Faster Quantum Alternative to Softmax Selection in Deep Learning and Deep Reinforcement Learning

Oscar Galindo, Christian Ayub,
Martine Ceberio, and Vladik Kreinovich
Department of Computer Science
University of Texas at El Paso
500 W. University
El Paso, TX 79968, USA
ogalindomo@miners.utep.edu, cayub@miners.utep.edu,
mceberio@utep.edu, vladik@utep.edu

***Abstract*—Deep learning and deep reinforcement learning are, at present, the best available machine learning tools for use in engineering problems. However, at present, the use of these tools is limited by the fact that they are very time-consuming, usually requiring the use of a high performance computer. It is therefore desirable to look for possible ways to speed up the corresponding computations. One of the time-consuming parts of these algorithms is softmax selection, when instead of selecting the alternative with the largest possible value of the corresponding objective function, we select all possible values, with probabilities increasing with the value of the objective function. In this paper, we propose a significantly faster quantum-computing alternative to softmax selection.**

***Keywords*—Engineering applications; machine learning; deep reinforcement learning; softmax; quantum computing**

I. Formulation of the Problem: Need for a Faster Alternative to Softmax Selection

Need for machine learning in engineering and science. One of the main objectives of science and engineering in general is to find the state of the world, to predict the future state of the world, and to find the control and/or design that leads to a better future state. The state of the world can be described by values of the corresponding physical quantities. A control can be described by the values of the corresponding control parameters, and the details of a design can be described by the values of the corresponding design parameters. In these terms, the prediction and control problems can be described in a similar way:

- we know – e.g., from measurements – the values of some quantities $x_1, \ldots, x_n$, and
- we want to find the value of the desired quantity y based on the known values x_i.

In some cases, the situation is straightforward: we know the algorithm $f(x_1, \ldots, x_n)$ that enables us to estimate y based on the known values x_i, and this algorithm can be feasibly implemented.

In many cases, we do not have the equations, all we have is the past experience of observing the values x_i and the corresponding values y. In such cases, based on such records $\left(x_1^{(k)}, \ldots, x_n^{(k)}, y^{(k)}\right)$, $1 \le k \le K$, we need to find a feasible algorithm $f(x_1, \ldots, x_n)$ for which $y^{(k)} \approx f\left(x_1^{(k)}, \ldots, x_n^{(k)}\right)$ for all k.

The procedure of producing such an algorithm based on the known records is known as *machine learning*; see, e.g., [4], [8].

In many other situations, while the corresponding equations are known, the resulting system is so complex that the corresponding algorithm would produce the result way after the event that we are trying to predict. This is the case, e.g., of predicting tornado trajectories, and of many other practical problems. In such situations, we cannot use the model directly. Instead, we can use the results $y^{(k)}$ of running this model on different inputs $\left(x_1^{(k)}, \ldots, x_n^{(k)}\right)$ to form the corresponding records, and then apply machine learning to get a feasible algorithm based on these records. In this case, it is OK that this model takes too long to run – we can spend as much time as necessary on generating the records, as long as they help us to eventually come up with the feasible algorithm that we will later use for actual prediction or control.

Need for reinforcement learning. In many practical situations, the available records are not sufficient to reconstruct the algorithm $f(x_1, \ldots, x_n)$. In such situations, we need to perform additional measurements (or, in case of complex models, additional simulations). In this case, it is desirable that the computer will instruct us which values $(x_1, \ldots, x_n)$ to try next, so as to minimize the measuring effort and ultimately, get the desired feasible algorithm as soon as possible. Learning that also involves additional requests for data is known as *reinforcement learning*; see, e.g., [10].

Deep learning and deep reinforcement learning. Many different machine learning techniques have been proposed. At present, the most efficient machine learning technique is a modern form of neural networks called *deep learning* (see, e.g., [4], [8]).

