University of Texas at El Paso
ScholarWorks@UTEP

Departmental Technical Reports (CS) | Computer Science

7-1-2023

How to Best Retrain a Neural Network If We Added One More Input Variable

Saeid Tizpaz-Niari
The University of Texas at El Paso, saeid@utep.edu

Vladik Kreinovich
The University of Texas at El Paso, vladik@utep.edu

Follow this and additional works at: https://scholarworks.utep.edu/cs_techrep

 Part of the Computer Sciences Commons, and the Mathematics Commons

Comments:
Technical Report: UTEP-CS-23-42

Recommended Citation

Tizpaz-Niari, Saeid and Kreinovich, Vladik, "How to Best Retrain a Neural Network If We Added One More Input Variable" (2023). *Departmental Technical Reports (CS)*. 1827.
https://scholarworks.utep.edu/cs_techrep/1827

This Article is brought to you for free and open access by the Computer Science at ScholarWorks@UTEP. It has been accepted for inclusion in Departmental Technical Reports (CS) by an authorized administrator of ScholarWorks@UTEP. For more information, please contact lweber@utep.edu.

How to Best Retrain a Neural Network If We Added One More Input Variable

Saeid Tizpaz-Niari and Vladik Kreinovich

Abstract Often, once we have trained a neural network to estimate the value of a quantity y based on the available values of inputs $x_1, \ldots, x_n$, we learn to measure the values of an additional quantity that have some influence on y. In such situations, it is desirable to re-train the neural network, so that it will be able to take this extra value into account. A straightforward idea is to add a new input to the first layer and to update all the weights based on the patterns that include the values of the new input. The problem with this straightforward idea is that while the result is a minor improvement, such re-training will take a lot of time, almost as much as the original training. In this paper, we show, both theoretically and experimentally, that in such situations, we can speed up re-training – practically without decreasing resulting accuracy – if we only update some weights.

1 Formulation of the Problem

Need for machine learning: a brief reminder. In many real-life situations, the value of a quantity y is, to a large extent, determined by the values of related quantities $x_1, \ldots, x_n$. This situation is typical for predictions, where the future value y of some quantity (e.g., tomorrow's temperature) is largely determined by today's values of temperature, wind speed, humidity, etc., at this location and at nearby locations.

In some cases, we know explicit formulas – or at least effective algorithms – for determining y based on $x_1, \ldots, x_n$; this is, e.g., the case of celestial mechanics, where we can predict solar eclipses hundreds of years ahead. However, in many other cases, no such formula or algorithm is known. In such cases, all we have is many (P) cases when we know both the values $x_1^{(p)}, \ldots, x_n^{(p)}$ of the input and the

Saeid Tizpaz-Niari and Vladik Kreinovich
Department of Computer Science, University of Texas at El Paso, 500 W. University
El Paso, Texas 79968, USA, e-mail: saeid@utep.edu, vladik@utep.edu

value $y^{(p)}$ of the output. Based on these patterns $\left(x_1^{(p)},\ldots,x_n^{(p)},y^{(p)}\right)$, we need to find an algorithm $f(x_1,\ldots,x_n)$ for which, for each of these patterns $p=1,\ldots,P$, we have

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

Finding such algorithm based on the given patterns is known as *regression* in statistics and as *machine learning* in computer science.

Deep learning: a brief reminder. At present, one of the most effective machine learning tools is *deep learning* that uses deep (multi-layer) networks of (artificial) neurons. In general, in a neural network, each neuron – except for the very last one – transforms its inputs $z_1,\ldots,z_m$ into the output

$$t = s(w_1\cdot z_1+\ldots+w_m\cdot z_m+w_0),$$

where:

- $s(z)$ is a given nonlinear function called *activation function*, and
- w_i are numerical parameters called *weights*.

The last neuron simply returns the value $y=w_1\cdot z_1+\ldots+w_m\cdot z_m+w_0$, without any additional nonlinear transformation.

Neurons are usually divided into layers:

- Neurons from the first layer process the original data $x_1,\ldots,x_n$ (e.g., the original measurement results).
- Neurons from the second layer use, as inputs, outputs of the neurons from the first layer.
- In general, neurons from the $(\ell+1)$-st layer use, as inputs, outputs of the neurons from the ℓ-th layer.

The output(s) of the neuron(s) of the last layer is the final answer that is returned to the user.

- In some cases, there are only two layers; such neural networks are called *shallow*. This was a traditional way neural networks were used in the past, see, e.g., [3]. Shallow neural networks usually use the activation function $s(z)=1/(1+\exp(-z))$ called *sigmoid*.
- When a neural network contains a reasonably large number of layers, it is called *deep*. In deep learning, the neurons use the activation function $s(z)=\max(0,z)$. This function is known as *rectified linear unit*, or *ReLU*, for short.

