# Why embedding-decoder arrangement helps machine learning

Miroslav Svítek, Olga Kosheleva, Vladik Kreinovich, and Nguyen Hoang Phuong

**Abstract** Usually, in machine learning, we directly train the neural network (or any other machine learning tool) to generate the desired output for each given input. However, recently, an alternative two-stage approach was shown to be more efficient, when we first train the neural network to recognize pairs of input tuples that leads to the same output, and then train another machine learning tool (called decoder) to assign the desired value to each tuple. In this paper, we provide a theoretical explanation for this empirical success.

## 1 Formulation of the problem

**What is machine learning: a very brief reminder.** In general, machine learning is about learning a dependence $y = f(x_1, \ldots, x_n)$ based on known examples $\left(x_1^{(k)}, \ldots, x_n^{(k)}, y^{(k)}\right)$ of tuples that satisfy this dependence, i.e., tuples for which $y^{(k)} = f\left(x_1^{(k)}, \ldots, x_n^{(k)}\right)$.

Miroslav Svítek
Faculty of Transportation Sciences, Czech Technical University in Prague, Konviktska 20
CZ-110 00 Prague 1, Czech Republic, e-mail: svitek@fd.cvut.cz

Olga Kosheleva
Department of Teacher Education, University of Texas at El Paso, 500 W. University
El Paso, Texas 79968, USA, e-mail: olgak@utep.edu

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

Nguyen Hoang Phuong
Artificial Intelligence Division, Information Technology Faculty, Thang Long University
Nghiem Xuan Yem Road, Hoang Mai District, Hanoi, Vietnam
e-mail: nhphuong2008@gmail.com

**Traditional approach to machine learning.** In the traditional approach to machine learning, we directly train the machine learning tool – e.g., a deep neural network – to predict $y$ based on $x_i$'s; see, e.g., [1, 3].

**Embedding-decoder arrangement: an alternative to the traditional approach.** The paper [4], following ideas from [5], uses a different approach, in which:

- first, we use a machine learning tool to learn some embeddings of the inputs $x_i$ into values $X_i = f_i(x_i)$ and some function $F(X_1,\ldots,X_n)$ for which $F(f_1(x_1),\ldots,f_n(x_n)) = F(f_1(x'_1),\ldots,f_n(x'_n))$ if and only if $f(x_1,\ldots,x_n) = f(x'_1,\ldots,x'_n)$, and
- then, we use machine learning to come up with a *decoder* $d(z)$ for which, for all $x_i$, we have:
$$d(F(f_1(x_1),\ldots,f_n(x_n))) = f(x_1,\ldots,x_n).$$

On several examples, this two-stage approach is more efficient that the traditional approach to machine learning.

**A natural question:** why is this two-stage approach more efficient?

**What we do in this paper.** In this paper, we explain the efficiency of the two-stage approach.

## 2 Analysis of the problem and the resulting explanation

**First, let us simplify the formulation of the problem.** In order to explain the efficiency of the two-stage approach, let us reformulate it in a simpler form, ignoring details that – as we will show – are not critical from the viewpoint of our explanation. Specifically, instead of considering separately the functions that are solicited on the first stage of the two-stage process – i.e., functions $f_1(x_1)$, …, $f_n(x_n)$, and $f(X_1,\ldots,X_n)$ – let us consider a single function

$$G(x_1,\ldots,x_n) \stackrel{\text{def}}{=} F(f_1(x_1),\ldots,f_n(x_n)).$$

In these simplified terms, the two-stage process takes the following form:

- first, we look for a function $G(x_1,\ldots,x_n)$ for which $G(x_1,\ldots,x_n) = G(x'_1,\ldots,x'_n)$ if and only if $f(x_1,\ldots,x_n) = f(x'_1,\ldots,x'_n)$, i.e., equivalently, for which $f(x_1,\ldots,x_n) = d(G(x_1,\ldots,x_n))$ for some 1-1 function $d(z)$; and
- then, we look for the function $d(z)$.

On each stage, we have, in effect, a search problem: out of all possible alternatives (this time, alternatives are functions), we need to find a function that satisfies the desired property. So, the question becomes: why is the two-stage search more efficient than the usual one-stage search? To answer this question, let us analyze how to estimate the time complexity of a search problem.

**How to estimate the time complexity of a search problem?** In general, if we have $N$ alternatives, and the list of possible alternatives is not sorted in any way, then the exhaustive search takes, in the worst case, $N$ computational steps, and, on average, $N/2$ steps; see, e.g. [2]. For exhaustive search, the computation time $T$ is therefore proportional to $N$.