In general, in neural networks, neurons – elementary data processing units – are places in several consequent layers, starting with the input layer (that inputs the values $x_1, \ldots, x_n$), going through so-called *hidden layers* where the signals are processed, and ending up in the output layer that generates the desired approximation to the value $y = f(x_1, \ldots, x_n)$. The signals coming out of the neurons of each layer serve as inputs to neurons of the next layer. At first, the network is *trained*, i.e., we find the weights in the formulas describing how each neuron processes its inputs so as to get the results of applying this network to the inputs $(x_1^{(k)}, \ldots, x_n^{(k)})$ become as close as possible to the desired values $y^{(k)}$. Once the network is trained, the weights are fixed ("frozen"), and the network is ready to use.

Traditional neural networks mostly used one hidden layer. In contrast, modern neural networks use from three to sixteen and more hidden layers; because of the presence of multiple layers, they are called *deep* neural networks, and the corresponding machine learning techniques are known as *deep learning.* This technique has led to many successes [4], such as winning over human champions of Go, a complex game in which previously computer programs were not very successful.

Deep learning and deep reinforcement learning: remaining challenges. One of the main challenges with deep learning and deep reinforcement learning is that they require a large amount of computation time. In contrast to most other machine learning techniques which can be easily performed on the user's computer, deep learning and deep reinforcement learning require the use of high performance computers. This limitation severely limits the use of deep learning techniques, especially in engineering applications.

It is therefore desirable to be able to speed up deep learning and deep reinforcement learning.

Softmax selection: one of the steps that requires a large amount of computation time. The main objective of machine learning is to find a function that provides the best match for all the known records, i.e., for which, an appropriately defined distance between the observed values $y^{(k)}$ and the values $f\left(x_1^{(k)}, \ldots, x_n^{(k)}\right)$ predicted by the algorithm is the smallest possible. Since deep learning takes a lot of computation time, a natural idea to speed up the corresponding computations is to parallelize them. A natural way to parallelize the process is to run several optimization techniques in parallel for some time, and select the one that leads to the most promising result, i.e., the result for which the distance is the smallest possible.

However, in optimization, it is well known that this can lead us to a local minimum. To avoid local minima, it is necessary to not always select the best alternative, it is necessary to sometimes select other alternatives as well – so that still the best alternative is selected with the highest probability, and not so good alternatives are selected with much smaller probabilities.

This need appeared way before deep learning. For this purpose, a heuristic technique called *simulated annealing* was invented (see, e.g., [1], [7]) that often (but not always) allows is to avoid local minima. In this technique, when we want to minimize an objective function $J(a)$, we select each alternative a with the probability, e.g., proportional to $\exp(-c \cdot J(a))$ for some value c. Clearly, the smaller $J(a)$, the higher the probability of selecting the alternative a. When $c \to \infty$, the probability of selecting any other alternative except for the one which $J(a)$ attains the minimum tends to 0. From this viewpoint, the above formula can be viewed as a "soft" version of minimum, where we usually get the minimum, but sometimes we also get alternatives with a larger value of $J(a)$. Because of this, this method is also known as *softmin.*

Similarly, if we want to maximize $J(a)$, we select each alternative a with probability proportional to $\exp(c \cdot J(a))$. Here too, when $c \to \infty$, we get the alternative with the maximum value of $J(a)$ with probability 1. So, this method can be viewed as a soft version of maximum, and it is thus known as *softmax.* Softmax is exactly what is usually used in deep learning – although, to be fair, it should be noted that other probabilistic ways were also tried, and they did not seem to lead to worse results [4].

In reinforcement learning, there is an additional need for such softening. Namely, we would like to produce the values $(x_1, \ldots, x_n)$ for which estimating y and adding the resulting record will add the largest amount of information to our knowledge. In this case too, to avoid the local maximum, it is desirable to use something like softmax instead of the real maximum.

Softmax usually takes much more computation time than the usual optimization. Indeed, in optimization, if we know that for some class of alternatives, the values of the objective functions are smaller than the maximum-so-far (i.e., that the largest value that we have seen so far), then we can simply dismiss all these alternatives and concentrate on the remaining ones. In contrast, in softmax, we cannot dismiss anything, we need to take all alternatives into account and estimate their values $J(a)$ – so that we will be able to generate them with the corresponding probabilities.