The weights w_i are selected so as to minimize, for each of the given patterns p, some measure of difference between the desired output $y^{(p)}$ and the result of applying the network with current weights to this pattern's inputs $x_1^{(p)},\ldots,x_n^{(p)}$. This minimization is usually performed by gradient descent, with a special algorithm called *backpropagation* that speed up the computation of the corresponding gradients.

Terminological comment. The only layer whose results are visible to the user is the last layer. Because of this, all other layers are known as *hidden* layers. In these terms:

- a shallow neural network contains only one hidden layer, while
- a deep neural network contains a reasonable large number of hidden layers.

Computational comment. Usually, as we have mentioned, the inputs to each neuron on the $(\ell+1)$-st layer comes from outputs of the neurons on the previous level ℓ. However, sometimes, it is convenient to add, as additional inputs for this neuron, some outputs from the previous layers $\ell-1$, $\ell-2$, all the way to (some of the) input signals $x_1,\ldots,x_n$. Such neural networks are called *residual*. A simple explanation of why residual networks are sometimes more effective is given, e.g., in [6].

Often, we need to add an extra variable. The list of inputs $x_1,\ldots,x_n$ is usually limited to quantities whose values are available – and about which we know that they affect the value of the quantity y. This list may miss some quantities whose use may lead to a better prediction of y:

- either because we do not know that the use of this extra variable will be useful,
- or because we do not know how to measure the corresponding quantity.

Later on, we may learn that this new variable is useful – e.g., we may learn that one of the obscure numbers included in a routin blood test may help to better diagnose some disease.

In such situation, we face the following problem:

- we already have a neural network trained to predict the desired value y based on the values of the quantities $x_1,\ldots,x_n$;
- we also have some patterns $\left(x_1^{(p)},\ldots,x_n^{(p)},x_{n+1}^{(p)},y^{(p)}\right)$ that include the values of the new variable x_{n+1};
- we would like to have a neural network trained to predict y based on the values of all available quantities $x_1,\ldots,x_n,x_{n+1}$, including the new quantity x_{n+1}.

A straightforward idea, its limitations, and the resulting problem. A natural idea is to start with the current trained neural network (with n inputs):

- we add a new input to the first layer, with (e.g.) random weights from x_{n+1} to all the neurons in the first layer, and then
- we use backpropagation to train the resulting network based on the newly available patterns.

The problem is that, in general, training takes a long time. There is not much that we can do in general, when we start "from scratch". However, in our situation, we are not starting from scratch, we start with the model that already has good predictions, we are just making small improvements to these predictions.

In general, in numerical computations, the knowledge of an approximate solution enables us to speed up computations in comparison with situations when no approximate solution is known and we need to start from scratch. A natural question is: can we speed up this re-training?

What we do in this paper. In this paper, we show that it is indeed possible to speed up the re-training caused by adding an extra quantity.

2 Analysis of the Problem

Main idea. The only way to speed up re-training is *not* to perform the *full* backpropagation, i.e., in effect, *not* to reach the absolute minimum of the objective function. In other words, if we do not use full re-training, the resulting network may be not as accurate as it could be.

This may not be so bad if we take into account that neural networks provide, in general, only an approximate description of the actual dependence. So:

- if the inaccuracy caused by not doing the full re-training is comparable with the usual approximation errors of the usual neural networks,
- then this minor inaccuracy is quite acceptable.

This acceptability can be explained on a simple example:

- if we measure the car's weight with an accuracy of 1 kg, and
- then we add load to it to find the total weight of the loaded car,

it does not make sense to measure the weight of the load with a 1 gram accuracy: when we weigh the load, the accuracy of 1 kg is quite sufficient.

To use this idea, let us analyze how accurately can a generic function be approximated by a neural network. For this purpose, let us first analyze how accurately a function can be approximated in general.

How accurately can a function be approximated in general? In real life, most dependencies are analytical – or at least well described by analytical functions; see, e.g., [4, 7]. A natural way to approximate such functions is to expand the corresponding expression into Taylor series and to keep the first few terms in this expansion. If we keep only linear terns, we get an expression with $n+1$ parameters a_i:

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

If we also retain quadratic terms, then we get an expression with $O(n^2)$ parameters:

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

If we also retain cubic terms, we need $O(n^3)$ parameters:

$$f(x_1,\dots,x_n) = a_0 + \sum_{1\le i\le n} a_i\cdot x_i + \sum_{1\le i\le j\le n} a_{i,j}\cdot x_i\cdot x_j + \sum_{1\le i\le j\le k\le n} a_{i,j,k}\cdot x_i\cdot x_j\cdot x_k. \quad (1)$$

To get a more accurate representation, we can also retain 4th order terms, this will require $O(n^4)$ parameters:

$$f(x_1,\dots,x_n) = a_0 + \sum_{1\le i\le n} a_i\cdot x_i + \sum_{1\le i\le j\le n} a_{i,j}\cdot x_i\cdot x_j + \sum_{1\le i\le j\le k\le n} a_{i,j,k}\cdot x_i\cdot x_j\cdot x_k + \sum_{1\le i\le j\le k\le \ell\le n} a_{i,j,k,\ell}\cdot x_i\cdot x_j\cdot x_k\cdot x_\ell.$$

How much accuracy can we attain with different approximation schemes? We would like to know how accurate is the approximation provided by a deep neural network. To answer this question, let us analyze the general situation: what accuracy can we attain if we a general approximation scheme

$$F(x_1,\dots,x_n,c_1,\dots,c_N)$$

with N parameters?

