp. 227–234.

[27] C. Papadimitriou. (1994). *Computational Complexity*, Addison-Wesley, Reading, Massachusetts.

[28] V. M. Patel and R. Chellappa, *Sparse Representations and Compressive Sensing for Imaging and Vision*, Springer, New York, 2013.

[29] I. Steinwart and A. Christmann, *Support Vector Machines*, Springer, New York, 2008.

[30] Y. Tsaig and D. Donoho, "Compressed sensing", *IEEE Transactions on Information Theory*, 2006, Vol. 52, No. 4, pp. 1289–1306.

[31] R. J. Vanderbei, *Linear Programming: Foundations and Extensions*, Springer, New York, 2014.

[32] L. Xiao, J. Shao, L. Huang, and Z. Wei, "Compounded regularization and fast algorithm for compressive sensing deconvolution", *Proceedings of the 6th International Conference on Image Graphics*, 2011, pp. 616–621.