A GPU Implementation of a Bat Algorithm Trained Neural Network

Amit Roy Choudhury(✉), Rishabh Jain, and Kapil Sharma

Department of Computer Engineering, Delhi Technological University, New Delhi, India
amitrc17@gmail.com

Abstract. In recent times, there has been an exponential growth in the viability of Neural Networks (NN) as a Machine Learning tool. Most standard training algorithms for NNs, like gradient descent and its variants fall prey to local optima. Metaheuristics have been found to be a viable alternative to traditional training methods. Among these metaheuristics the Bat Algorithm (BA), has been shown to be superior. Even though BA promises better results, yet being a population based metaheuristic, it forces us to involve many Neural Networks and evaluate them on nearly every iteration. This makes the already computationally expensive task of training a NN even more so. To overcome this problem, we exploit the inherent concurrent characteristics of both NNs as well as BA to design a framework which utilizes the massively parallel architecture of Graphics Processing Units (GPUs). Our framework is able to offer speed-ups of upto 47× depending on the architecture of the NN.

Keywords: Neural Networks · Bat Algorithm · GPU

1 Introduction

Neural Networks have received immense attention from the international research community owing to their universality and adaptability. There has been a growth in the use of Artificial Neural Networks (ANN) for real world applications [1] in recent years, mostly because of availability of powerful hardware. ANNs find their use in both classification and regression tasks [2] but in this paper we focus on the classification application of feed-forward Neural Networks. These NNs consist of an input layer, an output layer and one or more hidden layers in between them composed of nodes called neurons. The number of nodes in the input layer is dictated by the number of dimensions of input. Similarly, the number of nodes in the output layer is equal to the number of output classes. The number of nodes in any hidden layer is a parameter of the system and is set empirically. Each node is connected to all nodes of the next layer. Each connection is associated with a weight. The value associated to any neuron, called it's activation value, can be calculated as follows

$$V = g(\sum_{i=0}^{n} I_i.W_i) \tag{1}$$

A. Hirose et al. (Eds.): ICONIP 2016, Part III, LNCS 9949, pp. 326–334, 2016.
DOI: 10.1007/978-3-319-46675-0_36

where W_i is the weight associated with a connection from input i to current node, I_i is input value i and $g(x)$ is called the activation function. The activation function is generally taken to be a non-linear function such as logistic function, hyperbolic tangent or rectified linear units.

Training of neural networks has been a widely researched topic [3,4]. The most popular methods of training are gradient based algorithms like Gradient Descent or Conjugate Gradient. Though these are widely used, they are known to fall prey to local minima [5]. Other Metaheuristics have shown comparable or better results in comparison to these methods [4,6]. Among them, the Bat Algorithm (BA) has been shown to be superior [7].

BA is a population based optimization technique. Such techniques are used for optimization problems i.e. problems in which the goal is to find the minimum (or maximum) of an objective function. BA is inspired by the hunting behaviour of bats [8]. It uses a process similar to the echolocation used by bats to identify and track down its prey. We consider a population of N bats. The i^{th} of which is flying with a velocity of V_i^t and is at a position X_i^t at time t. The bats vary their frequency f (and thus wavelength) and loudness A to locate prey. They alter their frequencies and velocities as follows

$$f = f_{min} + (f_{max} - f_{min}) * \delta \tag{2}$$

$$V_i^t = V_i^{t-1} + (X_i^t - X_{gBest}^t) * f_i \tag{3}$$

where X_{gBest} (global best) is the position of the bat which is closest to the prey, f_{max} and f_{min} are maximum and minimum frequencies respectively which are parameters of the system that are decided empirically and $\delta \in [0,1]$ is a randomly generated number. The best bat is identified by the value of the fitness function at its position. This fitness function is the function which is being optimized. The position of a bat varies with time given by

$$X_i^t = X_i^{t-1} + V_i^t \tag{4}$$

Any bat may perform a local search instead of moving along its velocity according to its pulse rate. For the local search procedure each bat takes a random walk creating a new position (solution) for itself, this is given by

$$X_{new} = X_{old} + \alpha.A_{avg} \tag{5}$$

where A_{avg} is the average loudness of the population and $\alpha \in [-1,1]$ is a random number. The loudness of a bat decreases and the pulse rate increases as the bat gets closer to its prey.

$$A^{t+1} = \alpha.A^t \tag{6}$$

$$r^{t+1} = r_0.(1 - e^{-\gamma.t}) \tag{7}$$

where alpha and gamma are constants which are decided empirically and r_0 is the final pulse rate generally equal to 1.

