

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

Saeid Tizpaz-Niari and Vladik Kreinovich

Department of Computer Science, University of Texas at El Paso
500 W. University, El Paso, Texas 79968, USA, saeid@utep.edu, vladik@utep.edu

1. 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, \dots, 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, \dots, 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.

2. Need for machine learning: a brief reminder (cont-d)

- In such cases, all we have is many (P) cases when we know both:
 - the values $x_1^{(p)}, \dots, x_n^{(p)}$ of the input and
 - the value $y^{(p)}$ of the output.
- Based on these patterns $(x_1^{(p)}, \dots, x_n^{(p)}, y^{(p)})$, we need to find an algorithm $f(x_1, \dots, x_n)$ for which:
 - for each of these patterns $p = 1, \dots, P$,
 - we have $y^{(p)} \approx f(x_1^{(p)}, \dots, x_n^{(p)})$.
- Finding such algorithm based on the given patterns is known:
 - as *regression* in statistics and
 - as *machine learning* in computer science.

3. 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, \dots, z_m into the output
$$t = s(w_1 \cdot z_1 + \dots + w_m \cdot z_m + w_0).$$
- Here:
 - $s(z)$ is a given nonlinear function called *activation function*, and
 - w_i are numerical parameters called *weights*.
- The last neuron simply returns the value $y = w_1 \cdot z_1 + \dots + w_m \cdot z_m + w_0$, without any additional nonlinear transformation.
- Neurons are usually divided into layers.
- Neurons from the first layer process the original data x_1, \dots, x_n (e.g., the original measurement results).

4. Deep learning: a brief reminder (cont-d)

- 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.
- 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*.

5. Deep learning: a brief reminder (cont-d)

- 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 $x_1^{(p)}, \dots, x_n^{(p)}$.
- This minimization is usually performed by gradient descent:
 - with a special algorithm called *backpropagation*
 - that speed up the computation of the corresponding gradients.
- 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.

6. Deep learning: a brief reminder (cont-d)

- 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.

7. Computational comment

- Usually, the inputs to each neuron on the $(\ell + 1)$ -st layer come 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, \dots, x_n .
- Such neural networks are called *residual*.

8. Often, we need to add an extra variable

- The list of inputs x_1, \dots, 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.
- For example, we may learn that one of the obscure numbers included in a routine blood test may help to better diagnose some disease.

9. Often, we need to add an extra variable (cont-d)

- 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, \dots, x_n ;
 - we also have some patterns $\left(x_1^{(p)}, \dots, x_n^{(p)}, x_{n+1}^{(p)}, y^{(p)}\right)$ 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, \dots, x_n, x_{n+1} .

10. 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.

11. A straightforward idea, its limitations, and the resulting problem (cont-d)

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

12. 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 neural network approximation errors,
 - then this minor inaccuracy is quite acceptable.

13. Main idea (cont-d)

- This acceptability can be explained on a simple example.
- Suppose that we measure the car's weight with an accuracy of 1 kg.
- Then we add load to it and want to find the total weight of the loaded car.
- In this case, 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.

14. How accurately can a function be approximated in general?

- In real life, most dependencies are analytical – or at least well described by analytical functions.
- 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 terms, we get an expression with $n + 1$ parameters a_i :

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

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

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

15. How accurately can a function be approximated in general (cont-d)

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

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

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

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

16. 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 use 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.

17. How much accuracy can we attain with different approximation schemes (cont-d)

- From this viewpoint, a natural idea is to also expand the general approximating expression in Taylor series:

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

- 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:

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

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

18. How much accuracy can we attain with different approximation schemes (cont-d)

- 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.
- 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.
- This is true 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.

19. How much accuracy can we attain with different approximation schemes (cont-d)

- Hence, in principle:
 - with an approximating scheme with $N = n + 1$ parameters,
 - we can accurately fit linear terms in the expansion of the function $f(x_1, \dots, x_n)$.
- On the other hand:
 - with this approximation scheme,
 - we cannot, in general, fit all quadratic terms as well.
- Indeed:
 - to exactly fit all these terms,
 - we will also need to satisfy additional equations $A_{i,j}(c_1, \dots, c_N) = a_{i,j}$ for all i and j for which $1 \leq i \leq j \leq n$.
- There is a total of $c \cdot n^2$ equations.

20. How much accuracy can we attain with different approximation schemes (cont-d)

- 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.
- However, this approximation ignores quadratic (and higher order) terms.