This takes a lot of time. So, to speed up deep learning and reinforcement deep learning, it is desirable to speed up the corresponding computations. Let us repeat that it does not have to be necessarily exactly softmax, but we do need some way to, in general, generate alternatives with larger values of objective function with higher probabilities and the alternatives with smaller values of the objective function with lower probabilities.

II. Solution: Quantum Computing Leads to a Faster Alternative to Softmax Selection

We need a soft version of optimization. Whether we use softmax or any other alternative, what we are looking for is a "soft" version of optimization, in the following sense:

- in optimization, we select the alternative with the largest possible value of the objective function (or the smallest), while
- what we want is to select the alternative with the largest value with a high probability but also other alternatives with some probability: in general, the larger the value, the higher the probability.

In view of this fact, to come up with a quantum version of soft optimization, let us first recall how quantum computing can help with the exact optimization.

How quantum computing can help with exact optimization: a brief reminder. The main idea of using quantum computing for optimization is described, e.g., in [2]. This idea is based on one of the most well-known quantum-computing algorithms: Grover's algorithm for finding an element in an unsorted list [5], [6], [9]. In non-quantum computing, the only way to make sure that this element is found is to check all n elements of this list – and checking all these elements is sufficient. Thus, in non-quantum computing, the solution of this problem requires n computational steps.

In contrast, in quantum computing, Grover's algorithm enables us to find this element in time $O(\sqrt{n})$. To be more precise, Grover's algorithm finds the element with some probability $1-\varepsilon$. For any desired value $\varepsilon > 0$, we can find the number of iterations for which we guarantee the correct answer with the probability $\geq 1-\varepsilon$; and no matter how many iterations we select, the computation time remains $O(\sqrt{n})$. In particular, we can select so many iterations that the probability ε of the wrong answer can be smaller than the probability that the computer itself will fail – or even smaller than 1 mistake in 20 billion years. For such small $\varepsilon > 0$, we can safely ignore the possibility of a wrong answer and conclude that, from the practical viewpoint, we always get the correct answer.

Let us show how Grover's algorithm can be used for optimization. Let us assume that we have n alternatives, with the values $v_1, \ldots, v_n$ of the corresponding objective function. In mathematical terms, we want to find the index i_0 for which the value v_{i_0} is the largest, i.e., for which $v_{i_0} = \max\limits_i v_i$.

It is important to take into account that from the practical viewpoint, the values v_i are only known with some accuracy δ. Thus, it is sufficient to find the value which is δ-close to the desired maximum, i.e., for which $\left|v_{i_0} - \max\limits_j v_j\right| \leq \delta$.

Usually, we know a priori bounds for all the values of the objective function – like we usually know the lower and upper bound for each physical quantity. Thus, we know some values $m < M$ for which $m \leq v_i \leq M$ for all i. In other words, we know that all the values v_i – and, in particular, the desired optimizing value v_{i_0} – lies in the interval $[\underline{v}, \overline{v}] = [m, M]$.

This interval $[m, M]$ will be the starting interval of our iterative process, at each stage of which we will find a narrower interval $[\underline{v}, \overline{v}]$ containing v_{i_0} – until the resulting interval gets width $\leq \delta$.

The corresponding narrowing can be done as follows. At the beginning of each narrowing step, we compute a midpoint $\widetilde{v} \stackrel{\text{def}}{=} \dfrac{\underline{v} + \overline{v}}{2}$. Then, we use Grover's algorithm to check if there exists an index i for which $v_i \geq \widetilde{v}$. Depending on the result of applying this algorithm, we can make the following conclusion:

- if there exists an index i for which $v_i \geq \widetilde{v}$, this means that
$$v_{i_0} = \max_j v_j \geq v_i \geq \widetilde{v}.$$
so we can conclude that the desired largest value v_{i_0} is contained in the half-size interval $[\widetilde{v}, \overline{v}]$;
- if there is no index i for which $v_i \geq \widetilde{v}$, this means that $v_i < \widetilde{v}$ for all i and thus, $v_{i_0} = \max\limits_j v_j \leq \widetilde{v}$; in this case, we can conclude that the desired largest value v_{i_0} is contained in the half-size interval $[\underline{v}, \widetilde{v}]$.

