CUDA-accelerated genetic feedforward-ANN training for data mining

Catalin Patulea, Robert Peace and James Green

School of Systems and Computer Engineering, Carleton University, Ottawa, Canada K1S 5B6

E-mail: cpatulea@sce.carleton.ca, rpeace@sce.carleton.ca, jrgreen@sce.carleton.ca

Abstract. We present an implementation of genetic algorithm (GA) training of feedforward artificial neural networks (ANNs) targeting commodity graphics cards (GPUs). By carefully mapping the problem onto the unique GPU architecture, we achieve order-of-magnitude speedup over a conventional CPU implementation. Furthermore, we show that the speedup is consistent across a wide range of data set sizes, making this implementation ideal for large data sets. This performance boost enables the genetic algorithm to search a larger subset of the solution space, which results in more accurate pattern classification. Finally, we demonstrate this method in the context of the 2009 UC San Diego Data Mining Contest, achieving a world-class lift on a data set of 94682 e-commerce transactions.

1. Introduction

Genetic algorithms (GAs) are a stochastic, evolutionary approach to machine learning for pattern classification. While greedy methods will get stuck at local extrema, GAs are theoretically capable of asymptotically reaching the global optimum [1]. However, because they require the evaluation of approximately 50 to 100 classifiers at each iteration, and the algorithm is repeated for several thousand iterations, they are extremely compute-intensive. The use of artificial neural networks (ANNs) as the classifier exacerbates the computational requirements because ANNs themselves are very compute-intensive.

Compared to a traditional sequential implementation of GA training of ANNs, the use of graphical processing units (GPUs) as parallel processors provides significant performance improvements at a fraction of the price of alternatives such as cluster computing. GPUs, however, use a specialized programming paradigm which must be taken into account to leverage their full processing power. In this report, we show an implementation of GA training of ANNs which achieves an order-of-magnitude speedup over a sequential algorithm.

2. Background

2.1. Artificial Neural Networks

ANNs are composed of nodes connected by weighted directed edges. In a feed-forward network architecture, nodes are organized into an input layer, one or more hidden layers, and an output layer. Each layer is fully connected to the next. Figure 1 demonstrates

the feedfoward ANN structure. Each of the hidden nodes at hidden layer n perform a transcendental function with inputs equal to the outputs of layer $n-1$ each multiplied by the weight associated with the edge through which they are fed from layer $n-1$ to layer n. In a radial basis function (RBF) network, the nodes in each hidden layer calculate the distance between their inputs and a centre vector, and pass a weighted distance to a RBF. Nodes in a sigmoid function network perform a similar operation with a sigmoid function in place of the RBF. The output node is typically linear, computing a weighted sum of its inputs. The output of the ANN is a single real value which is interpreted as a classification confidence. ANN computation is dominated by the number of multiplications at the directed edges, proportional to the number of features in the data set and the number of hidden nodes, and by the transcendental functions at the hidden nodes, which are typically exponentiations. ANN training has been implemented previously on GPU hardware, however these efforts have focused on training through backpropagation [2] as opposed to genetic algorithms.

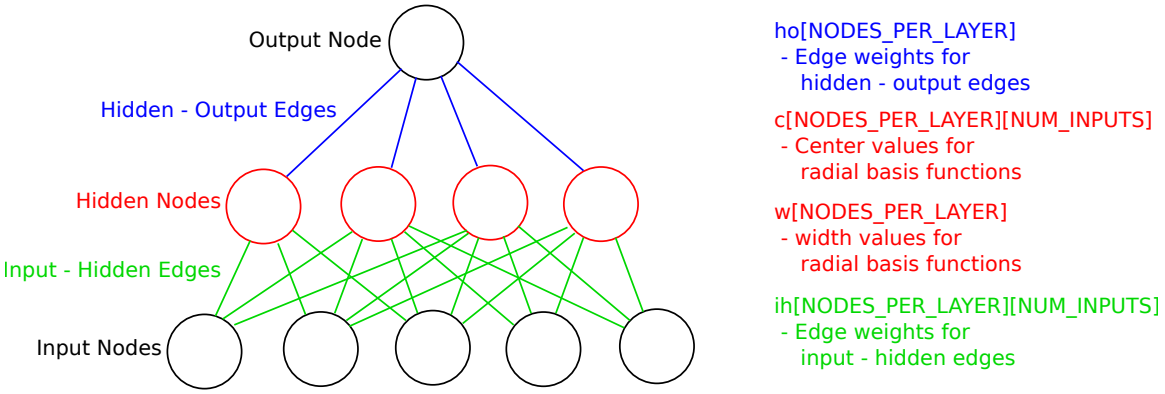

Figure 1. ANN structure and related chromosome design. Each element in the chromosome is a real-valued number.