Clearly, potential accuracy depends on the number of parameters: the larger the number of parameters, the more accurately we can approximate different functions. To analyze how exactly the approximation accuracy depends on the number of parameters, let us take into account that we described approximation accuracy in terms of the numbers of terms in the Taylor expansion that are accurately reproduced. From this viewpoint, a natural idea is to also expand the general approximating expression in Taylor series:

$$F(x_1,\dots,x_n,c_1,\dots,c_N) = A_0(c_1,\dots,c_N) + \sum_{1\le i\le n} A_i(c_1,\dots,c_N)\cdot x_i + \dots$$

If we have $N = n+1$, then, in principle, we can perfectly fit all linear terms in the Taylor expansion of the function $f(x_1,\dots,x_n)$ that we want to approximate. Indeed, for this to be possible, we need to satisfy the following $n+1$ equations:

$$A_0(c_1,\dots,c_N) = a_0,\ \ A_1(c_1,\dots,c_N) = a_1,\ \ \dots,\ \ A_n(c_1,\dots,c_N) = a_n.$$

Here, we have $N = n+1$ equations to determine $N = n+1$ unknowns $c_1,\dots,c_N$. In general:

- If the number of equations is equal to (or smaller than) the number of unknowns, the system *has* a solution. This is true in the generic case of a linear system, and – due to the possibility of linearization – it is usually true for nonlinear systems as well.

- On the other hand, if we have more equations than unknowns, then, in general, the corresponding system *does not have* a solution – even in the general case of linear equations.

In our case, the number of equations is equal to the number of unknowns. Thus, the above system of equations has a solution. Hence, in principle, with an approximating scheme with $N = n + 1$ parameters, we can accurately fit linear terms in the expansion of the function $f(x_1, \ldots, x_n)$.

On the other hand, with this approximation scheme, we cannot, in general, fit all quadratic terms as well. Indeed, exactly fit all these terms, we will also need to satisfy additional equations $A_{i,j}(c_1, \ldots, c_N) = a_{i,j}$ for all i and j for which $1 \le i \le j \le n$ – to the total of $c \cdot n^2$ equations. So, to find the coefficients c_i, we would need to satisfy $c \cdot n^2$ equations. But since we only have $n + 1$ unknowns, the number of equations is much larger than the number of unknowns – so this system does not have a solution.

So, if we use an approximating scheme with $N = n + 1$ parameters, we fit all linear terms, but this approximation ignores quadratic (and higher order) terms.

Similarly, if we use an approximation scheme with $N = n^2$ parameters c_i, then we can always have a solution to $O(n^2)$ equations corresponding to matching all the $O(n^2)$ coefficients a_0, a_i, and $a_{i,j}$: since in this case, the number of equations is smaller than the number of unknowns. However, in this case, we cannot exactly fit cubic terms – this would mean satisfying $c \cdot n^3$ equations, and we have much fewer unknown than that: $n^2 \ll c \cdot n^3$. So, with $N = n^2$ parameters:

- we can fit all quadratic terms, and
- the largest ignored terms are cubic terms.

Same arguments show that if we use an approximation scheme with $N = n^3$ parameters, then:

- we can perfectly fit all cubic terms, and
- the largest ignored terms are 4th order terms, etc.

From this viewpoint, to find out how accurate is the approximation provided by a deep neural network, it is necessary to analyze how many parameters this approximation scheme has.

How many parameters does the deep learning approximation has: a rough estimate. In general, in a deep neural network, to process a reasonably large number n of inputs, we use a reasonably large number of layers, with each layer containing a reasonably large number of neurons. In this phrase, we use the term "reasonably large" three times:

- to describe the number of inputs,
- to describe the number of layers, and
- yo describe the number of neurons in each layer.

In general, these numbers may be different. However, for the purpose of providing a rough estimate, let us assume that these numbers are equal. In other words, we

assume that we have n layers, each of which has exactly n neurons in this layer. (The only exception of the last layer: since we want to output a single number y, the last layer contains only one neuron.)

How many parameters do we have here? In a deep neural networks, parameters are weights. Each of n inputs x_i can becomes an input to each of n neurons j in the first layer, with some weight $w_{j,i}$. Thus, to fully describe all the weights of the first layer, we need to describe $n \cdot n = n^2$ parameters. Similarly, for each layer k, the output of each of n neurons i from this layer can serve as the input to each of n neurons j in the next layer, with some weights $w_{j,i}$ – so, again, we have n^2 parameters. (Here too, the exception is the last layer – it only has n parameters.)