In both cases, by spending $O(\sqrt{n})$ computational steps of Grover's algorithm, we divide the width of the interval $[\underline{v}, \overline{v}]$ containing v_{i_0} by half. In k, steps, the interval's width decreases to 2^{-k} of its original width $M - m$. Since the values v_i are known with accuracy δ, it makes no sense to locate the value v_{i_0} with higher accuracy – so we should stop when the width $2^{-k} \cdot (M - m)$ of the resulting interval $[\underline{v}, \overline{v}]$ becomes smaller than δ. At this stage, we know that $v_{i_0} \geq \underline{v}$, so we apply Grover's algorithm one more time to find the corresponding index i_0.

This computation requires $k+1$ applications of Grover's algorithm. The value k can be determined as the smallest value for which $2^{-k} \cdot (M-m) \leq \delta$, i.e., $k = \left\lceil \dfrac{M-m}{\delta} \right\rceil$. This value does not depend on n, so the overall computational complexity of this algorithm is $O(\sqrt{n})$, which, for large n, is much smaller than the fastest possible non-quantum algorithm – non-quantum computations would require that we look at every single value v_i and thus, would require $n \gg \sqrt{n}$ computational steps.

From quantum computing-based exact optimization to quantum computing-based analogue of softmax. As we have mentioned earlier, Grover's algorithm is an iterative algorithm that provides an answer with some probability $\geq 1 - \varepsilon$, where the probability of error ε depends on the number of this algorithm's iterations.

- Usually, we select this number of iterations in such a way that the probability ε is very small, to avoid deviations from the actual maximum.
- However, in our problem, we *are* in interested in deviating from the exact maximum.

Thus, a natural way to get the desired quantum version of softmax is to decrease the number of iterations – so that the probability ε becomes larger.

If we use this version of Grover's algorithm in the above optimization scheme, we get exactly what we want:

- we get the exact maximum with high probability but also
- we get other, smaller values with a reasonable non-zero probability.

Also, in contrast to softmax, that needs $O(n)$ computational steps, the proposed quantum computing-based algorithm takes much smaller time $O(\sqrt{n}) \ll n$.

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) and HRD-1242122 (Cyber-ShARE Center of Excellence).

The authors are thankful to the anonymous referees for valuable suggestions.

References

[1] E. Aarts and J. Korst, *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*, Wiley, Chichester, New York, Brisbane, Toronto, Singapore, 1997.

[2] C. Ayub, M. Ceberio, and V. Kreinovich, "How quantum computing can help with (continuous) optimization", *Proceedings of the 12th International Workshop on Constraint Programming and Decision Making CoProd'2019, Part of the World Congress of the International Fuzzy Systems Association and the Annual Conference of the North American Fuzzy Information Processing Society IFSA/NAFIPS'2019*, Lafayette, Louisiana, June 17–21, 2019, Springer Verlag.

[3] C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, New York, 2006.

[4] I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning*, MIT Press, Cambridge, Massachusetts, 2016.

[5] L. K. Grover, "A fast quantum mechanical algorithm for database search", *Proceedings of the 28th ACM Symposium on Theory of Computing*, 1996, pp. 212–219.

[6] L. K. Grover, "Quantum mechanics helps in searching for a needle in a haystack", *Physical Reviews Letters*, 1997, Vol. 79, No. 2, pp. 325–328.

[7] V. Kreinovich, "Group-theoretic approach to intractable problems," *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1990, Vol. 417, pp. 112–121.

[8] V. Kreinovich, "From traditional neural networks to deep learning: towards mathematical foundations of empirical successes", In: S. N. Shahbazova et al. (eds.), *Proceedings of the World Conference on Soft Computing*, Baku, Azerbaijan, May 29–31, 2018.

[9] M. Nielsen and I. Chuang, *Quantum Computation and Quantum Information*, Cambridge University Press, Cambridge, 2000.

[10] R. S. Sutton and A. G. Barto, *Reinforcement Learning. An Introduction*, 2nd edition, MIT Press, Cambridge, Massachusetts, 2018.