2.2. Genetic Algorithms as ANN Training Agents

Genetic algorithms (GAs) use populations of candidate solutions to simultaneously explore the search space of ANN parameters [1]. The candidate ANNs are randomly mutated, mated and selected during each of several generations. Mutation and mating is done using a problem-specific representation of candidate solutions and "genetic operators." Selection requires calculating the fitness of each candidate ANN then preferentially selecting candidate ANNs based on fitness. The fitness of a classifier is a classifier accuracy metric such as sensitivity. The competitive bias imposed by selection attempts to mimic the "survival of the fittest" principle often seen in nature. Because GAs perform an unbiased search of a solution space and ANNs can be of arbitrary complexity, this technique is suitable for data mining applications using large or complex data sets.

The computation required for GA training of ANNs is proportional to the number of generations (10^1-10^4), to the size of each generation (10^1-10^2 candidates) and to the size of the data set (10^2-10^6 instances).

2.3. The CUDA Platform

NVidia's Compute Unified Device Architecture (CUDA) is a programming platform for massively parallel GPUs found in off-the-shelf graphics cards. GPUs consist of several dozen

independent floating-point units, providing significant speedup to data-parallel compute-intensive applications [3]. Each unit is connected to on-board memory using a very wide bus, enabling high memory bandwidth provided certain memory access rules are respected. These features make CUDA an ideal platform for GA-ANN training.

2.4. Sample Application

Our demonstration classifier is designed for the 2009 UC San Diego Data Mining Contest "E-commerce Transaction Anomaly Classification" [4] and with training data thereof. The training data consist of 94682 instances with 19 features of mixed types and a 1:50 binary class imbalance. The evaluation metric is "lift at 20%", which can be understood as the ratio of the true positive rate in the top 20% ranked instances to the overall positive rate of the data set. Lift at 20% is commonly used in marketing research in order to select the subset of a population which is most likely to respond to solicitation.

3. Classifier Design

3.1. ANN Structure Design

While our GA-ANN classifier was designed for the 2009 UC San Diego Data Mining Contest data set, it has been designed in order to be capable of adapting to a variety of data sets. We have developed a flexible ANN topology, allowing for either one or two layers of hidden nodes and an arbitrary number of hidden nodes per layer (within the limits of GPU memory). In addition, each layer of the ANN is capable of performing RBF or sigmoid calculations.

The classifier receives as inputs real-valued features which are standardized prior to processing by the ANN. The classifier outputs a single real value, in the range of 0 and 1, which represents the confidence with which the pattern is a member of the positive class. These confidence values can be used for ranking during lift calculations, or for sensitivity or specificity measures with an arbitrary value chosen as a threshold for positive identification of a pattern. Therefore, the fitness function is not limited to lift at 20% as presented in this report.

3.2. Genetic Algorithm Design

For a single-layer ANN, as used in our final design, each candidate ANN is represented by a chromosome which consists of four arrays of real-valued numbers. Edge weights between input nodes and hidden layers, edge weights between hidden and output layers, width values for RBFs at hidden nodes and center values for RBFs at hidden nodes are all encoded, and can take on any real value. Figure 1 demonstrates the chromosome structure and its relation to ANN structure. This structure is simple yet powerful; individual edges are 'removed' by the genetic algorithm when their weight is set to zero and nodes are 'removed' by the genetic algorithm when their RBF width value is set to zero. Thus, while the topology of the ANN is limited to those with m layers and n nodes per layer, the genetic algorithm searches all topologies with m layers and between 0 and n nodes per layer.

The genetic mutation and crossover operations which we have implemented are based on the methods defined by Montana and Lawrence [1]. Of the operations presented in this paper, the MUTATE NODES and CROSSOVER WEIGHTS operations were determined to provide an optimal balance between performance and classifier accuracy, and have been implemented in all of our experiments.