21. How much accuracy can we attain with different approximation schemes (cont-d)

- 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 $a_{i,j}$;
 - indeed, 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.

22. How much accuracy can we attain with different approximation schemes (cont-d)

- 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.

23. 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
 - to 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.

24. How many parameters does the deep learning approximation has: a rough estimate (cont-d)

- In other words, we assume that:
 - we have n layers, and
 - each layer has exactly n neurons in this layer.
- The only exception is 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.

25. How many parameters does the deep learning approximation has: a rough estimate (cont-d)

- 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.

26. 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 $f(x_1, \dots, x_n)$,
 - while the 4th order terms will be ignored.

27. Comment

- What if we have 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 layer to the output neuron, and
 - a free term of the output neuron,
- Totally, we have $n^2 + 2n + 1 = (n + 1)^2$ parameters.

28. Comment (cont-d)

- 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.
- However, already cubic terms will not be covered.

29. What if we add an extra variable x_{n+1} ?

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

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

30. What if we add an extra variable x_{n+1} (cont-d)

- Let us separate the dependence on x_{n+1} :

$$\begin{aligned} f(x_1, \dots, x_n, x_{n+1}) = & a_0 + \sum_{1 \leq i \leq n} a_i \cdot x_i + a_{n+1} \cdot x_{n+1} + \sum_{1 \leq i \leq j \leq n} a_{i,j} \cdot x_i \cdot x_j + \\ & \sum_{1 \leq i \leq n} a_{i,n+1} \cdot x_i \cdot x_{n+1} + a_{n+1,n+1} \cdot x_{n+1}^2 + \sum_{1 \leq i \leq j \leq k \leq n} a_{i,j,k} \cdot x_i \cdot x_j \cdot x_k + \\ & \sum_{1 \leq i \leq j \leq n} a_{i,j,n+1} \cdot x_i \cdot x_j \cdot x_{n+1} + \\ & \sum_{1 \leq i \leq n} a_{i,n+1,n+1} \cdot x_i \cdot x_{n+1}^2 + a_{n+1,n+1,n+1} \cdot x_{n+1}^3. \end{aligned}$$

31. 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 y .
- Thus, if we had a sufficient number of patterns and spend sufficient time on training, we would have the network learning this expression.
- What is a neural network does not have access to the value x_{n+1} ?

32. What does the neural network learn if it can only use the first n inputs (cont-d)

- Then:
 - for each combination of inputs (x_1, \dots, x_n) ,
 - the neural network will have several slightly different outputs $y^{(p)} = f(x_1, \dots, x_n, x_{n+1}^{(p)})$;
 - these values correspond to different values $x_{n+1}^{(p)}$ ($1 \leq p \leq P_0$) of the extra quantity x_{n+1} .
- The result $F(x_1, \dots, x_n)$ of the training is determined by the following condition.
- We need to minimize the 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 $y^{(1)}, \dots, y^{(P_0)}$.

33. What does the neural network learn if it can only use the first n inputs (cont-d)

- For example, we can minimize the least squares difference – as was typical for shallow neural networks, i.e., minimize the expression

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

- Then we get

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

- For generic Q , we get a more complex expression of $F(x_1, \dots, x_n)$ as a function of the values $y^{(p)}$:

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

- It may so happens that for some tuple (x_1, \dots, x_n) , the dependence on x_{n+1} is negligible.

34. What does the neural network learn if it can only use the first n inputs (cont-d)

- Then, for all possible values of the extra variable x_{n+1} , we have $f(x_1, \dots, x_{n+1}) = f(x_1, \dots, x_n, \bar{x}_{n+1})$.
- Here by \bar{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 $F(x_1, \dots, x_n) = f(x_1, \dots, x_n, \bar{x}_{n+1})$.
- 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

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

35. What does the neural network learn if it can only use the first n inputs (cont-d)

- This way, we get

$$\begin{aligned} F(x_1, \dots, x_n) &= f(x_1, \dots, x_n, \bar{x}_{n+1}) + \sum_{1 \leq p \leq P_0} J_p \cdot \Delta y^{(p)} + \\ &\quad \sum_{1 \leq p \leq p' \leq P_0} J_{p,p'} \cdot \Delta y^{(p)} \cdot \Delta y^{(p')} + \\ &\quad \sum_{1 \leq p \leq p' \leq p'' \leq P_0} J_{p,p',p''} \cdot \Delta y^{(p)} \cdot \Delta y^{(p')} \cdot \Delta y^{(p'')}, \end{aligned}$$

- Here, we denoted

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

- We are interested only in the terms up to the third order – as we have found out, higher order terms are ignored anyway.
- So, it is sufficient to only consider terms up to this order in the above expression.