BA (or any other population based metaheuristic) requires many evaluations of the fitness function. When applying BA for the training of NNs we must perform evaluations of NNs multiple times in an iteration. This is a very computationally expensive task and might not be feasible for large networks. Thus we propose a parallel approach to this training mechanism, by performing the entire process on a Graphical Processing Unit (GPU). We use the Compute Unified Device Architecture (CUDA) [9] which is NVIDIA's proprietary framework for general purpose GPU computing. It is based on the Single Instruction Multiple Thread (SIMT) model of parallelism. The same instruction is run simultaneously on all threads of execution. We use the CUDA C/C++ for programming which is entirely like C/C++ except for the fact that we can distinguish which functions to run on the GPU(device) and which to run on the CPU(host). Functions that run on the GPU are called kernels. We limit our discussion on CUDA here, further details are available in [9].

In this paper we provide a mapping of BA to the training procedure of an NN and we go on to propose an approach that can be used to parallelize the process. We discuss this approach in detail and also carry out extensive testing on various standard as well as non - standard datasets. We are able to achieve a maximum speedup of 47× on dense networks trained using large bat populations.

The rest of the paper is organized as follows. In Sect. 2 we discuss the implementation details of our method. In Sect. 3 we analyze our framework and test it on various datasets which are themselves described as well. Then, in Sect. 4 we conclude and mention some future work that may emerge.

2 Implementation

Each bat is represented as a complete Neural Net. To minimize memory transfers between the CPU and GPU, the entire population of bats (the neural networks) is kept on the GPU Global memory. The fitness function for BA used was the accuracy of the neural net i.e. the number of correctly classified instances. The position of a bat is considered to be defined by each weight of each weight matrix of the neural net. Hence the number of dimensions of the position (and velocity) of a bat is given by

$$\sum_{i=0}^{n-1} L_i.L_{i+1} \tag{8}$$

where L_i is the number of neurons in the i^{th} layer of the NN, with L_0 being the size of input layer, L_n being size of output layer and n the number of layers. We now discuss our approach to designing a parallel framework for the training of NNs.

In our approach we parallelize both BA and the NN so that both are able to utilize the GPU to its maximum potential. This can be done using the fact that, each bat is independent of the others. This means each neural network can be evaluated in parallel. We exploit this fact by giving the responsibility of a neural network to a block of GPU threads, so that each block utilizes its threads to forward propagate the input.

The algorithm is broken down into several kernel calls. First we use a kernel to update all the bats' positions according to their velocities and their velocities as

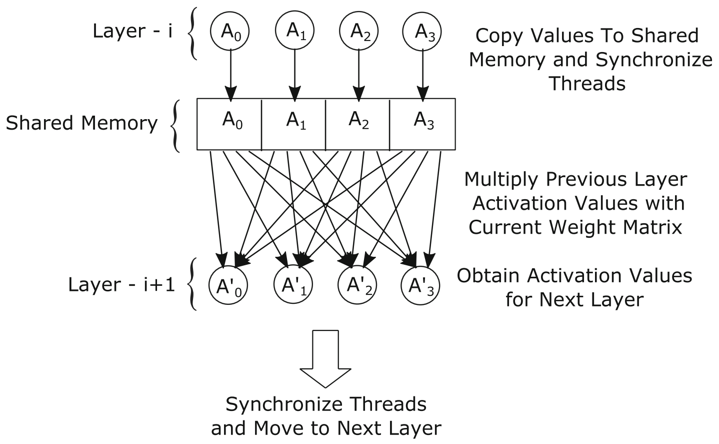
Layer - i
A_0
A_1
A_2
A_3
Copy Values To Shared Memory and Synchronize Threads
Shared Memory
A_0
A_1
A_2
A_3
Multiply Previous Layer Activation Values with Current Weight Matrix
Layer - i+1
A'_0
A'_1
A'_2
A'_3
Obtain Activation Values for Next Layer
Synchronize Threads and Move to Next Layer

Fig. 1. Procedure for forward-propagate kernel. Synchronization is done twice per layer, once after copying data to shared memory and again after computing activation values.

well using Eqs. (2)–(4). Only a single block is launched for this kernel containing as many threads as there are bats.

