Journal of Uncertain Systems
Vol.8, No.3, pp.193-204, 2014
Online at: www.jus.org.uk

Why a Model Produced by Training a Neural Network is Often More Computationally Efficient than a Nonlinear Regression Model: A Theoretical Explanation

Jaime Nava, Vladik Kreinovich*
Department of Computer Science, University of Texas at El Paso, El Paso, TX 79968, USA

Received 27 July 2012; Revised 23 August 2013

Abstract

Many real-life dependencies can be reasonably accurately described by linear functions. If we want a more accurate description, we need to take non-linear terms into account. To take nonlinear terms into account, we can either explicitly add quadratic terms to the regression equation, or, alternatively, we can train a neural network with a non-linear activation function. At first glance, regression algorithms lead to simpler expressions, but in practice, often, a trained neural network turns out to be a more computationally efficient way of predicting the corresponding dependence. In this paper, we provide a reasonable explanation for this empirical fact.

Keywords: neural networks, linear regression, nonlinear regression

1 Formulation of the Problem

Need for indirect measurements. Some quantities describing the physical world are easy to measure directly: e.g., we can directly measure the temperature at different Earth locations, we can directly measure the road width, etc. However, many other quantities y are difficult (or even impossible) to measure directly. For example, it is difficult to directly measure the distance to a faraway star, or the amount of oil in an oil well.

Such quantities are measured *indirectly*; see, e.g., [13]. Specifically, to estimate the values of such a difficult-to-measure quantity y, we measure easier-to-measure quantities $x_1, \ldots, x_n$ which are related to y by a dependence $y = f(x_1, \ldots, x_n)$, and then we use this dependence to estimate the value of the desired quantity y. For example, to estimate the distance to a star, we measure the angular direction to this star at two different seasons, when the Earth is at the opposite sides of the Sun, and then use trigonometry to transform the known distance from the Earth to the Sun and the measured angles into the distance to the star; see, e.g. [6]. Similarly, to estimate the amount of oil in an oil well, we perform seismic, gravity, and other measurements, and then use all these measurement results to determine first the distribution of density at different depths and locations and, finally, the estimated amount of oil; see, e.g., [16].

Need for prediction. An important example of quantities which cannot be directly measured are *future values* of the quantities. The only way to predict the future value of the quantities is to measure the current values of these and other quantities, and to use the known dynamical laws to predict the future values.

For example, we know the exact equations that describe the planets' motion, and we know how to (numerically) solve these equations. Thus, once we know the current values $x_1, \ldots, x_n$ of the planets' coordinates and velocities, we can determine the position y of each planet at a desired future moment of time.

*Corresponding author.
Emails: nava.jaime@gmail.com (J. Nava), vladik@utep.edu (V. Kreinovich).

Need to determine dependencies from the experimental data. In the above examples, the dependence is known. In many other practical situations, however, we know (or conjecture) that there is a dependence, but we do not know the equations that describe the corresponding dependence. In such situations, this dependence must be determined experimentally. Specifically, in several (S) situations $s = 1, \ldots, S$, we measure the values of both the dependent variable y and of the independent variables x_i. Then, we use the results $\left(x_1^{(s)}, \ldots, x_n^{(s)}, y^{(s)}\right)$ of these measurements to find a function $f(x_1, \ldots, x_n)$ which is consistent with all these measurement results, i.e., for which

$$y^{(s)} \approx f\left(x_1^{(s)}, \ldots, x_n^{(s)}\right)$$

for all s from 1 to S. (The equality is usually approximate since the measurements are approximate, and moreover, often the value y is often only approximately determined by the values of the variables $x_1, \ldots, x_n$.)

Once the dependence $f(x_1, \ldots, x_n)$ is determined from the experimental results, we can use this function to estimate the value y of the desired quantity (in particular, to predict the future value of the quantity of interest).

Need to make sure that the resulting prediction models are computationally efficient. In comparison with super-fast computations on modern computers, measurements take a lot of time. So, it is usually tolerable to spend a significant amount of computation time processing the results of these measurements – as long as these computations lead to an efficient model for estimating (in particular, for predicting) the value of the desired quantity.

Once the dependence is determined, we can use it for the actual estimation and prediction. In many practical situations, these estimates and predictions need to be made fast. For example, a tornado can reach its destructive power in a period ranging from minutes to an hour. Thus, to make a useful prediction of the direction in which the tornado will move, we need to finish the corresponding computations faster than in a few minutes (or at least faster than in an hour). In control systems, we need even faster computations: e.g., in a fly-by-wire airplane, a system needs to predict the future trajectory and make needed adjustments in sub-second time.

In other words, in most dependency-determination problems, it is not that critical how long it take to come up with a dependency model $y = f(x_1, \ldots, x_n)$, but it is very important that the resulting model is computationally efficient.

How to determine the dependence from the experimental data. There are many different ways of determining the dependence from the experimental data. In the past, most researchers and practitioners used more traditional *regression* methods, in which we select a model, and use numerical techniques to find the parameters of the model from the observed data. At present, in many cases, this determination is done by using *machine learning* algorithms, algorithms that try to implement the ideas borrowed on how we humans learn different dependencies. Historically the first successful machine learning algorithms were algorithms based on neural networks. These algorithms still constitute the bulk of practical applications of machine learning techniques to the problem of determining the dependence from the experimental data; see, e.g., [3].

Let us briefly describe the main ideas behind both classes of algorithms, and then the challenging empirical fact that we are explaining in this paper. Let us start with the more traditional regression techniques.

