Naturalizing the Neural Network

Akash Jagannathan
San José State University
013856531
San José, CA, USA
akash.jagannathan@sjsu.edu

***Abstract*—Previous research has shown that esoteric algorithms for basic arithmetic, known as Vedic mathematics, improved computation speed and memory usage in a hardware implementation of an artificial neural network (ANN). This study extends this research by testing computation speeds and memory usage of a software implementation of an ANN using Vedic mathematics. Although implementation was limited in scope, results show increased execution speed (in certain cases) for a gradient descent algorithm utilizing Vedic addition compared to a similar algorithm using default addition techniques.**

Keywords—Artificial Neural Network, Vedic Mathematics

I. Introduction

Scientists are still left with many questions when it comes to the intricacies of brain function. While there are many mysteries outstanding, one benchmark that is generally agreed upon in the neuroscientific community is that the brain is one of the most complex biological organisms known to man. Besides having 100 billion basic processing units known as neurons [1], it is teeming with myriad connections and a complex systems hierarchy that is thought to give rise to the complex thinking abilities of man. Everything from memory, attention, speech, and even relatively abstract phenomena such as hallucinations and consciousness have been hypothesized to arise from this proverbial "black box" in science. Leaving the unknowns aside, for the purpose of this paper it is enough to reason that when thinking about how to build a robust information processing system, one need not venture any further into nature to find an appropriate example of such a system than what is in one's own head.

Alongside the increased fascination with how the brain works, in the last decades science has made equally prolific advancement in building artificial yet robust informational processing systems, known as computer systems. The devices many of us hold in our pockets these days boast as much computational power as computer scientists a few decades ago had in their primary research centers! A natural trend in this line of research has been building autonomous systems; systems capable of processing information and acting on it completely on their own. Therefore, it is quite natural that computer scientists have modeled the brain in developing these systems. One offshoot of this direction of research is known as the Artificial Neural Network. The term neural network may mislead one to believe that it is an exact replica of the brain. However, as cleverly put in [2], similar to how we model planes after birds, we model neural networks to the brain.

The fundamental building block of an ANN is a perceptron, similar to how neurons are the fundamental units in the brain. ANN's are built of an arbitrary number of layers of these perceptrons. The first layer is the input layer, and the final layer is known as the output layer. Data is fed in through the input layer and percolates through the layers before reaching the output layer. Each perceptron aggregates the inputs of the previous layers combined with weights on each input which mimic synaptic connections between neurons in the brain.

Earlier, it was mentioned that there is much still unknown in neuroscience, and one of these unknowns is about how the brain actually passes signals between neurons, i.e. the specific computations it does. Typical implementations of ANN's have assumed conventional addition multiplication processes. However, the researchers in [3] experimented with another way of doing basic arithmetic, specifically multiplication, which was sourced from text published in the early 1900s in India [4] but was likely derived much earlier in time. This technique of multiplication is known as "Urdhva Tiryagbhyam", and it has subtle differences to the more conventional method taught in Western schools. Fig. 1 gives one an idea of how this works for 2, 3, and 4 digits. The carrying of tens digits from one column to the next is similar to conventional multiplication. However, one notices a more crosswise technique when aggregating the digits between columns. This is what is thought to be easier for the brain to do in terms of computational intensity and time taken.

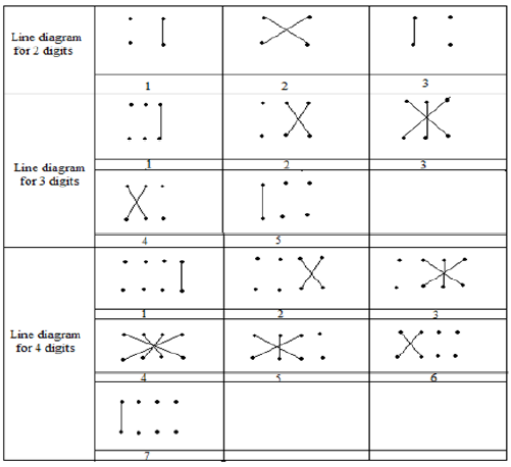

Fig 1. Diagrammatic algorithm of Vedic multiplication for 2, 3, and 4 digits

The "Urdhva Tiryagbhyam" algorithm can similarly be used for bitwise multiplication. The authors in [3] implemented this technique of multiplication within a single hardware implementation of a perceptron. In hardware implementations, a component known as a "multiplier" is responsible for determining the signal (the sum of the products of each input of the previous layer with its corresponding weight, also known as $s^{(l)}$. They replaced a conventional multiplier with a multiplier using the Vedic technique described above. The results they obtained are shown in Fig 2.