Next we need to calculate the fitness values for the bats at their new positions. For this, we use 2 kernels for each input. The first one aligns the current input with all of the NNs as shown in Fig. 2. The i^{th} block copies the i^{th} dimension of input from its shared memory to the input layer of the NNs. Within each block the j^{th} thread copies to the j^{th} NN. So this kernel will launch as many blocks as there are dimensions of input and the number of threads in each block will be equal to N (population size).

The next kernel is used to forward-propagate the input through the NNs. As each NN is independent of any other so each block of this kernel is assigned to each NN. Within each block work done is as shown in Fig. 1. A single thread performs all computations required to calculate the activation value of a single neuron of the next layer according to Eq. (1). All threads within a block are synchronized before continuing on to propagate to the next layer. This ensures that previous values have been assigned before they are used. The activation values of the previous layer are kept in shared memory as they will need to be repeatedly accessed and shared memory has much faster access times in comparison to global memory [9].

Now we check the condition required for random walk of each bat using another kernel. As we will need to re-evaluate those bats which will perform the random walk but not the others, we take a boolean vector which indicates which bats are to be re-evaluated and set this vector using the kernel in parallel. This vector is kept in global memory for access by the forward-propagating kernel which is then called again to re-evaluate the bats that have moved.

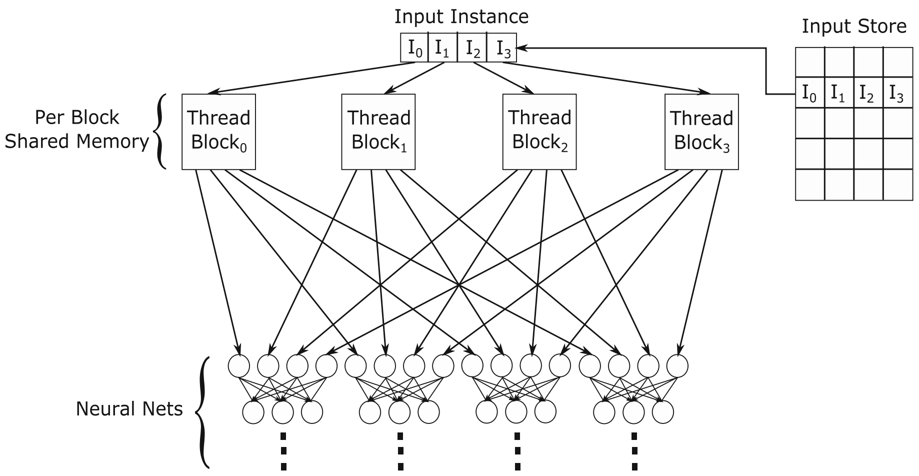

Fig. 2. Alignment of input to all NNs for a simple population of 4 NNs and an input with 4 attributes.

Next we check all bats to see which ones have moved to better positions and update their personal best positions. Again only a single block of threads is launched containing N threads. We also adjust their loudness and pulse rates according to Eqs. (6) and (7).

Finally we obtain the new best bat according to their fitness values. This can be done in parallel using a technique similar to parallel reduction [11], or for small populations we can even transfer the fitness values to host memory and find the best one without significant increase in execution time.

This implementation is designed so that once the entire input and all the networks are moved to device memory, provided they can fit, we do not need any further (significant) memory transfers between host and device for the entire training process. This helps avoid bottlenecks due to memory transfers, and keeps the GPU busy all the time.

3 Results

For all the tests that were conducted, we were able to produce an ideal environment where we needed minimum memory transfers between the GPU and the CPU. Before starting the learning process, we copied all the input as well as initial bats' parameters on to the GPU global memory. We have restricted our work to classification tasks. We have used a mix of easy and difficult to classify datasets. Apart from the Malware dataset, all others are well known benchmark datasets available publicly. We implemented both CPU and GPU versions of BA trained NNs. The tests were run on an Intel Core i7 3610 QM (2.3 GHz) CPU, 6 GB memory and an NVIDIA GT 650M GPU (835 MHz core clock) with 1 GB global memory and CUDA 7.0 toolkit with compute capability 3.0.

Table 1. Accuracies for BA trained NN. Trained to 200 epochs.

Dataset	Instances	Accuracy
Breast Cancer	569	98.24
Wine	178	100.00
Iris	150	98.00
EEG eye state	14980	56.72
Malware	4000	87.90

We briefly describe the datasets used for testing (Table 1). All datasets were obtained from the UCI machine learning repository [10] unless stated otherwise.

Wine Dataset. It consists of 178 instances belonging to 3 classes and having 13 attributes each. It can be learned even by a simple network easily.