Linear regression. In many practical situations, the dependence $f(x_1, \ldots, x_n)$ is smooth: informally, this means that small changes in x_i lead to equally small changes in y. In the first approximation, a smooth function can be approximated by a linear expression

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

for appropriate coefficients c and c_i.

It is reasonable to describe the corresponding dependence by explicitly listing the coefficients c and c_i. Thus, to determine the corresponding model from the experimental data, we need to estimate the values of

these coefficients based on the measurement results $\left(x_1^{(s)}, \ldots, x_n^{(s)}, y^{(s)}\right)$, i.e., based on the system of equations

$$y^{(s)} = c + \sum_{i=1}^{n} c_i \cdot x_i^{(s)} + \Delta y^{(s)},$$

where $\Delta y^{(s)}$ is the approximation error (a.k.a. noise). This task is known as *linear regression*; see, e.g., [15].

The linear regression task can be described as the need to find the coefficients c and c_i for which the S-dimensional vector $\mathbf{f} = \left(f^{(1)}, \ldots, f^{(s)}, \ldots, f^{(S)}\right)$ formed by the predicted values $f^{(s)} = f\left(x_1^{(s)}, \ldots, x_n^{(s)}, y^{(s)}\right) = c + \sum\limits_{i=1}^{n} c_i \cdot x_i^{(s)}$ is as close as possible to the vector formed by the actually observed values $\mathbf{y} = \left(y^{(1)}, \ldots, y^{(s)}, \ldots, y^{(S)}\right)$. A natural way to describe the distance between two vectors $\mathbf{a} = \left(a^{(1)}, \ldots, a^{(s)}, \ldots, a^{(S)}\right)$ and $\mathbf{b} = \left(b^{(1)}, \ldots, b^{(s)}, \ldots, b^{(S)}\right)$ is to use the usual Euclidean distance $d(\mathbf{a}, \mathbf{b}) = \sqrt{\sum\limits_{s=1}^{S} \left(a^{(s)} - b^{(s)}\right)^2}$. Thus, a natural idea is to find the values c and c_i for which this distance is the smallest possible. From the mathematical viewpoint, minimizing the distance is equivalent to minimizing the square of the distance, i.e., the sum $d^2(\mathbf{a}, \mathbf{b}) = \sum\limits_{s=1}^{S} \left(a^{(s)} - b^{(s)}\right)^2$. Thus, we arrive at the problem of finding the values c and c_i for which the corresponding sum

$$\sum_{s=1}^{S} \left(\Delta y^{(s)}\right)^2 = \sum_{s=1}^{S} \left(y^{(s)} - \left(c + \sum_{i=1}^{n} c_i \cdot x_i^{(s)}\right)\right)^2$$

is the smallest possible. This approach to finding the coefficients c and c_i is usually called the *Least Squares approach*.

A usual way to find the corresponding values c and c_i is to differentiate the objective function with respect to each of the variables c and c_i and equate all the derivatives to 0. Since the objective function is quadratic, the derivatives are linear functions of the unknowns c and c_i. Thus, we arrive at the system of linear equations – and many efficient algorithms are known that solve such systems.

The Least Squares approach is optimal when the measurement errors are independent and normally distributed. In other cases, more sophisticated methods are needed but the result is the same – a linear model described by the coefficients c and c_i.

Some coefficients can be safely ignored. It is worth mentioning that some of these coefficients can be safely ignored. Indeed, we started with the situation in which we do not know the actual dependence. To find this dependence, we measured all the quantities x_i which could potentially influence the value of the desired quantity y. In practice, it sometimes happens that not all of these quantities are important. In case of an exact measurement, this would mean that in the resulting dependence $y = f(x_1, \ldots, x_n)$, there is no dependence on some of the variables x_i. In the case of linear regression, this means that some of the corresponding coefficients c_i are equal to 0. Because of the measurement errors, even when the actual value of c_i is 0, the estimated values of c_i will be, in general, different from 0 – but still close to 0, i.e., small. Thus, a reasonable idea is to ignore all the coefficients which are sufficiently close to 0, and only keep the ones whose absolute value is above a certain threshold.

A similar possibility of ignoring some coefficients (and saving some computation time) comes from the fact that in practice, we rarely need to know the exact values of y_i. For example, when we predict the amount of oil in an oil well, it is usually sufficient to have a crude estimate to see if exploiting this well is economically viable; in predicting the path of a tornado, it is desirable to get at least a general direction in which it will move, if we cannot predict an exact direction in reasonable time. In all these cases, it is sufficient to get a good approximation to the actual value y; so, variables x_i for which the coefficients c_i are small can be safely ignored.

Need to go beyond linear dependencies. Many real-life dependencies are non-linear. for such dependencies, linear regression – an approximation by a linear function – is imprecise. To get a more accurate description of the desired dependence, we need to go beyond the linear approximation.

Quadratic regression: a natural mathematical approach. How can we go beyond a linear approximation? In many cases, the dependence $y = f(x_1, \ldots, x_n)$ is smooth. A sufficiently smooth function can be expanded in Taylor series, i.e., represented as a sum of linear, quadratic, cubic, etc. terms. For small values x_i, the linear terms are the main ones, the quadratic terms are much smaller, the cubic terms are even more smaller, etc. For example, for $x_i \approx 0.$, linear terms are of order 0.1, quadratic terms are of order $0.1^2 = 0.01$, cubic terms are of order $0.1^3 = 0.001$, etc. So, in the first approximation, we can simply keep the linear terms and ignore all the higher-order terms. When the linear approximation is not sufficient, the natural next idea is to keep both linear and quadratic terms and ignore cubic and higher-order terms.