So, we have n layers, and to describe each layer, we need n^2 parameters. Thus, overall, we need $n \cdot n^2 = n^3$ parameters to describe a deep neural network.

So what is the resulting accuracy. We know that deep neural network contains about n^3 parameters. Thus, based on our general analysis of approximation schemes, we can conclude that:

- a deep neural network can perfectly describe all cubic terms in the expansion of the desired function $f(x_1,\ldots,x_n)$,
- while the 4th order terms will be ignored.

In other words, we get an approximate expression of the type (1).

Comment. If, instead of a deep neural network, we had a "shallow" network, with only one hidden layer, with n neurons in this layer, then this network would contain:

- n^2 parameters relating each of n inputs with each of n neurons in the hidden layer,
- n free terms w_0 of n neurons in the hidden layer,
- n parameters relating each neuron from the hidden later to the output neuron, and
- a free term of the output neuron,

to the total of $n^2 + 2n + 1 = (n+1)^2$ parameters. This number is larger than the number of coefficient in the general quadratic expression, but much smaller than the number of coefficients in the general cubic expression. Thus, this shallow network would be able to fit quadratic terms – but already cubic terms will not be covered.

What if we add an extra variable x_{n+1}? If we add an extra variable x_{n+1}, then, instead of the original expression (1), we have a similar expression, but with $n+1$ variables instead of the original n ones:

$$f(x_1,\ldots,x_n,x_{n+1}) = a_0 + \sum_{1\le i\le n+1} a_i\cdot x_i + \sum_{1\le i\le j\le n+1} a_{i,j}\cdot x_i\cdot x_j + \sum_{1\le i\le j\le k\le n+1} a_{i,j,k}\cdot x_i\cdot x_j\cdot x_k, \tag{2}$$

i.e., if we separate the dependence on x_{n+1}:

$$f(x_1,\ldots,x_n,x_{n+1} = a_0 + \sum_{1\le i\le n} a_i\cdot x_i + a_{n+1}\cdot x_{n+1} + \sum_{1\le i\le j\le n} a_{i,j}\cdot x_i\cdot x_j +$$

$$\sum_{1\le i\le n} a_{i,n+1}\cdot x_i\cdot x_{n+1} + a_{n+1,n+1}\cdot x_{n+1}^2 + \sum_{1\le i\le j\le k\le n} a_{i,j,k}\cdot x_i\cdot x_j\cdot x_k +$$
$$\sum_{1\le i\le j\le n} a_{i,j,n+1}\cdot x_i\cdot x_j\cdot x_{n+1} +$$
$$\sum_{1\le i\le n} a_{i,n+1,n+1}\cdot x_i\cdot x_{n+1}^2 + a_{n+1,n+1,n+1}\cdot x_{n+1}^3.$$

What does the neural network learn if it can only use the first n inputs? Let us analyze what exactly the original neural network learns when it only uses the first n inputs $x_1,\dots,x_n$ during training.

If the neural network could use all $n+1$ inputs, then, for each combination of inputs $(x_1,\dots,x_n,x_{n+1})$, the neural network would see the corresponding value (2). Thus, if we had a sufficient number of patterns and spend sufficient time on training, we would have the network learning this expression (2).

In situations when the neural network does not have access to the value x_{n+1}, then, for each combination of inputs $(x_1,\dots,x_n)$, the neural network will have several slightly different outputs

$$y^{(p)} = f\left(x_1,\dots,x_n,x_{n+1}^{(p)}\right) \tag{4}$$

corresponding to different values $x_{n+1}^{(p)}$ ($1\le p\le P_0$) of the extra quantity x_{n+1}. The result $F(x_1,\dots,x_n)$ of the training is determined by the condition that an appropriate quantity $Q(y^{(1)},\dots,y^{(P_0)},F(x_1,\dots,x_n))$ – that describes how close is the value $F(x_1,\dots,x_n)$ to all the observed outputs $y^{(1)},\dots,y^{(P_0)}$ – should be minimized. For example, if we minimize the least squares difference – as was typical for shallow neural networks, i.e., minimize the expression

$$Q(y^{(1)},\dots,y^{(P_0)},F(x_1,\dots,x_n)) = \sum_{1\le p\le P_0}\left(y^{(p)}-F(x_1,\dots,x_n)\right)^2,$$

then we get

$$F(x_1,\dots,x_n) = \frac{1}{P_0}\cdot\sum_{1\le p\le P_0} y^{(p)}.$$

In general, if we use a generic minimized expression Q, then we get a more complex expression of $F(x_1,\dots,x_n)$ as a function of the values $y^{(p)}$:

$$F(x_1,\dots,x_n) = J\left(y^{(1)},\dots,y^{(P_0)}\right).$$

If it so happens that for some tuple $(x_1,\dots,x_n)$, the dependence on x_{n+1} is negligible, i.e., if for all possible values of the extra variable x_{n+1}, we have $f(x_1,\dots,x_{n+1}) = f(x_1,\dots,x_n,\overline{x}_{n+1})$, where by $\widetilde{x}_{n+1}$ denotes a "typical" (e.g., average) value of x_{n+1}, then we will have $y^{(1)} = \dots = y^{(P_0)}$. In this case, the best fit

is attained when $F(x_1,\dots,x_n)$ is equal to all these values, i.e., when $F(x_1,\dots,x_n) = f(x_1,\dots,x_n,\overline{x}_{n+1})$.