Iris Dataset. It consists of 150 instances belonging to 3 classes. It is a very basic dataset which can be learned (at times even 100 %) within a few epochs.

Wisconsin Breast Cancer Diagnostic Dataset. It consists of 569 instances belonging to 2 classes malignant or benign. Each instance has 30 attributes. This is more difficult to learn when compared to the previous two datasets.

EEG Eye State Dataset. It consists of 14980 instances belonging to 2 classes either open or closed eye state. Each instance has 14 attributes. This dataset is much harder to learn then the previous ones.

Malware Dataset. We compiled this dataset ourselves. We obtained malware files from VX heaven[1] and assembled clean files from Windows operating system directories. We filtered the files for size so that the sizes of both clean and malware files were around 10 KB. We then extract features in the form of 16 bit words from the binary files by grouping each block of 16 bits together. We used tf-idf vectors to quantify the features of each file. As there can be upto 2^{16} features, we try and reduce this by using F-scores of the features and then selecting the top 256 features. There were a total of 4000 files with 3000 clean and 1000 malware. This dataset is difficult to classify and to put things in perspective we also learn it using Gradient-Descent (Back-Propagation) and compare accuracies.

The speedups obtained can be somewhat co-related to the size of the network. Larger networks exhibit higher speedups. We can quantify the size of a network by considering the total number of weight elements in the network, which will be directly related to the number of calculations to be made to evaluate the entire population. We display this quantity in Table 2 under the *parameters* column.

For the Wine dataset we use a network with 1 hidden layer with 1024 hidden units and a population size of 128. For the Iris dataset we use network with 2 hidden layers with 256 units each and a population size of 128. We are able to get a speedup of 36× over a serial implementation on this dataset. For the Wisconsin

[1] http://vxheaven.org/.

Table 2. Speedups for BA trained NN. The time is average time for one epoch

Dataset	Network	Parameters	Time (CPU)	Time (GPU)	Speedup
Breast Cancer	30-256-2, 128 bats	1048576	84.72 s	5.06 s	16.74×
Wine	13-1024-3, 128 bats	2097152	61.17 s	4.78 s	12.79×
Iris	4-256-256-3, 128 bats	8617984	177.56 s	4.82 s	36.83×
EEG eye	14-256-2, 64 bats	262144	287.32 s	64.79 s	4.43×
Malware	256-256-2, 64 bats	4227072	1379 s	77.1 s	17.88×
Malware	256-640-2, 256 bats	42270720	8881 s	186 s	47.74×

Breast Cancer Dataset we use a network with a single hidden layer of 256 units and a population of 128 bats (Fig. 4). This gives us a speedup similar to that of the Wine dataset as the sizes of both networks are comparable. The EEG eye dataset uses the smallest network in terms of number of parameters and exhibits the smallest speedup. Though the number of parameters do not always dictate the speedups, as can be seen from the difference in speedups between the Wine and Wisconsin Breast Cancer datasets, yet they are a good indicator most of the time.

For the Malware dataset we use 2 different networks. The first one is a large network with a single hidden layer containing 640 units and a population of 256 bats while the second is a smaller network with 256 hidden units and 64 bats. As expected, the larger network obtains higher speedup than the smaller one, with a speedup of 47× over the serial implementation. In this case training with the serial implementation is unfeasible as a single epoch takes around 150 min.

As the Malware dataset has not been studied before we compare the accuracies obtained for our BA trained NNs with that of one trained with gradient descent. Although, we must stress that our work is focused on speeding up the training process rather than obtaining state-of-the-art accuracies, so we refrain from using advanced feature extractors and other methods as in [12] to obtain higher accuracies. The comparison from Fig. 3 reveals that BA outperforms Gradient Descent in this case.

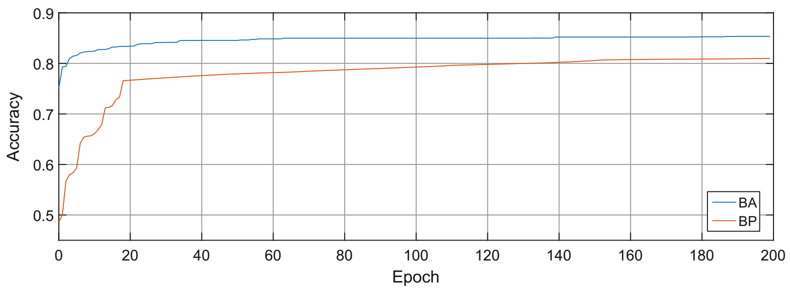

Fig. 3. BAT vs BP on the Malware dataset (256-256-2 network and 64 bats). Average of 10 runs