Of course, neural networks do not provide an exhaustive search of all possible functions – sometimes, they cannot find the function, and sometimes, they only provide an approximate function. Because of this, the actual computation time $t$ – be it worst-case or average-case – is usually smaller than $T$. The actual value of $t$ depends on the specifics of an algorithm. We do not know these specifics, so let us come up with estimates of the computation time $t$ based on our knowledge of the time $T$. In other words, we need to find an algorithm $a(T)$ that would provide, for each exhaustive search algorithm requiring time $T$, a reasonable estimate $a(T)$ for the time $t$ needed for a practical (non-exhaustive) search algorithm.

The numerical values of both times $t$ and $T$ depend on our choice of the measuring unit: e.g., we can measure computation time in second, in milliseconds, in minutes, etc. When we replace the original measuring unit by a new unit which is $c$ times smaller, than all numerical values multiply by $c$. For example, if we replace minutes with seconds – which is $c = 60$ times smaller unit – then 2 minutes becomes $60 \cdot 2 = 120$ seconds. In general, under this change, instead of $t$ and $T$, we get $t' = c \cdot t$ and $T' = c \cdot T$.

There is no reason to believe that one unit of time is better than the other. So, it makes sense to require that the function $t = a(T)$ should not change is we simply change the unit of time. In other words, if we have $t = a(T)$, then for every $c > 0$, we should have $t' = a(T')$, where $t' = c \cdot t$ and $T' = c \cdot T$. Let us describe this property in precise terms.

**Definition.** *We say that a function a from positive real numbers to positive real numbers is* unit-invariant *if for all possible $t > 0$, $T > 0$, and $c > 0$, when $t = a(T)$, then we have $t' = a(T')$, where we denoted $t' \stackrel{\text{def}}{=} c \cdot t$ and $T' \stackrel{\text{def}}{=} c \cdot T$.*

**Proposition.** *A function $a(T)$ is unit-invariant if and only if it has the form $a(T) = a_0 \cdot T$ for some constant $a_0 > 0$.*

**Proof.** It is easy to see that a linear function $a(T) = a_0 \cdot T$ is unit-invariant.

Vice versa, let us assume that we have a unit-invariant function. Substituting the expressions for $t'$ and $T'$ into the formula $t' = a(T')$, we conclude that $c \cdot t = a(c \cdot T)$. Here, $t = a(T)$, so this formula takes the form $c \cdot a(T) = a(c \cdot T)$. This equality should be true for all possible values $c > 0$ and $t > 0$. In particular, for $T = 1$, we get $a(c) = a_0 \cdot c$, where we denoted $a_0 \stackrel{\text{def}}{=} a(1)$. The proposition is proven.

**What we can conclude from this.** So, our reasonable estimate for the computation time of the actual algorithm takes the form $t = a_0 \cdot T$.

Now, we are ready to provide a comparative analysis of the computation time of both 1-stage and 2-stage searches.

**1-stage case.** Theoretically, we may have an infinite number of possible inputs, but in the computer, whatever input we consider, it is represented by a finite number of

0s and 1s, and the number of such bits (0s and 1s) is limited by the size $B$ bits of the corresponding areas. The number of such binary sequences is this limited by $2^B$ – the overall number of binary sequences of length $B$. So, in practice, we have a finite number of possible inputs and a finite number of possible outputs.

Let $X$ denote the overall number of possible inputs $(x_1,\ldots,x_n)$, and let $Y$ denote the overall number of all possible outputs $y$. Thus, if we do not impose any restrictions on the possible functions $f(x_1,\ldots,x_n)$, then the number of possible functions is equal to the number of possible functions from the set of $X$ elements to the set of $Y$ elements. For each of $X$ elements, we can have $Y$ possible values, so each function can be described as listing all $X$ such values. The number of such tuples of $y$-values is $Y\cdot\ldots\cdot Y$ ($X$ times), i.e., $Y^X$. Thus, exhaustive search would require time proportional to $Y^X$: $T=c_0\cdot Y^X$, for some coefficient $c_0$, and the actual time is equal to:

$$a_0\cdot T=a_0\cdot c_0\cdot Y^X.$$

**2-stage case.** On the first stage, we look not for a function, but for an equivalence class of functions – modulo 1-1 transformation, i.e., modulo permutations of the set of $y$'s. It is known that for a set of $Y$ elements, there are exactly

$$Y!\stackrel{\text{def}}{=}1\cdot 2\cdot\ldots\cdot Y$$

permutations. So, each equivalence class contains $Y!$ elements. (To be more precise, some classes will have fewer elements – e.g., a constant is not changing under permutation, so it forms a whole equivalence class, but such cases are rarely and can be, in the first approximation safely ignored.)