In the resulting approximation, we approximate the desired dependence by the following dependence:

$$f(x_1, \ldots, x_n) = c + \sum_{i=1}^{n} c_i \cdot x_i + \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} \cdot x_i \cdot x_j$$

for appropriate coefficients c, c_i, and c_{ij}.

It is reasonable to describe the corresponding dependence by explicitly listing the coefficients c, c_i, and c_{ij}. Thus, to determine the corresponding model from the experimental data, we need to estimate the values of these coefficients based on the measurement results $\left(x_1^{(s)}, \ldots, x_n^{(s)}, y^{(s)}\right)$, i.e., based on the system of equations

$$y^{(s)} = c + \sum_{i=1}^{n} c_i \cdot x_i^{(s)} + \Delta y^{(s)},$$

where $\Delta y^{(s)}$ is the approximation error (a.k.a. noise). To find the values of these coefficients, we can use the same Least Squares techniques as for the linear regression, i.e., find the values c, c_i, and c_{ij} for which the corresponding sum

$$\sum_{s=1}^{S} \left(\Delta y^{(s)}\right)^2 = \sum_{s=1}^{S} \left(y^{(s)} - \left(c + \sum_{i=1}^{n} c_i \cdot x_i^{(s)} + \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} \cdot x_i^{(s)} \cdot x_j^{(s)} \right) \right)^2$$

is the smallest possible. Differentiating the objective function with respect to each of the variables c, c_i, and c_{ij}, and equating all the derivatives to 0, we arrive at the system of linear equations in terms of the unknowns c, c_i, and c_{ij}. Similarly to the case of linear regression, we can then safely ignore the coefficients which are close to 0.

Neural networks as an alternative to quadratic regression: main idea. To describe non-linear dependencies, instead of using quadratic functions – inspired by mathematical analysis – another possibility is to use an approach inspired by biological data processing, i.e., the approach of neural networks; see, e.g., [3].

Let us briefly describe the main ideas behind neural networks. In a living being, optical, acoustic etc. sensor cells generate sequences of pulses whose frequency x_i is proportional to the intensity of the corresponding inputs. These signals are then processed by special cells called *neurons*. A neuron collects signals from other cells; some of these signals come unchanged, some are attenuated by the connections, i.e., get reduced from the original value x_i to a new value $w_i \cdot x_i$, where w_i is the coefficient of this attenuation. By adding up all the pulses that come to this neuron from different other neurons, we get the total input signal $\sum w_i \cdot x_i$. The neuron has an activation threshold w_0, so that the neuron is activated when the input signal exceeds this threshold. The level of activation depends on how much it exceeds, i.e., it depends on the difference $\sum w_i \cdot x_i - w_0$. In other words, the output signal y of a neuron depends on this difference $y = s_0\left(\sum w_i \cdot x_i - w_0\right)$, for an appropriate function $s_0(z)$. This function $s_0(z)$ is called an *activation function*. Usually, the "sigmoid" activation function is used:

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

Not every function can be represented in the form $y = s_0\left(\sum w_i \cdot x_i - w_0\right)$ corresponding to processing by a single neuron. To perform more general data processing, it is necessary to form a *neural network*, where output signals from neurons become inputs to other neurons. It turns out (see, e.g., [3]) that to be able to approximate an arbitrary continuous function on a bounded set with an arbitrary accuracy, it is sufficient to have *two* layers of processing neurons, and moreover, it is sufficient to have, in the second processing layer, simplified *linear* neurons, with a trivial activation function $s_0(z) = z$. As a result, we arrive at the following traditional 3-layer neural networks.

Traditional 3-layer neural networks: description. In the traditional 3-layer neural networks, we start with n input signals $x_1, \ldots, x_n$. These signals are then fed into several (m) non-linear neurons each of which produces a signal

$$y_i = s_0\left(\sum_{j=1}^{n} w_{ij} \cdot x_j - w_{i0}\right), \quad 1 \le i \le m.$$

The outputs $y_1, \ldots, y_m$ of the non-linear neurons are then fed into the final linear neuron, resulting in the final output

$$y = \sum_{i=1}^{m} W_i \cdot y_i - W_0.$$

The activation function $s_0(z)$ is usually fixed, while the values W_i and w_{ij} (called *weights*) are selected so as to fit the data, i.e., that for all for all $s = 1, \ldots, S$, we have $y^{(s)} = f\left(x_1^{(s)}, \ldots, x_n^{(s)}\right) + \Delta y^{(s)}$ with small approximation errors $\Delta y^{(s)}$.

Comment. The outputs $y_1, \ldots, y_m$ of m intermediate neurons are not returned to the user as the final output. Because of this, the corresponding neurons are called *hidden.*

To describe a function represented by a neural network, we need to describe, for each of m hidden neurons,

- n weights $w_{i1}, \ldots, w_{in}$ (corresponding to n different inputs $x_1, \ldots, x_n$) and
- a weight w_{i0} (corresponding to the threshold),

to the total of $n + 1$ weights. We also need to describe $m + 1$ values $W_1, \ldots, W_m, W_0$ describing the output neuron: m values $W_1, \ldots, W_m$ corresponding to m hidden neurons and a value W_0 describing this neuron's threshold. Thus, to describe a function represented by a neural network with m hidden neurons, we need to describe the values of $m \cdot (n + 1) + (m + 1)$ parameters w_{ij} and W_j.