It is therefore reasonable to expand the dependence J of the optimal value $F(x_1,\dots,x_n)$ on the values $y^{(p)}$ in Taylor series around the point

$$\left(y^{(1)},\dots,y^{(P_0)}\right) = (f(x_1,\dots,x_n,\overline{x}_{n+1}),\dots,f(x_1,\dots,x_n,\overline{x}_{n+1})).$$

This way, we get

$$F(x_1,\dots,x_n) = f(x_1,\dots,x_n,\overline{x}_{n+1}) + \sum_{1\le p\le P_0} J_p\cdot\Delta y^{(p)} + \sum_{1\le p\le p'\le P_0} J_{p,p'}\cdot\Delta y^{(p)}\cdot\Delta y^{(p')} + \sum_{1\le p\le p'\le p''\le P_0} J_{p,p',p''}\cdot\Delta y^{(p)}\cdot\Delta y^{(p')}\cdot\Delta y^{(p'')}, \tag{5}$$

where we denoted

$$\Delta y^{(p)} \stackrel{\text{def}}{=} y^{(p'')} - f(x_1,\dots,x_n,\overline{x}_{n+1}).$$

Since we are interested only in the terms up to the third order – as we have found out, higher order terms are ignored anyway – it is sufficient to only consider terms up to this order in the expression (5).

Here, due to (4), we have

$$y^{(p)} - f(x_1,\dots,x_n,\overline{x}_{n+1}) = f\left(x_1,\dots,x_n,x_{n+1}^{(p)}\right) - f(x_1,\dots,x_n,\overline{x}_{n+1}).$$

In general, substituting the explicit expression (3) for the function $f(x_1,\dots,x_n,x_{n+1})$ into the expression for the difference

$$f(x_1,\dots,x_n,x_{n+1}) - f(x_1,\dots,x_n,\overline{x}_{n+1}),$$

and taking into account that terms not depending on x_{n+1} cancel each other, we conclude that

$$f(x_1,\dots,x_n,x_{n+1}) - f(x_1,\dots,x_n,\overline{x}_{n+1}) = a_{n+1}\cdot(x_{n+1}-\overline{x}_{n+1}) + \sum_{1\le i\le n} a_{i,n+1}\cdot x_i\cdot(x_{n+1}-\overline{x}_{n+1}) + a_{n+1,n+1}\cdot\left(x_{n+1}^2-(\overline{x}_{n+1})^2\right) + \sum_{1\le i\le j\le n} a_{i,j,n+1}\cdot x_i\cdot x_j\cdot(x_{n+1}-\overline{x}_{n+1}) + \sum_{1\le i\le n} a_{i,n+1,n+1}\cdot x_i\cdot\left(x_{n+1}^2-(\overline{x}_{n+1})^2\right) + a_{n+1,n+1,n+1}\cdot\left(x_{n+1}^3-(\overline{x}_{n+1})^3\right). \tag{6}$$

In particular, the case when $x_{n+1} = x_{n+1}^{(p)}$, we get

$$y^{(p)} - f(x_1,\ldots,x_n,\overline{x}_{n+1}) = a_{n+1}\cdot\left(x_{n+1}^{(p)} - \overline{x}_{n+1}\right) +$$

$$\sum_{1\le i\le n} a_{i,n+1}\cdot x_i\cdot\left(x_{n+1}^{(p)} - \overline{x}_{n+1}\right) + a_{n+1,n+1}\cdot\left(\left(x_{n+1}^{(p)}\right)^2 - (\overline{x}_{n+1})^2\right) +$$

$$\sum_{1\le i\le j\le n} a_{i,j,n+1}\cdot x_i\cdot x_j\cdot\left(x_{n+1}^{(p)} - \overline{x}_{n+1}\right) +$$

$$\sum_{1\le i\le n} a_{i,n+1,n+1}\cdot x_i\cdot\left(\left(x_{n+1}^{(p)}\right)^2 - (\overline{x}_{n+1})^2\right) +$$

$$a_{n+1,n+1,n+1}\cdot\left(\left(x_{n+1}^{(p)}\right)^3 - (\overline{x}_{n+1})^3\right). \tag{7}$$