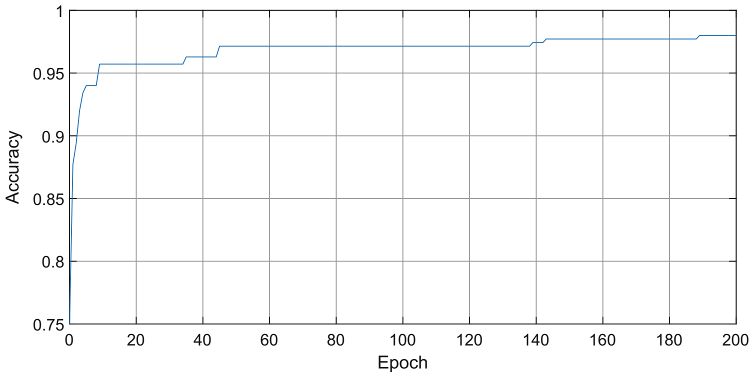

Fig. 4. Accuracy for BA trained NN on Wisconsin Breast Cancer dataset

4 Conclusion and Future Work

The Bat algorithm puts forward a good alternative to traditional gradient based algorithms for Neural Network training. In this paper we make the use of BA in this context much more feasible than before by running the entire process on GPUs. We propose a framework to do the same and test it on various datasets with varied number of instances as well as different difficulties of classification. Our framework is able to minimize the communication between the CPU and GPU to a bare minimum, where communication is needed only to either transfer results at the end or for transferring data before starting. We have also observed that the speedups obtained for various networks are loosely related to the size of those networks.

Further one may explore the possibilities of applying dynamic parallelism to BA as it allows launching of new threads from the GPU itself. This allows us to combine the entire training process in a single kernel launch from the host. A comparative study of parallel BA and other parallel metaheuristics in the context of NN training is needed. Also methods to choose hyper-parameters for metaheuristic trained NNs is required.

References

1. Widrow, B., Rumelhart, D.E., Lehr, M.A.: Neural networks: applications in industry, business and science. Commun. ACM **37**(3), 93–106 (1994)
2. Specht, D.F.: A general regression neural network. IEEE Trans. Neural Netw. **2**(6), 568–576 (1991)
3. Hagan, M.T., Menhaj, M.B.: Training feedforward networks with the Marquardt algorithm. IEEE Trans. Neural Netw. **5**(6), 989–993 (1994)
4. Montana, D.J., Davis, L.: Training feedforward neural networks using genetic algorithms. In: IJCAI, vol. 89, pp. 762–767, August 1989
5. Gupta, J.N., Sexton, R.S.: Comparing backpropagation with a genetic algorithm for neural network training. Omega **27**(6), 679–684 (1999)

6. Gudise, V.G., Venayagamoorthy, G.K.: Comparison of particle swarm optimization and backpropagation as training algorithms for neural networks. In: Proceedings of the 2003 IEEE Swarm Intelligence Symposium, 2003. SIS 3003, pp. 110–117. IEEE, April 2003
7. Khan, K., Sahai, A.: A comparison of BA, GA, PSO, BP and LM for training feed forward neural networks in e-learning context. Int. J. Intell. Syst. Appl. **4**(7), 23 (2012)
8. Yang, X.S.: A new metaheuristic bat-inspired algorithm. In: González, J.R., Pelta, D.A., Cruz, C., Terrazas, G., Krasnogor, N. (eds.) NICSO 2010. Studies in Computational Intelligence, vol. 284, pp. 64–74. Springer, Heidelberg (2010)
9. Nvidia, C.U.D.A.: C programming guide version 4.0. NVIDIA Corporation, Santa Clara (2011)
10. Lichman, M.: UCI machine learning repository. University of California, School of Information and Computer Science, Irvine (2013). http://archive.ics.uci.edu/ml
11. Harris, M.: Optimizing parallel reduction in CUDA. NVIDIA DeveloperTechnology **2**(4), (2007). http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf
12. Rieck, K., Holz, T., Willems, C., Düssel, P., Laskov, P.: Learning and classification of malware behavior. In: Zamboni, D. (ed.) DIMVA 2008. LNCS, vol. 5137, pp. 108–125. Springer, Heidelberg (2008)