To exactly describe all possible continuous functions, we need infinitely many parameters: we need two parameters c_0 and c_1 to describe a general linear function $c_0 + c_1 \cdot x$, we need three parameters c_i to describe a general quadratic function $c_0 + c_1 \cdot x + c_2 \cdot x^2$, etc.

The more hidden neurons we use, the more parameters we can change, and thus, the more accurate approximation we can get.

Orthogonal neurons: a brief description. Sometimes, in the process of learning, i.e., in the process of adjusting the values of the weights to fit the data, some of the hidden neurons are *duplicated*, in the sense that for some hidden neurons $i' \ne i''$, we get the exact same weights $w_{i'j} = w_{i''j}$ for all $j = 1, \ldots, n$ and for $j = 0$. Since the weights $w_{i'j}$ and $w_{i''j}$ of these hidden neurons are the same, their outputs

$$y_{i'} = s_0\left(\sum_{j=1}^{n} w_{i'j} \cdot x_j - w_{i'0}\right) \text{ and } y_{i''} = s_0\left(\sum_{j=1}^{n} w_{i''j} \cdot x_j - w_{i''0}\right)$$

are the same as well: $y_{i'} = y_{i''}$.

In this duplication case, we can combine these two neurons into one. Namely, the output y of the neural network takes the form

$$y = \sum_{i=1}^{m} W_i \cdot y_i - W_0 = \sum_{i \ne i', i \ne i''} W_i \cdot y_i + W_{i'} \cdot y_{i'} + W_{i''} \cdot y_{i''} - W_0.$$

Since $y_{i'} = y_{i''}$, we can combine the corresponding two terms into one:

$$W_{i'} \cdot y_{i'} + W_{i''} \cdot y_{i''} = W_{i'} \cdot y_{i'} + W_{i''} \cdot y_{i'} = (W_{i'} + W_{i''}) \cdot y_{i'}.$$

Thus, we can simplify the above expression for the final output y into a simpler expression (thus merging the two neurons into one):

$$y = \sum_{i \ne i', i \ne i''} W_i \cdot y_i + W'_{i'} \cdot y_{i'} - W_0,$$

where we denoted $W'_{i'} = W_{i'} + W_{i''}$. This expression is the output of a neural network with $m-1$ hidden neurons ($i = 1, \ldots, i''-1, i''+1, \ldots, m$) in which:

- all $m-1$ hidden neurons are the same as in the original neural network,
- the weight $W_{i'}$ which is now replaced by a new value $W'_{i'} = W_{i'} + W_{i''}$ that combines the weights corresponding to both original neurons i' and i'', and
- all the other weights W_i and W_0 of the linear output neuron are the same as before.

This shows that the output of the original neural network, with m hidden neurons $i = 1, \ldots, i''-1, i''$, $i''+1$, …, m, can be also described as the output of a neural network with only $m-1$ hidden neurons.

We have already mentioned that the more hidden neurons we have, the more accurate approximations we can, in principle, achieve. To make sure that we do get a better approximation, when we add an extra hidden neuron, we must avoid duplication situations like above, when we get the exact same approximation as without this new hidden neuron.

To avoid this duplication problem, B. Apolloni and others suggested [1, 2, 12, 14, 17] that we *orthogonalize* the neurons during training, e.g., that we make sure that the weight vectors $w_i = (w_{i1}, \ldots, w_{in})$ corresponding to different hidden neurons are orthogonal to each other.

It is known that two vectors $a = (a_1, \ldots, a_n)$ and $b = (b_1, \ldots, b_n)$ are orthogonal if their dot (scalar) product $\langle a, b\rangle \stackrel{\text{def}}{=} \sum_{j=1}^{n} a_j \cdot b_j$ is equal to 0. Indeed, the dot product is equal to $\langle a, b\rangle = |a| \cdot |b| \cdot \cos(A)$, where $|a| \stackrel{\text{def}}{=} \sqrt{a_1^2 + \ldots + a_n^2}$ is the length of the vector a, $|b|$ is the length of the vector b, and A is the angle between these two vectors. Thus, the dot product is equal to 0 if and only if the cosine is equal to 0, i.e., if the angle A is equal to 90°.

So, for neural networks, orthogonalization means that we have

$$\langle w_i, w_{i'}\rangle \stackrel{\text{def}}{=} \sum_{j=1}^{n} w_{ij} \cdot w_{i'j} = 0$$

for all $i \neq i'$; see also [9, 10].

Orthogonalization prevents duplication: indeed, if we had $w_{i'j} = w_{i''j}$ for all j, then we would have

$$\langle w_{i'}, w_{i''}\rangle \stackrel{\text{def}}{=} \sum_{j=1}^{n} w_{i'j} \cdot w_{i''j} = \sum_{j=1}^{n} (w_{i'j})^2 > 0,$$

and orthogonalization makes sure that $\langle w_{i'}, w_{i''}\rangle = 0$.

Which is better: quadratic regression or neural networks? To describe a non-linear dependence, we can use either a quadratic regression or a neural network. Which of the techniques should we use?

Our main objective is to use the resulting approximate model to predict the value of y based on the values of the inputs $x_1, \ldots, x_n$. In many practical situations, it is important to be able to perform the resulting computation of y as soon as possible.

Expected answer and a somewhat surprising empirical fact. At first glance, the formulas describing neural networks are more complex than the formulas describing quadratic regression: quadratic regression only uses addition and multiplication, the basic computer operations, while neural network formulas – in their computation of the activation function – also involve the more difficult-to-compute exponential function. So, one would expect that computations using quadratic regression should be faster than computations using a pre-trained neural network.