36. What does the neural network learn if it can only use the first n inputs (cont-d)

- Here, we have

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

- Let us substitute the explicit expression for the function $f(x_1, \dots, x_n, x_{n+1})$ into the expression for the difference

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

- Terms not depending on x_{n+1} cancel each other, so we conclude that

$$\begin{aligned} f(x_1, \dots, x_n, x_{n+1}) - f(x_1, \dots, x_n, \bar{x}_{n+1}) &= a_{n+1} \cdot (x_{n+1} - \bar{x}_{n+1}) + \\ &\sum_{1 \leq i \leq n} a_{i,n+1} \cdot x_i \cdot (x_{n+1} - \bar{x}_{n+1}) + a_{n+1,n+1} \cdot (x_{n+1}^2 - (\bar{x}_{n+1})^2) + \\ &\sum_{1 \leq i \leq j \leq n} a_{i,j,n+1} \cdot x_i \cdot x_j \cdot (x_{n+1} - \bar{x}_{n+1}) + \sum_{1 \leq i \leq n} a_{i,n+1,n+1} \cdot x_i \cdot (x_{n+1}^2 - (\bar{x}_{n+1})^2) + \\ &a_{n+1,n+1,n+1} \cdot (x_{n+1}^3 - (\bar{x}_{n+1})^3). \end{aligned}$$

37. What does the neural network learn if it can only use the first n inputs (cont-d)

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

$$\begin{aligned} y^{(p)} - f(x_1, \dots, x_n, \bar{x}_{n+1}) &= a_{n+1} \cdot \left(x_{n+1}^{(p)} - \bar{x}_{n+1} \right) + \\ \sum_{1 \leq i \leq n} a_{i,n+1} \cdot x_i \cdot \left(x_{n+1}^{(p)} - \bar{x}_{n+1} \right) &+ a_{n+1,n+1} \cdot \left(\left(x_{n+1}^{(p)} \right)^2 - (\bar{x}_{n+1})^2 \right) + \\ \sum_{1 \leq i \leq j \leq n} a_{i,j,n+1} \cdot x_i \cdot x_j \cdot \left(x_{n+1}^{(p)} - \bar{x}_{n+1} \right) &+ \\ \sum_{1 \leq i \leq n} a_{i,n+1,n+1} \cdot x_i \cdot \left(\left(x_{n+1}^{(p)} \right)^2 - (\bar{x}_{n+1})^2 \right) &+ \\ a_{n+1,n+1,n+1} \cdot \left(\left(x_{n+1}^{(p)} \right)^3 - (\bar{x}_{n+1})^3 \right). \end{aligned}$$

- We can see that in this expression, all the terms are proportional to the difference $x_{n+1}^{(p)} - \bar{x}_{n+1}$.

38. What does the neural network learn if it can only use the first n inputs (cont-d)

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

$$F(x_1, \dots, x_n) - f(x_1, \dots, x_n, \bar{x}_{n+1}) = b_0 + \sum_{1 \leq i \leq n} b_i \cdot x_i + \sum_{1 \leq i \leq j \leq n} b_{i,j} \cdot x_i \cdot x_j \text{ for some } b_0, b_i, b_{i,j}$$

39. What does the neural network learn if it can only use the first n inputs (cont-d)

- We are interested in the difference

$$\Delta f(x_1, \dots, x_{n+1}) \stackrel{\text{def}}{=} f(x_1, \dots, x_n, x_{n+1}) - F(x_1, \dots, x_n) = \\ (f(x_1, \dots, x_n, x_{n+1}) - f(x_1, \dots, x_n, \bar{x}_{n+1})) - (F(x_1, \dots, x_n) - f(x_1, \dots, x_n, \bar{x}_{n+1})).$$

