Why embedding-decoder arrangement helps machine learning

Miroslav Svítek[1], Olga Koshevela[2],
Vladik Kreinovich[3], and Nguyen Hoang Phuong[4]

[1]Faculty of Transportation Sciences, Czech Technical University in Prague, Konviktska 20
CZ-110 00 Prague 1, Czech Republic, svitek@fd.cvut.cz
[2,3]Departments of [2]Teacherc Education and [3]Computer Science,
University of Texas at El Paso, 500 W. University,
El Paso, TX 79968, USA, olgak@utep.edu, vladik@utep.edu
[4]Artificial Intelligence Division, Information Technology Faculty,
Thang Long University Nghiem Xuan Yem Road, Hoang Mai District,
Hanoi, Vietnam, nhphuong2008@gmail.com

1. What is machine learning: a very brief reminder

- In general, machine learning is about:
  - earning 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 $y^{(k)} = f\left(x_1^{(k)}, \ldots, x_n^{(k)}\right)$.
- 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.

2. Embedding-decoder arrangement: an alternative to the traditional approach

- A recent paper uses a different approach.
- 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

$$F(f_1(x_1), \ldots, f_n(x_n)) = F(f_1(x_1'), \ldots, f_n(x_n')) \Leftrightarrow$$

$$f(x_1, \ldots, x_n) = f(x_1', \ldots, x_n').$$

- 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.

3. A natural question

- A natural question: why is this two-stage approach more efficient?
- In this talk, we explain the efficiency of the two-stage approach.

4. 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.
- We will ignore 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

$$f_1(x_1), \ldots, f_n(x_n), \text{ and } f(X_1, \ldots, X_n),$$

  - 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.

5. Let us simplify the formulation of the problem (cont-d)

- First, we look for a function $G(x_1, \ldots, x_n)$ for which

$$G(x_1, \ldots, x_n) = G(x'_1, \ldots, x'_n) \Leftrightarrow f(x_1, \ldots, x_n) = f(x'_1, \ldots, x'_n).$$

- This is equivalent to looking for a function $G(x_1, \ldots, x_n)$ for which $f(x_1, \ldots, x_n) = d(G(x_1, \ldots, x_n))$ for some 1-1 function $d(z)$.
- 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.

6. 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.
- 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.

7. How to estimate the time complexity of a search problem (cont-d)

- 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.

8. How to estimate the time complexity of a search problem (cont-d)

- 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

$$t' = c \cdot t \text{ and } T' = c \cdot T.$$

- 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.

9. Definition and Proposition

- *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 $c \cdot a(T) = a(c \cdot T)$.

10. Definition and Proposition (cont-d)

- 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.

11. 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.

12. 1-stage case

- Theoretically, we may have an infinite number of possible inputs.
- However, in a computer, whatever input we consider, it is represented by a finite number of 0s and 1s.
- Rhe 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.

13. 1-stage case (cont-d)

- 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:

$$T = c_0 \cdot Y^X, \text{ for some coefficient } c_0.$$

- So, the actual time is equal to: $a_0 \cdot T = a_0 \cdot c_0 \cdot Y^X$.

14. 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

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

- 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.
- However, such cases are rarely and can be, in the first approximation safely ignored.

15. 2-stage case (cont-d)

- 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 $N_1 = \dfrac{Y^X}{Y!}$.
- So, the computation time for the first stage in the exhaustive case would be equal to: $T_1 = c_0 \cdot \dfrac{Y^X}{Y!}$.
- Thus, the computation time of a realistic search algorithm implementing the first stage will be equal to $t_1 = a_0 \cdot T_1 = a_0 \cdot c_0 \cdot \dfrac{Y^X}{Y!}$.
- In the second stage, we select one of the permutations.
- As we have mentioned, the number of permutations is equal to

$$N_2 = Y!.$$

16. 2-stage case (cont-d)

- 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:

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

17. 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 $\dfrac{Y^X}{Y!} + Y!$ 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:

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

18. Comparing the computation times explains the empirical efficiency of the 2-stage approach (cont-d)

- So, if $a > 2$ and $b > 2$, we have $a - 1 > 1$ and $b - 1 > 1$, thus $(a - 1) \cdot (b - 1) > 1$.
- 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.

19. References

- 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.
- 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.

20. Acknowledgments

This work was supported in part by:

- Deutsche Forschungsgemeinschaft Focus Program SPP 100+ 2388, Grant Nr. 501624329;
- 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).