Surprisingly, in many case, pre-trained neural networks work faster than regression; see, e.g., [3].

What we do in this chapter. In this chapter, we provide a possible theoretical explanation of this somewhat surprising empirical fact.

Comment. It is worth emphasizing that in comparing the computation times, we assume that both a regression model and the neural network have already been trained, i.e., that we have already found the values of the corresponding parameters that fit all the observations. At this stage, once both the regression model and the neural network are pre-trained and used, the neural network is often faster.

The situation is radically different if we consider the time needed to train the corresponding models. In the regression model, the dependence on the parameters is linear. As a result, training is easy and relatively fast, e.g., by using the least squares techniques. In contrast, for the neural network, the dependence on the parameters is, in general, non-linear. In this case, instead of explicit methods for producing the exact solution, we need to perform iterative methods. As a result, training a neural network often requires several thousand iterations – and is, thus, much slower than training a regression model.

2 Towards an Explanation

How to compare the computation time of two algorithms. Our objective is to come up with an expression that, given the inputs $x_1, \ldots, x_n$, would generate the value $y = f(x_1, \ldots, x_n)$, for the corresponding pre-trained model $f(x_1, \ldots, x_n)$.

Which operations are the most efficient on modern computers? In numerical computations that form the bulk of modern high performance computer usage, the most time-consuming operation is the *dot product*, i.e., computing the $\langle a, b\rangle$ for given vectors a and b.

The prevalence of dot product makes sense from the mathematical viewpoint, since most numerical methods are based on linearization, and in the linear approximation, any function of n variables is approximated as $c + \sum_{i=1}^{n} c_i \cdot x_i$, i.e., as a constant plus a dot product between the vector of inputs and the vector of coefficients.

Not surprisingly, most computer speed-up innovations are aimed at computing the dot product faster – e.g., the multiply-accumulate operation which is an important part of digital signal processing or fused multiple-add operation which is now hardware supported on many modern computers; see, e.g., [7].

From this viewpoint, the way to speed up any computation is to reduce it to as few dot products as possible.

In this chapter, we show that neural networks require, in general, fewer dot product computations than quadratic regression. This explains the empirical fact that pre-trained neural networks are often faster than pre-trained regression models.

Comment. Of course, comparing the number of values to fetch is also an approximate description of the computation time. The actual computation time also depends on the order in which we fetch these values, how easy it is to pre-fetch the corresponding values beforehand, etc. Because of the approximate character of this description, when one find out that one class of algorithms requires fewer values to fetch than another class, it does not necessarily mean that the first class of algorithms always works faster – but it usually means that algorithms from this class often work faster. This will be exactly the case with regression and neural networks:

- while in general, pre-trained neural networks are often faster than pre-trained regression models,
- sometimes the opposite is true: pre-trained regression models work faster than pre-trained neural networks.

How to efficiently compute $f(x_1, \ldots, x_n)$ using pre-trained quadratic regression model. Computing the value of the linear part $c_0 + \sum_{i=1}^{m} c_i \cdot x_i$ of the quadratic regression requires computing exactly one dot product: the sum $\sum_{i=1}^{m} c_i \cdot x_i$.

Computing the value of the quadratic form $\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij} \cdot x_i \cdot x_j$ requires $n+1$ dot products:

- first, we compute n dot products $c'_i \stackrel{\text{def}}{=} \sum_{j=1}^{n} c_{ij} \cdot x_j$ for $i = 1, \ldots, n$;

- then, to find the desired value of the quadratic form, we compute the dot product $\sum_{i=1}^{n} c'_i \cdot x_i$.

Thus, totally, we need to compute $1 + (n+1) = n+2$ dot products.

How to efficiently compute $f(x_1, \ldots, x_n)$ using a pre-trained neural network: analysis of the problem. In our analysis, we have restricted regression models to quadratic terms. In other words, we consider the second approximation, in which each function is approximated by a quadratic expression – e.g., by the sum of the constant, linear, and quadratic terms of its Taylor expansion, so that cubic and higher orders can be safely ignored. It is reasonable to use the same approximation when considering neural networks as well.

In this second approximation, we can approximate the non-linear activation function $s_0(x)$ by the sum of its constant, linear, and quadratic terms:

$$s_0(x) \approx s + s_1 \cdot x + s_2 \cdot x^2.$$

In this case, the above formula for the output of an intermediate neuron takes the following form:

$$y_i = s_0 + s_1 \cdot \left(\sum_{j=1}^{n} w_{ij} \cdot x_j - w_{i0} \right) + s_2 \cdot \left(\sum_{j=1}^{n} w_{ij} \cdot x_j - w_{i0} \right)^2.$$

The quadratic term in this expression can be described as

$$\left(\sum_{j=1}^{n} w_{ij} \cdot x_j - w_{i0} \right)^2 = \left(\sum_{j=1}^{n} w_{ij} \cdot x_j \right)^2 - 2w_{i0} \cdot \left(\sum_{j=1}^{n} w_{ij} \cdot x_j \right) + w_{i0}^2.$$