- So, this difference has the form:

$$\Delta f(x_1, \dots, x_n, x_{n+1}) = a_{n+1} \cdot (x_{n+1} - \bar{x}_{n+1}) + \sum_{1 \leq i \leq n} a_{i,n+1} \cdot x_i \cdot (x_{n+1} - \bar{x}_{n+1}) + \\ a_{n+1,n+1} \cdot \left(x_{n+1}^2 - (\bar{x}_{n+1})^2 \right) + \sum_{1 \leq i < j \leq n} a_{i,j,n+1} \cdot x_i \cdot x_j \cdot (x_{n+1} - \bar{x}_{n+1}) + \\ \sum_{1 \leq i \leq n} a_{i,n+1,n+1} \cdot x_i \cdot \left(x_{n+1}^2 - (\bar{x}_{n+1})^2 \right) + a_{n+1,n+1,n+1} \cdot \left(x_{n+1}^3 - (\bar{x}_{n+1})^3 \right) - \\ b_0 - \sum_{1 \leq i \leq n} b_i \cdot x_i - \sum_{1 \leq i < j \leq n} b_{i,j} \cdot x_i \cdot x_j.$$

40. What does the neural network learn if it can only use the first n inputs (cont-d)

- This expression contains the following unknowns:

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

- The total number is

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

41. What does the neural network learn if it can only use the first n inputs (cont-d)

- Let us recall that:
 - in 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 $F(x_1^{(p)}, \dots, x_n^{(p)})$ 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.

42. 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.
- Indeed, training time is, crudely speaking, proportional to the number of parameters that we need to determine.
- 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, \dots, x_n, x_{n+1} .

43. Discussion (cont-d)

- This will not cover 4th order terms.
- However, these terms, as we have mentioned, are not covered by deep learning anyway.
- Let us describe our proposal in precise terms.

44. 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, \dots, x_n .
- We will denote the result of applying this trained network to the inputs x_1, \dots, x_n by $F(x_1, \dots, x_n)$.
- *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?

45. Resulting proposal

- 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 $Y^{(p)} = y^{(p)} - F\left(x_1^{(p)}, \dots, x_n^{(p)}\right)$
 - on all the inputs $x_1^{(p)}, \dots, x_n^{(p)}, x_{n+1}^{(p)}$ 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, \dots, x_n, x_{n+1} by $S(x_1, \dots, x_n, x_{n+1})$.
- Then, as a re-trained neural network, we take the network that computes the value $F(x_1, \dots, x_n) + S(x_1, \dots, x_n, x_{n+1})$.

46. Resulting proposal (cont-d)

- 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.

47. 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”).

48. Further discussion (cont-d)

- Which layer should we choose?
- We need to involve an additional input x_{n+1} .
- So, a reasonable idea is:
 - to train only the weights of the first layer,
 - since this is the layer that (directly) processes the inputs.
- All other layers are kept unchanged.
- This general idea is what we tried.

49. 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.
- The goal is to estimate the value of the failure function – that gauges the stability f of an untreated pavement layer.
- 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:

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

50. Experiments (cont-d)

- 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

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

- 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 $f = f_0 - c \cdot \cos(\varphi)$.

51. Experiments (cont-d)

- To test our idea, we formed 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 above formula to compute the corresponding value $f^{(p)}$ of the failure function.
- **Task 1 (Without c):** We trained the neural network on the patterns $(\sigma_1^{(p)}, \sigma_2^{(p)}, \sigma_3^{(p)}, \varphi^{(p)}, y^{(p)})$ without c .
- **Task 2 (1 layer training):**
 - we added an extra input corresponding to the extra variable c ,
 - we froze the weights of all the layers except for the first one,
 - and trained the weights of the first layer on the full tuples $(\sigma_1^{(p)}, \sigma_2^{(p)}, \sigma_3^{(p)}, \varphi^{(p)}, c^{(p)}, y^{(p)})$.
- **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 $[128 \times 64 \times 32 \times 16]$,
 - the Adam optimizer with a learning rate of 0.001,
 - a batch size of 64, and
 - mean squared error as the loss function.
- To freeze the weights, we customized PyTorch library to define layers, linear operators, and ReLU non-linear activation function.
- We used `torch.no_grad()` option to freeze every weights and layers except ones that connect the new variable to the next layer.
- 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 .
- 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.
- 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.
- The highest accuracy shows the best results that can be achieved in each approach.

# 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.

52. Acknowledgments

This work was supported in part by:

- National Science Foundation grants 1623190, HRD-1834620, HRD-2034030, and EAR-2225395;
- AT&T Fellowship in Information Technology;
- program of the development of the Scientific-Educational Mathematical Center of Volga Federal District No. 075-02-2020-1478, and
- a grant from the Hungarian National Research, Development and Innovation Office (NRDI).