We can see that in this expression, all the terms are proportional to the difference $x_{n+1}^{(p)} - \overline{x}_{n+1}$. Thus, terms in (5) which are quadratic or of third order with respect to the difference $y^{(p)} - f(x_1,\ldots,x_n,\overline{x}_{n+1})$ are also proportional to the difference $x_{n+1}^{(p)} - \overline{x}_{n+1}$. Thus, since we are only interested in terms which are at most cubic in terms of all the variables $x_1,\ldots,x_n,x_{n+1}$, and each of these terms contains some value of x_{n+1}, all these terms are at most quadratic in terms of $x_1,\ldots,x_n$. So, from the formula (5), we conclude that the difference $F(x_1,\ldots,x_n) - f(x_1,\ldots,x_n,\overline{x}_{n+1})$ is a quadratic function of $x_1,\ldots,x_n$, i.e., that

$$F(x_1,\ldots,x_n) - f(x_1,\ldots,x_n,\overline{x}_{n+1}) = b_0 + \sum_{1\le i\le n} b_i\cdot x_i + \sum_{1\le i\le j\le n} b_{i,j}\cdot x_i\cdot x_j \tag{8}$$

for some coefficients b_0, b_i, and $b_{i,j}$. From the formulas (6) and (8), we conclude that the difference

$$\Delta f(x_1,\ldots,x_{n+1} \stackrel{\text{def}}{=} f(x_1,\ldots,x_n,x_{n+1}) - F(x_1,\ldots,x_n) =$$

$$(f(x_1,\ldots,x_n,x_{n+1}) - f(x_1,\ldots,x_n,\overline{x}_{n+1})) - (F(x_1,\ldots,x_n) - f(x_1,\ldots,x_n,\overline{x}_{n+1}))$$

has the form

$$\Delta f(x_1,\ldots,x_n,x_{n+1}) = a_{n+1}\cdot(x_{n+1} - \overline{x}_{n+1}) + \sum_{1\le i\le n} a_{i,n+1}\cdot x_i\cdot(x_{n+1} - \overline{x}_{n+1}) +$$

$$a_{n+1,n+1}\cdot\left(x_{n+1}^2 - (\overline{x}_{n+1})^2\right) + \sum_{1\le i\le j\le n} a_{i,j,n+1}\cdot x_i\cdot x_j\cdot(x_{n+1} - \overline{x}_{n+1}) +$$

$$\sum_{1\le i\le n} a_{i,n+1,n+1}\cdot x_i\cdot\left(x_{n+1}^2 - (\overline{x}_{n+1})^2\right) + a_{n+1,n+1,n+1}\cdot\left(x_{n+1}^3 - (\overline{x}_{n+1})^3\right) -$$

$$b_0 - \sum_{1\le i\le n} b_i\cdot x_i - \sum_{1\le i\le j\le n} b_{i,j}\cdot x_i\cdot x_j. \tag{9}$$

This expression contains the following unknowns:

- on parameter a_{n+1},

- n parameters $a_{i,n+1}$,
- one parameter $a_{n+1,n+1}$,
- $\dfrac{n\cdot(n+1)}{2}$ parameters $a_{i,j,n+1}$,
- n parameters $a_{i,n+1,n+1}$,
- one parameter $a_{n+1,n+1,n+1}$,
- one parameter b_0,
- n parameters b_i, and
- $\dfrac{n\cdot(n+1)}{2}$ parameters $b_{i,j}$.

to the total of

$$1+n+1+\frac{n\cdot(n+1)}{2}+n+1+1+\frac{n\cdot(n+1)}{2}=n^2+3\cdot n+2 \text{ parameters.}$$

Let us recall that a shallow network with $n+1$ inputs and $n+1$ neurons in the hidden layer, we have $(n+2)^2=n^2+4\cdot n+4$ parameters, and

$$n^2+4\cdot n+4>n^2+3\cdot n+2.$$

Thus, we can fit all the new cubic terms if we train a shallow neural network to recognize the difference between:

- the actual values $y^{(p)}$ of the output, and
- the values $F\left(x_1^{(p)},\ldots,x_n^{(p)}\right)$ produced by the pre-trained neural network (that does not take the new quantity x_{n+1} into account).

Then, the result of this training should be simply added to the result of pre-trained neural network.

Discussion. Specifically, the output of the shallow network should be added, to the pre-trained neural network, as an extra neuron in the penultimate layer, with weight 1 from this neuron to the final output neuron of the whole neural network. (Strictly speaking, this will make the resulting network residual.)

This will save time, since:

- training time is, crudely speaking, proportional to the number of parameters that we need to determine, and
- in our case, we decrease this number from n^3 – for the straightforward re-training – to the value n^2 needed to train the shallow network.

This way, we will cover all linear, quadratic, and cubic terms in the dependence of the output on all $n+1$ quantities $x_1,\ldots,x_n,x_{n+1}$. This will not cover 4th order terms – but these terms, as we have mentioned, are not covered by deep learning anyway.

Let us describe our proposal in precise terms.

3 Resulting Proposal

Formulation of the problem: reminder.