Here, the term

$$\left(\sum_{j=1}^{n} w_{ij} \cdot x_j \right)^2 = (\langle w_i, x \rangle)^2$$

(in which we denote $x = (x_1, \ldots, x_n)$) is the only quadratic term, the other terms are linear. Thus, the output $y = \sum_{i=1}^{m} W_i \cdot y_i - W_0$ of the neural networks consists of a linear part plus a quadratic part of the type

$$Q_n = \sum_{i=1}^{n} W_i \cdot \langle w_i, x \rangle^2.$$

This part corresponds to the quadratic part $\sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} \cdot x_i \cdot x_j$ of the original Taylor-series representation:

$$Q_n = \sum_{i=1}^{n} W_i \cdot \langle w_i, x \rangle^2 = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} \cdot x_i \cdot x_j.$$

As we have mentioned, it is reasonable to select the vectors w_i to be orthogonal. By dividing each vector w_i by its length (and appropriately multiplying W_i by this length), we can assume that the vectors are also orthonormal, i.e., that $\langle w_i, w_i \rangle = 1$ for all i. In the orthonomal basis formed by these vectors w_i,

- the corresponding matrix c_{ij} becomes a diagonal matrix,
- with values W_i on the diagonal.

Thus:

- the vectors w_i are eigenvectors of the matrix c_{ij}, while
- the values W_i are the eigenvalues of this matrix.

In the neural network representation, to compute the value with a certain accuracy, we can dismiss the terms corresponding to small eigenvalues W_i. As a result, instead of the original formula with n eigenvalues, we get a simplified formula with $n' < n$ eigenvalues:

$$Q_n \approx \sum_{i=1}^{n'} W_i \langle w_i, x\rangle^2.$$

How to efficiently compute $f(x_1, \ldots, x_n)$ for a pre-trained neural network. As we have just mentioned, in the neural network representation, to compute the value with a certain accuracy, we can dismiss the terms corresponding to small eigenvalues W_i. As a result, instead of the original formula with n eigenvalues, we get a simplified formula with $n' < n$ eigenvalues:

$$Q_n \approx \sum_{i=1}^{n'} W_i \langle w_i, x\rangle^2.$$

From this representation, we can see that fewer than $n+1$ dot products are needed:

- first, we compute $n' < n$ dot products $z_i = \langle w_i, x\rangle$ corresponding to n' non-dismissed eigenvectors w_i;
- then, we perform a component-wise vector operation to compute the values $t_i = z_i \cdot z_i$; such vector operations are highly parallelizable and can be performed really fast on most modern computers; see, e.g., [7];
- finally, to find the desired result, we compute the dot product $\sum\limits_{i=1}^{n'} W_i \cdot t_i$.

So, totally, we need to compute $n'+1$ dot products.

Resulting comparison. In the Appendix, we show that on average, the number n' of non-dismissed eigenvalues is smaller than the number n of all eigenvalues. Thus, when we use a pre-trained neural network, we need to compute fewer dot products and thus, our computations are, on average, faster. This explains the empirical fact that in data processing, pre-trained neural networks are often more efficient than more traditional statistical methods such as pre-trained regression models.

Comment. In terms of the matrix c_{ij}, the two representations correspond to the following:

- in the traditional regression representation, we store all the components c_{ij} of the original matrix;
- in the neural network representation, we store instead the eigenvectors and eigenvalues of this matrix.

This conclusion prompts a natural analogy with quantum physics; see, e.g., [4]. In quantum physics, from the mathematical viewpoint, an observable quantity can be described by a corresponding matrix c_{ij}. However, a more physically natural description is to describe possible values of this quantity – which are exactly eigenvalues of this matrix – and states in which this quantity has these exactly values, which are eigenvectors of the matrix. In this example, a representation via eigenvalues and eigenvectors is clearly intuitively preferable.

Acknowledgments

This work was supported in part by the National Science Foundation grants HRD-0734825 and DUE-0926721, by Grant 1 T36 GM078000-01 from the National Institutes of Health, by Grant MSM 6198898701 from MŠMT of Czech Republic, and by Grant 5015 "Application of fuzzy logic with operators in the knowledge based systems" from the Science and Technology Centre in Ukraine (STCU), funded by European Union. The authors are very thankful to the anonymous referees for valuable suggestions.

References

[1] Apolloni, B., Bassis, S., and L. Valerio, A moving agent metaphor to model some motions of the brain actors, *Abstracts of the Conference "Evolution in Communication and Neural Processing from First Organisms and Plants to Man ... and Beyond"*, 2010.

[2] Apolloni, B., Bassis, S., and L. Valerio, Training a network of mobile neurons, *Proceedings of the 2011 International Joint Conference on Neural Networks*, 2011.

[3] Bishop, C.M., *Pattern Recognition and Machine Learning*, Springer, New York, 2007.

[4] Feynman, R., Leighton, R., and M. Sands, *The Feynman Lectures on Physics*, Addison Wesley, Boston, Massachusetts, 2005.

[5] Götze, F., and A. Tikhomirov, Rate of convergence in probability to the Marchenko-Pastur law, *Bernoulli*, vol.10, no.3, pp.503–548, 2004.

[6] Gregory, S.A., and M.A. Zeilik, *Introductory Astronomy & Astrophysics*, Brooks Cole Publisher, Florence, Kentucky, 1997.

[7] Hennessy, J.L., and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, California, 2007.

[8] Marchenko, V.A., and L.A. Pastur, Distribution of eigenvalues for some sets of random matrices, *Matematicheski Sbornik*, vol.72, no.4, pp.507–536, 1967.

[9] Nava, J., and V. Kreinovich, Orthogonal bases are the best: a theorem justifying Bruno Apolloni's heuristic neural network idea, *Abstracts of the 9th Joint NMSU/UTEP Workshop on Mathematics, Computer Science, and Computational Sciences*, 2011.