FPGA device package : xc2s15-6-cs144	Delay (ns)	Memory (Kb)
Using standard multiplier	32.969	72772
Using Vedic multiplier	25.675	73668

Fig 2. Results from hardware implementation of Vedic multiplier in ANN

Here, delay refers to the "delay/speed" of the ANN. One can see that the perceptron utilizing the Vedic multiplier is faster than the perceptron using the standard multiplier. Other results mentioned in the paper reveal that the perceptron using the Vedic multiplier uses markedly less logic than the perceptron using a standard multiplier.

At the end of their paper, the authors in [3] mentioned their results could be extended by implementing the Vedic addition method instead of standard addition. In this study, the attempt has been made for a software implementation of Vedic addition on an Artificial Neural Network.

II. Software Implementations of Vedic Mathematics on an ANN

A. Vedic Addition

The author in [5] mentions three main sutras used for addition in the Vedas: 1) Ekadhikena Purvena Sutra, 2) Vilokanam Sutra, and 3) Nikhilam Sutra. Of particular interest in relation to ease of computation is the Nikhilam Sutra. This method works by a) finding a base number nearest to each addend, b) finding the difference/sum between the respective bases and addends, c) adding the bases, and d) subtracting/adding the difference/sum found in b) with the sum found in c). When reflecting on how one adds very large complex numbers, some may realize that they unknowingly use this method!

The ease of this method can be illustrated with a simple example. If one tries to add the numbers 87 and 95 using conventional methods, the main computational issue that arises is remembering all of the different parts one is adding and carrying over. However, using the Nikhilam Sutra, this becomes less of a pain to the brain. One can choose 100 as the base number for both addends. After completing step b) above, one finds the overall difference to be -18. We then add the base numbers, which in the case of 100+100, is a very trivial mental operation. Finally we subtract the difference from the sum in c), which gives us 200-18 or 182.

Using this method, one can see that there are much less parts to keep in memory while one proceeds through the steps. This relaxation of memory constraints seems to be a hallmark feature of the Vedic Math Sutras. It is interesting that the researchers in [3] found that a computer might react the same way.

B. Software Implementation

Full source code available at:
https://github.com/ajag408/VedicDigits

The neural network used in this experiment was implemented using the Python programming language and the Jupyter notebook environment. A Python library known as 'vedic-py' was imported in order to do the Vedic additions [6]. The main developer of this library notes that it is incomplete.

The neural network tested in this study had two layers with one hidden layer consisting of 10 perceptrons. Implementation included separate methods for forward propagation for calculating signals and outputs of the network, back propagation for calculating sensitivities in the network, and a gradient descent method which looks to minimize the error by adjusting the weights on successive iterations.

The sample data that was fed through the neural network was the set of handwritten digits known as the MNIST Database [7]. Each handwritten digit is represented by 256 grayscale values. Feature reduction was accomplished by converting the representation of each image to contain two attributes: symmetry and intensity.

Initially, attempts were made to convert the entire neural network to one performing only Vedic mathematical techniques. However, due to current limitations of the vedic-py library, this was not possible. This library specifies an object type known as a VedicNumber, which is what is needed to be fed into the methods to perform the calculations. The constructor method requires a positive integer to be passed in for conversion to this data type; negative integers or floats are not compatible. This posed a substantial problem to this study, as many internal calculations in the methods that comprise the neural network consist of decimal numbers which are often negative. Furthermore, once converting to the VedicNumber type, it is not possible to convert it back to an integer. This made it impossible to aggregate results over multiple iterations, as is needed in the gradient descent method to update weights.

Due to these limitations, the only place possible to implement the vedic-py library and subsequent techniques in the neural network was in very limited areas of the gradient descent method. Specifically, the gradient was initialized to be a VedicNumber list of lists corresponding to the dimensions of the weight matrices in each layer of the neural network. Vedic addition was implemented when calculating $G^{(l)}$ for layer two of the network for the first data point propagated through the network. To mirror the changes made to the initial gradient descent implementation to incorporate Vedic addition, a "placebo" gradient descent method was created. This method exactly mimicked the Vedic version but did not use any Vedic mathematics.

Speed of operation and memory being used by these different versions was captured using the IPython magic commands %time (times execution of a single statement), %timeit (times repeated execution of single statement), and %memit (measures the memory use of a single statement). The statement being measured was one epoch of gradient descent, which includes forward propagation and back propagation.

TABLE I. SPEEDS AND MEMORY USE OF GRADIENT DESCENT IMPLEMENTATIONS

Measure	Gradient Descent Implementations		
	Normal	***Vedic***	***Placebo***
Single Run	113 ms	91.5 ms	98.9 ms
Multiple Runs	89.1 ms	77.2 ms	69.4 ms
Peak Memory	192.31 MiB	192.32 MiB	192.32 MiB

[a.] Times are total CPU times (user + sys). Multiple runs consist of 2 x 5 iterations

[b.]