- *What we have:* We have a neural network pre-trained to describe the dependence of a quantity y on quantities $x_1, \ldots, x_n$. We will denote the result of applying this trained network to the inputs $x_1, \ldots, x_n$ by $F(x_1, \ldots, x_n)$.
- *What we want:* We would like to modify this network, so that it will take into account dependence on an additional quantity x_{n+1} as well.

A straightforward way to do it is to add one more input x_{n+1}, and to re-train all the weights of the whole original network by using all the patterns that contain the value of this input. The problem is that this would take a long time, so a question is: can we do it faster?

Specific recommendation. We can speed up the process if we do the following:

- First, we train a shallow neural network with n intermediate neurons to describe the dependence of the difference $Y^{(p)} = y^{(p)} - F\left(x_1^{(p)}, \ldots, x_n^{(p)}\right)$ on all the inputs $x_1^{(p)}, \ldots, x_n^{(p)}, x_{n+1}^{(p)}$ for which we know the value of the extra variable. Let us denote the result of applying the resulting trained shallow neural network to the inputs $x_1, \ldots, x_n, x_{n+1}$ by $S(x_1, \ldots, x_n, x_{n+1})$.
- Then, as a re-trained neural network, we take the network that computes the value $F(x_1, \ldots, x_n) + S(x_1, \ldots, x_n, x_{n+1})$. For this purpose, we:
  - add the output neuron of the shallow network to the penultimate layer of the original network, and
  - set the weight connecting this new neuron to the output layer of the original network to 1.

Further discussion. The above recommendation is the one that is formally justified by our analysis. However, less formally, we can say that the above specific scheme encourages us to use similar simplified re-training schemes. Indeed, our main idea was that:

- since we need to determine n^2 new parameters,
- we should not re-train all n^3 weights, it is most probably sufficient to only change n^2 weights.

How can we find a part of the network that contains exactly n^2 weights? This is easy: as we have mentioned earlier, each layer of a deep neural network contains n^2 weights. Thus, a reasonable idea is to re-train only one layer, while leaving all other weights unchanged ("frozen").

Which layer should we choose? We need to involve an additional input x_{n+1}.

Resulting idea. So, a reasonable idea is to train only the weights of the first layer – the layer that (directly) processes the inputs – while keeping all other weights unchanged. This general idea is what we tried.

4 Experiments

Our preliminary results show that this faster training indeed leads to results which are as accurate as the full training. For this testing, we use the following example from pavement engineering, where the goal is to estimate the value of the failure function – that gauges the stability f of an untreated pavement layer [1, 2]. Traditionally, this function is estimated based on four inputs: the three principal stresses σ_1, σ_2, and σ_3, and the angle of internal friction φ. The algorithm for estimating f based these four values consists of the following steps:

- First, we compute the first two invariants of the stress tensor:

$$I_1 = \sigma_1 + \sigma_2 + \sigma_3, \text{ and } J_2 = \frac{1}{6} \cdot \left[(\sigma_1 - \sigma_2)^2 + (\sigma_1 - \sigma_3)^2 + (\sigma_2 - \sigma_3)^2 \right].$$

- Based on the inputs σ_i and on the first invariant I_1, we compute the value of the third invariant:

$$J_3 = \left[\sigma_1 - \frac{I_1}{3} \right] \cdot \left[\sigma_2 - \frac{I_1}{3} \right] \left[\sigma_3 - \frac{I_1}{3} \right].$$

- Then, we compute the angle θ based on the formula

$$\cos(3 \cdot \theta) = \frac{3 \cdot \sqrt{3}}{2} \cdot \frac{J_3}{J_2^{3/2}}.$$

- Finally, we estimate the value of the failure function f as

$$f_0 = \frac{I_3}{2} \cdot \sin(\varphi) + \sqrt{J_2} \cdot \sin\left(\theta + \frac{\pi}{2}\right) + \frac{\sqrt{J_3}}{3} \cdot \cos\left(\theta + \frac{\pi}{2}\right) \cdot \sin(\varphi). \tag{10}$$

A slightly more accurate estimate can be obtained if we take into account the value of an additional quantity: cohesion c. The corresponding formula has the form

$$f = f_0 - c \cdot \cos(\varphi). \tag{11}$$

To test our idea, for a large number of randomly selected tuples $(\sigma_1^{(p)}, \sigma_2^{(p)}, \sigma_3^{(p)}, \varphi^{(p)}, c^{(p)})$. For each of these tuples, we used the formula (11) to computed the corresponding value $f^{(p)}$ of the failure function. In line with the above idea:

- **Task 1 (Without c)**: We trained the neural network on the patterns $(\sigma_1^{(p)}, \sigma_2^{(p)}, \sigma_3^{(p)}, \varphi^{(p)}, y^{(p)})$ that only included the values of the four original inputs.
- **Task 2 (1 layer training)**: We added an extra input corresponding to the extra variable c, froze the weights of all the layers except for the first one – so that only weights in the first layer that connects the input variable c to the next layer are changed – and trained these weights on the full tuples $(\sigma_1^{(p)}, \sigma_2^{(p)}, \sigma_3^{(p)}, \varphi^{(p)}, c^{(p)}, y^{(p)})$, including the values of the extra variable c.