In this first approximation, to estimate the number $N_1$ of such equivalence classes, we need to divide the overall number $Y^X$ of functions by the number $Y!$ of elements in each class. As a result, we get:

$$N_1=\frac{Y^X}{Y!}.$$

So, the computation time for the first stage in the exhaustive case would be equal to:

$$T_1=c_0\cdot\frac{Y^X}{Y!},$$

and the computation time of a realistic search algorithm implementing the first stage will be equal to:

$$t_1=a_0\cdot T_1=a_0\cdot c_0\cdot\frac{Y^X}{Y!}.$$

In the second stage, we select one of the permutations. As we have mentioned, the number of permutations is equal to $N_2=Y!$, so the exhaustive-case computation time for the second case would be $T_2=c_0\cdot N_2=c_0\cdot Y!$, and the real computation time of the second stage will be $t_2=a_0\cdot T_2=a_0\cdot c_0\cdot Y!$.

Thus, the overall computation time $t$ of the two-stage process is equal to the sum of computation times of the two stages:

$$t = t_1 + t_2 = a_0 \cdot c_0 \cdot \frac{Y^X}{Y!} + a_0 \cdot c_0 \cdot Y! = a_0 \cdot c_0 \cdot \left(\frac{Y^X}{Y!} + Y!\right).$$

**Comparing the 1-stage and 2-stage computation times explains the empirical efficiency of the 2-stage approach.** For both approaches, the computation time is equal to $a_0 \cdot c_0$ times some expression:

- the expression $Y^X$ for the 1-stage case, and
- the expression
$$\frac{Y^X}{Y!} + Y!$$
for the 2-stage case.

One can easily check that while the 2-stage case expression is the sum of two terms, the 1-stage case expression is the product of the same two terms:

$$\frac{Y^X}{Y!} \cdot Y! = Y^X.$$

In general, the product of any two sufficiently large numbers $a$ and $b$ is always larger than their sum. Indeed, the difference between the product and the sum has the form:

$$a \cdot b - (a + b) = a \cdot (b - 1) - (b - 1) - 1 = (a - 1) \cdot (b - 1) - 1.$$

So, if $a > 2$ and $b > 2$, we have $a - 1 > 1$ and $b - 1 > 1$, thus $(a - 1) \cdot (b - 1) > 1$ and therefore, $(a - 1) \cdot (b - 1) - 1 > 0$, i.e., $a \cdot b - (a + b) > 0$ and indeed $a \cdot b > a + b$.

In our case, both $Y!$ and $Y^X/Y!$ are huge numbers – clearly larger than 2, so the expression for the 1-stage case if larger than the expression for the 2-stage case. Thus, the computation time needed for the 1-stage approach is larger than what is needed for the 2-stage approach. This explains why the 2-stage approach is empirically more efficient.

## 3 Acknowledgments

This work was supported in part by the National Science Foundation grants 1623190 (A Model of Change for Preparing a New Generation for Professional Practice in Computer Science), HRD-1834620 and HRD-2034030 (CAHSI Includes), EAR-2225395 (Center for Collective Impact in Earthquake Science C-CIES), and by the AT&T Fellowship in Information Technology.

It was also supported by a grant from the Hungarian National Research, Development and Innovation Office (NRDI), by the Institute for Risk and Reliability, Leibniz Universitaet Hannover, Germany, and by the European Union under the project ROBOPROX (No. CZ.02.01.01/00/22 008/0004590).

## References

1. C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, New York, 2006.
2. Th. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts, 2022.
3. I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, Cambridge, Massachusetts, 2016.
4. J. Liu, O. Kitouni, N. Nolte, E. J. Michaud, M. Tegmark, and M. Williams, "Towards understanding grokking: an effective theory of representation learning", *Proceedings of the 36th Conference on Neural Information Processing Systems NeurIPS 2022*, New Orleans, Louisiana, USA, November 28 – December 9, 2022, Article 2511, pp. 34651–34663.
5. A. Power, Y. Burda, H. Edwards, I. Babuschkin, and V. Misra,*Grokking: Generalization Beyond Overfitting on Small Algorithmic Datasets*, arXiv preprint arXiv:2201.02177, 2022.