C. Results

Table 1 shows the results obtained in this experiment. The Vedic implementation seems faster in terms of total CPU time taken to complete one epoch iteration of gradient descent than both the normal and placebo implementations. In total, Vedic addition was used one time for the single run, 10 times for multiple runs, and one time for the memory experiment.

D. Analysis of Results

Although infrequently used, the use of Vedic addition for this neural network seems to speed up the overall processing time of the algorithm. The placebo seems to perform better on multiple runs, however the running times shown here have been found to vary. Memory constraints on the system seem to not change from one implementation to another. These results indicate that even in the software realm, the use of Vedic mathematics, even sparsely, positively affects the running time of a neural network.

III. LIMITATIONS OF PRESENT STUDY AND CONCLUSION

The lack of thoroughness in the vedic-py library placed severe constraints on the ability for software implementation of Vedic mathematical operations in this neural network. For reasons stated above, this implementation would not return practical results if used in a real-world setting. For this to be possible, some adjustments have to be made to the library.

The authors in [3] implemented Vedic multiplication. Incorporating that technique here might artificially increase the running time of the network. This is because certain places where multiplication is being used in gradient descent utilize specialized NumPy array operations such as computing the dot product. Expanding this into a series of multiplications might artificially increase the running time.

Future studies could look at modifying the vedic-py library to be more compatible with neural networks and applying Vedic mathematics in other realms of machine learning.

Based on the results found in [3] and this study, one obvious question that remains unanswered is exactly how using Vedic mathematics results in faster computation speeds in both hardware and software implementations. The authors in [3] just briefly mention that it is thought the brain performs better using a Vedic style of computation as this might be its natural way of computing. While the brain is difficult to dissect completely, a computer is man-made, and thus it might be more feasible to examine the exact mechanics of this phenomenon. Besides benefiting the technical fields such as computer science, it could have a backpropagating effect in helping neuroscientists understand brain mechanics better.

Although ancient, these studies show that Vedic mathematics is certainly not outdated. Indeed, it might have found an excellent use in the realm of artificial intelligence, machine learning, neural networks, and computations in general. With the rise in technological complexity, speed becomes an important factor for designing both hardware and software. Using ancient yet simple ways of arranging numbers might be the balancing factor and ultimately the solution to this pattern. Maybe our natural neural networks and their artificial counterparts in computers are not so different after all!

REFERENCES

[1] S. Herculano-Houzel, "The remarkable, yet not extraordinary, human brain as a scaled-up primate brain and its associated cost," *PNAS*, 26-Jun-2012. [Online]. Available: https://www.pnas.org/content/109/Supplement_1/10661. [Accessed: 10-Dec-2020].

[2] Y. S. Abu-Mostafa, L. Hsuan-Tien, and M. Magdon-Ismail, *Learning from data: a short course*. AMLbook, 2012.

[3] Y. SV, Anshika, N. Goel, and I. S, "High speed neuron implementation using Vedic mathematics," *Discovery*, pp. 25–32, Oct. 2015.

[4] B. K. Tirtha and V. S. Agrawala, Vedic mathematics: or Sixteen simple mathematical formulae from the Vedas. Motilal Banarsidass, 1992.

[5] *Vedic Mathematics Sutra for Addition*, 05-Apr-2019. [Online]. Available: https://navavedic.blogspot.com/2019/04/vedic-mathematics-sutra-for-addition.html. [Accessed: 10-Dec-2020].

[6] Techmoksha, "techmoksha/vedic-py," *GitHub*. [Online]. Available: https://github.com/techmoksha/vedic-py. [Accessed: 10-Dec-2020].

[7] "THE MNIST DATABASE," *MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges*. [Online]. Available: http://yann.lecun.com/exdb/mnist/. [Accessed: 10-Dec-2020].

[8] G. Guzun, "Classification & Regression." Gheorghi Guzun, San Jose.

[9] "memory-profiler," *PyPI*. [Online]. Available: https://pypi.org/project/memory-profiler/. [Accessed: 10-Dec-2020].

[10] "4.4. Profiling the memory usage of your code with memory_profiler," *IPython Cookbook, Second Edition*. [Online]. Available: https://ipython-books.github.io/44-profiling-the-memory-usage-of-your-code-with-memory_profiler/. [Accessed: 10-Dec-2020].

[11] "Python: math.floor() function," *GeeksforGeeks*, 11-Mar-2019. [Online]. Available: https://www.geeksforgeeks.org/python-math-floor-function/. [Accessed: 10-Dec-2020].