[10] Nava, J., and V. Kreinovich, Orthogonal bases are the best: a theorem justifying Bruno Apolloni's heuristic neural network idea, *Journal of Uncertain Systems*, vol.6, no.2, pp.122–127, 2012.

[11] Nica, A., and R. Speicher, *Lectures on the Combinatorics of Free Probability Theory*, Cambridge University Press, Cambridge, UK, 2006.

[12] Nicolau, V., Palade, V., Aiordachioaie, D., and C. Miholca, Neural network prediction of the roll motion of a ship for intelligent course control, *Knowledge-Based Intelligent Information and Engineering Systems*, vol.4694, pp.284–291, 2007.

[13] Rabinovich, S., *Measurement Errors and Uncertainties: Theory and Practice*, Springer-Verlag, New York, 2005.

[14] Sher, C.F., Tseng, C.-S., and C.-S. Chen, Properties and performance of orthogonal neural network in function approximation, *International Journal of Intelligent Systems*, vol.16, pp.1377–1392, 2001.

[15] Sheskin, D.J., *Handbook of Parametric and Nonparametric Statistical Procedures*, Chapman & Hall/CRC, Boca Raton, Florida, 2007.

[16] Snieder, R., and J. Trampert, *Inverse Problems in Geophysics*, Samizdat Press, Colorado School of Mines, Golden, Colorado, 2000.

[17] Yang, S.-S., and C.-S. Tseng, An orthogonal neural network for function approximation, *IEEE Transactions on Systems, Man, and Cybernetics*, vol.26 (Part B), pp.779–783, 1996.

A Appendix: Number of Dismissed Eigenvalues: Semi-Heuristic Statistical Analysis

Idea. The idea is to dismiss some eigenvalues because their contribution is small. Of course, the number of small eigenvalues depends on the matrix c_{ij}. We would like to know how many such eigenvalues are there *on average*. To formulate this question in precise terms, we need to describe a reasonable probability distribution on the set of all possible matrices.

Random matrices: motivation. In general, for each element c_{ij} of the matrix, we can have both positive and negative values. There are no reasons to expect positive values to be more probable than the negative ones or vice versa. In other words, the situation seems to be symmetric with respect to changing the sign. Thus, the expected value of the element c_{ij} should also be invariant with respect to this transformation. The only number that remains invariant when we change the sign is zero, so we conclude that the mean value of each component c_{ij} should be zero.

Similarly, there is no reason to assume that some of the elements have a different probability distribution; thus, we assume that they are identically distributed. Finally, there is no reason to assume that there is correlation between different elements. Thus, we assume that all the elements are independent. Thus, we arrive at the model in which all the elements are independent identically distributed random variables with mean 0 and a variance σ^2.

Eigenvalues of random matrices. For such random matrices, the distribution of their eigenvalues follows the *Marchenko-Pastur law*; see, e.g., [5, 8, 11]. To be more precise, this law describes the limit case of the following situation. We have an $m \times n$ random matrix X whose elements are independent identically distributed random variables with mean 0 and variance σ^2. Assume that m and n increase in such a way that the ratio m/n tends to a limit $\alpha > 0$. Then, for large n and m, the probability distribution of the eigenvalues of the matrix $Y = XX^T$ is asymptotically equivalent to

$$\rho(x) = \left(1 - \frac{1}{\alpha}\right) \cdot \delta(x) + \rho_c(x),$$

where $\delta(x)$ is Dirac's delta-function (i.e., the probability distribution which is located at the point 0 with probability 1), and $\rho_c(x)$ is different from 0 for $x \in [\alpha_-, \alpha_+]$, where $\alpha_\pm = \sigma^2 \cdot (1 \pm \sqrt{\alpha})^2$, and

$$\rho_c(x) = \frac{1}{2 \cdot \pi \cdot \sigma^2} \cdot \frac{\sqrt{(\alpha_+ - x) \cdot (x - \alpha_-)}}{\alpha \cdot x}.$$

In our case, matrices are square, so $m = n$, $\alpha = 1$ and thus, we have $\alpha_- = 0$, $\alpha_+ = 4\sigma^2$ and thus, the limit probability distribution takes the simplified form

$$\rho(x) = \frac{1}{2 \cdot \pi \cdot \sigma^2} \cdot \frac{\sqrt{(4\sigma^2 - x) \cdot x}}{x}.$$

Eigenvalues x of the matrix $Y = XX^T$ are squares of eigenvalues λ of the original matrix X: $x = \lambda^2$.

We are interested in small eigenvalues. For small eigenvalues, we have $x \ll \sigma$, so the above formula can be further simplified, into

$$\rho(x) \sim \frac{1}{2 \cdot \pi \cdot \sigma^2} \cdot \frac{\sqrt{4\sigma^2 \cdot x}}{x} = \frac{1}{2 \cdot \pi \cdot \sigma^2} \cdot \frac{2 \cdot \sigma \cdot \sqrt{x}}{x} = \frac{1}{\pi \cdot \sigma} \cdot \frac{1}{\sqrt{x}}.$$

The probability density ρ_λ for $\lambda = \sqrt{x}$ can thus be found as

$$\rho_\lambda = \frac{dp}{d\lambda} = \frac{dp}{dx} \cdot \frac{dx}{d\lambda}.$$

For $x = \lambda^2$, we get

$$\frac{dx}{d\lambda} = \frac{d(\lambda^2)}{d\lambda} = 2\lambda,$$

thus,

$$\rho_\lambda(\lambda) = \frac{1}{\pi \cdot \sigma} \cdot \frac{1}{\sqrt{x}} \cdot 2\lambda = \frac{1}{\pi \cdot \sigma} \cdot \frac{1}{\lambda} \cdot 2\lambda = \frac{2}{\pi \cdot \sigma}.$$

This expression for the probability density does not depend on λ at all. Thus, small eigenvalues have an approximately uniform distribution.