- **Task 3 (full network training)**: For comparison, we performed a similar re-training without freezing, when the weights in all the layers were allowed to change during training.

For each of the three training tasks, we used a feedforward neural network with four internal layers $[128 \times 64 \times 32 \times 16]$ using the Adam optimizer with a learning rate of 0.001, a batch size of 64, and mean squared error as the loss function. To froze the weights, we customized PyTorch library to define layers, linear operators, and ReLU non-linear activation function. We used `torch.no_grad()` option to froze every weights and layers except ones that connect the new variable to the next layer. Upon acceptance, we will release the source code of our implementations.

# of iterations	500	1,000	1,500	2,000
Loss (without c)	4.2	0.2	0.2	0.2
Accuracy (without c)	47.0%	97.0%	98.0%	99.0%
R-Regression (without c)	0.82	0.99	0.99	0.99
Training Time (without c)	147(s)	314(s)	436(s)	600(s)
Loss (1-layer training)	0.4	0.6	0.9	0.1
Accuracy (1-layer training)	85.6%	83.0%	81.4%	99.9%
R-Regression (1-layer training)	0.98	0.98	0.97	0.99
Training Time (1-layer training)	148(s)	292(s)	405(s)	552(s)
Loss (full training)	2.8	1.3	0.2	0.4
Accuracy (full training)	53.0%	72.2%	97.0%	86.0%
R-Regression (full training)	0.91	0.95	0.99	0.98
Training Time (full training)	164(s)	300(s)	410(s)	555(s)

Table 1 Experimental results.

Table 1 shows the results of our experiments. For each training, we ran 2,000 iterations and recorded the results after 500, 1000, 1500, and 2000 iterations (first row). The first part of table (rows 2 to 5) shows the results without having the variable c (Task 1), the second part of table (rows 6 to 9) shows the results with adaptive training of first layer after adding the variable c, and the third part of table (rows 10 to 13) shows the results of full network training after adding the variable c. We reported the loss, the accuracy that measure how many predictions were within the unit distance of ground truth, the R^2 metric that measures the correlations between predictions and ground truth, and the computation times of training in seconds. We fixed the seed to minimize the randomness of functional outcomes (loss, accuracy, and R^2). To control the noise of environment for time measurements, we used an isolated server computer with Intel Xeon CPU with 2 vCPUs (virtual CPUs) and 13GB of RAM.

We observed that the proposed approach of adaptive training of first layer achieves better computation times where the differences in the computation times are reduced as the number of iterations increase. Within 2,000 iterations, the three approaches (without c, first layer training, and full training) achieve 99.0%, 99.9%,

and 97% accuracy, respectively. Since, there are fluctuations in the accuracy, one can use an early stop in training the network, and the highest accuracy shows the best results that can be achieved in each approach.

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, and by the AT&T Fellowship in Information Technology.

It was also supported by the program of the development of the Scientific-Educational Mathematical Center of Volga Federal District No. 075-02-2020-1478, and by a grant from the Hungarian National Research, Development and Innovation Office (NRDI).

The authors are thankful to Reza Ashtiani and Edgar Rodriguez for their help with the pavement engineering example.

References

1. R. S. Ashtiani, *Anisotropic Characterization and Performance Prediction of Chemically and Hydraulically Bounded Pavement Foundations*, Doctoral dissertation, Texas A&M University. Available electronically from: https://hdl.handle.net/1969.1/ETD-TAMU-2009-08-7103
2. R. S. Ashtiani, R. Luo, and R. L. Lytton, "Performance prediction and moisture susceptibility of anisotropic pavement foundations", In: B. Huang, E. Tutumluer, I. L. Al-Qadi, J. Prozzi, and X. Shu (eds.), *Paving Materials and Pavement Analysis: Proceedings of the GeoShanghai 2010 Conference, Shanghai, China, June 3–5, 2010* American Society of Civil Engineers (ASCE), Reston, Virginia, 2010, pp. 327–334.
3. C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, New York, 2006.
4. R. Feynman, R. Leighton, and M. Sands, *The Feynman Lectures on Physics*, Addison Wesley, Boston, Massachusetts, 2005.
5. I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, Cambridge, Massachusetts, 2016.
6. S. Holguin and V. Kreinovich, "Why residual neural networks", In: M. Ceberio and V. Kreinovich (eds.), *Decision Making under Uncertainty and Constraints: A Why-Book*, Springer, Cham, Switzerland, 2023, pp. 117–120.
7. K. S. Thorne and R. D. Blandford, *Modern Classical Physics: Optics, Fluids, Plasmas, Elasticity, Relativity, and Statistical Physics*, Princeton University Press, Princeton, New Jersey, 2021.