Heuristic derivation of the number of eigenvalues that can be safely ignored. We would like to dismiss all the eigenvalues $\lambda_i = W_i$ whose absolute values are smaller than (or equal to) some small number $\delta > 0$. The overall contribution c of these eigenvalues is equal to

$$c = \sum_{i:|\lambda_i| \le \delta} W_i \cdot \langle w_i, x \rangle^2.$$

Since eigenvectors are orthonormal, the n values $\langle w_i, x \rangle^2$ add up to $\langle x, x \rangle^2$. In particular, for unit vectors x, these n values add up to 1. It is reasonable to assume that values corresponding to different eigenvalues are similarly distributed. Under this assumption, all these values have the same mean. The sum of n such means is equal to 1, so each mean is equal to $1/n$.

Each value W_i can be positive or negative. It is reasonable to assume that both negative and positive values are equally possible, so the mean value of each product $W_i \cdot \langle w_i, x \rangle^2$ is 0. Thus, the mean value of the sum is also 0.

Since $\langle w_i, x\rangle^2 \approx 1/n$, the variance should be approximately equal to $W_i^2 \cdot 1/n^2$. It is also reasonable to assume that the products $W_i \cdot \langle w_i, x\rangle^2$ corresponding to different eigenvalues are independent. Thus, the variance V_c of their sum c is equal to sum of their variances, i.e., to

$$V_c = \frac{1}{n^2} \cdot \sum_{i:|\lambda_i|\le\delta} W_i^2.$$

Since the mean is 0, and c is the sum of the large number of small independent components, it is reasonable to conclude, due to the Central Limit theorem, that it is approximately normally distributed; see, e.g., [15]. So, with probability 99.9%, all the values of this sum are located within the three sigma interval $[-3\sqrt{V_c}, 3\sqrt{V_c}]$.

Thus, the square root $\sqrt{V_c}$ is a good indication of the size of the dismissed terms. The size of the function itself can be similarly estimates as $\sqrt{V}$, where

$$V = \frac{1}{n^2} \cdot \sum_i W_i^2,$$

and the sum is taken over all eigenvalues. We want to make sure that the dismissed part does not exceed a given portion ε of the overall sum, i.e., that $\sqrt{V_c} \cdot \varepsilon \cdot \sqrt{V}$, or, equivalently, $V_c \le \varepsilon^2 \cdot V^2$.

Within this constraint, we want to dismiss as many eigenvalues as possible; thus, we should not have $V_c \ll \varepsilon^2 \cdot V^2$, because then, we would be able to dismiss more terms. We should thus have $V_c \approx \varepsilon^2 \cdot V^2$. Because of the above expressions for V_c and for V, we therefore get an equivalent formula

$$\frac{1}{n^2} \cdot \sum_{i:|\lambda_i|\le\delta} W_i^2 \approx \varepsilon^2 \cdot \frac{1}{n^2} \cdot \sum_i W_i^2.$$

Multiplying both sides by n^2, we can simplify this requirement into

$$\sum_{i:|\lambda_i|\le\delta} W_i^2 \approx \varepsilon^2 \cdot \sum_i W_i^2.$$

Since the probability distribution of eigenvalues is described by the density function ρ_λ, and the total number of these eigenvalues is n, we have

$$\sum_i W_i^2 \approx n \cdot \int_{-\infty}^{\infty} \lambda^2 \cdot \rho_\lambda(\lambda)\, d\lambda$$

and similarly,

$$\sum_{i:|\lambda_i|\le\delta} W_i^2 \approx n \cdot \int_{-\delta}^{\delta} \lambda^2 \cdot \rho_\lambda(\lambda)\, d\lambda.$$

Thus, the above requirement takes the form

$$n \cdot \int_{-\delta}^{\delta} \lambda^2 \cdot \rho_\lambda(\lambda)\, d\lambda \approx \varepsilon^2 \cdot n \cdot \int_{-\infty}^{\infty} \lambda^2 \cdot \rho_\lambda(\lambda)\, d\lambda.$$

Dividing both sides by n, we can simplify this into

$$\int_{-\delta}^{\delta} \lambda^2 \cdot \rho_\lambda(\lambda)\, d\lambda \approx \varepsilon^2 \cdot \int_{-\infty}^{\infty} \lambda^2 \cdot \rho_\lambda(\lambda)\, d\lambda.$$

For small λ, as we have derived, $\rho_\lambda \approx \text{const}$, so

$$\int_{-\delta}^{\delta} \lambda^2 \cdot \rho_\lambda(\lambda)\, d\lambda \approx \int_{-\delta}^{\delta} \lambda^2 \cdot \text{const}\, d\lambda = \text{const} \cdot \delta^3$$

(for a slightly different constant, of course).

Thus, the above requirement takes the form $\delta^3 \approx \text{const} \cdot \varepsilon^2$, i.e., $\delta \approx \varepsilon^{2/3}$.

Numerical example. So, for example, for $\varepsilon \approx 10\% = 0.1$, we get $\delta \approx 0.1^{2/3} \approx 0.2$, so $\approx 20\%$ of all the eigenvalues can be safely ignored. As a result, we get a 20% decrease in computation time.