3.3. Data Preprocessing Steps

In order to handle nominal feature values in data sets, we have implemented an algorithm which, given a tab separated values (TSV) file, converts nominal feature values in the data

set into numeric values. The algorithm replaces each nominal value in the data set with the logarithm of the probability ratio between the positive and negative classifications observed given the nominal value. This is inspired from the discriminant function of a Bayesian classifier trained only on that particular feature. As a result, meaningful numeric values can be extracted from nominal values without domain knowledge relating to the values. This improves on the traditional orthogonal coding method [5, slide 17] by avoiding allocation of a disproportionate number of input nodes to nominal features. The logarithm of the ratio of all positive classifications to all negative classifications is assigned to nominal values which do not appear sufficiently in the data set and for values which appear only in blind test data sets; this prevents the classifier from assigning significant values to nominal values for which little or no inference can be done.

Due to the inherent limitations of floating-point (FP) representation of real values, the range of inputs to the ANN plays a crucial part in the numerical stability of the classifier during training and during classification. In particular, our use of exponential functions resulted in an ANN very sensitive to output saturation. In these cases, while the input to the `expf` function is well within the range of representible FP numbers the mathematical output of the exponentiation is too large to be represented in FP. This causes degenerate output values such as infinity or NaN (not a number). These special values propagate through the ANN edges to produce degenerate values at the output of the classifier.

To avoid this issue, we standardize each feature individually before training and before classification. One method of standardization is to use the sample mean and sample standard deviation as shifting and scaling parameters (referred to as "z-score normalization" in [6]). However, sample standard deviation is not robust against outliers, while interquartile range (IQR) does tolerate outliers [7]. We use a variant of IQR: we estimate the range between the 10^{th} and 90^{th} percentiles and use that as the scaling parameter. After shifting and scaling, 80% of the resulting feature values are in the range $(-1, 1)$. This range of inputs to `expf` is narrow enough to always produce a valid FP value at the output. The standardization does not affect the classification itself because there are scaling and shifting parameters within the ANN (input to hidden edge weights and hidden center parameters, respectively) which are already part of the search space.

4. Implementation of GA Training of ANNs

Our GA begins by generating an initial population of random candidate ANNs, where each chromosome in the population is a set of parameters describing one ANN. Feature values for all training instances are loaded into GPU global memory in preparation for ANN computations. The following steps are then executed in sequence for each generation of the GA:

(i) Compute the ANN output corresponding to each training instance, as calculated by each candidate ANN in the population.

(ii) Find the threshold that defines the outputs which fall in the top 20% for each candidate ANN's set of output values.

(iii) Compute the number of top 20% instances which are truly positive, again for each candidate ANN. This allows us to calculate lift, which is used as the fitness value for each candidate ANN.

(iv) Apply genetic operators: mating, mutation and selection. The selected candidate ANNs become the new population, to be used in the next generation.

Items 1, 2 and 3 were all implemented on the CUDA platform and are described in detail below. Performance results are given as the combined time for one generation of these three

items but exclude one-time initialization. Item 4 is not very computationally intensive and was not parallelized.

4.1. Computation of ANN Output Values

ANN output values are computed by reading the training instances as real-valued feature vectors, applying them to the input nodes of the network (see Figure 1), and storing the output of the network. Specifically, each feature value is first scaled by the corresponding input-hidden edges. These values are then propagated to all hidden nodes, which apply a radial basis function. Finally, the hidden node outputs are scaled by the hidden-output edges and summed at the output node. A single real value is output from the network and stored in GPU memory. Parameters for edge weights and radial basis functions are given in the chromosome of each candidate ANN.

Feature data are organized to efficiently use memory bandwidth. The requirement for optimal use of CUDA memory bandwidth is that blocks of threads simultaneously access sequential addresses in memory, resulting in a coalesced single-clock cycle access. We store feature data in feature-major order (values for all instances of feature 1, all instances of feature 2, etc.). Therefore, when the thread block reads data for the first feature across different instances, the memory accesses can be coalesced. Similarly, accesses are coalesced for all subsequent features.

ANN output calculation is particularly well-suited to the CUDA architecture because it has a very high ratio of mathematical operations to memory accesses (160 multiplications, 4 exponentiations and 80 bytes of memory accesses for our ANN topology). In addition, each output value depends only on the current ANN parameters and one training instance's feature values. Therefore, parallelization of this step scales particularly well with hardware capabilities.

4.2. Calculation of Top 20% Threshold

Next, we must calculate the k^{th} largest value in each candidate's list of output values, where $k = 0.20 * n$ and n is the number of output values (number of training instances). We can accomplish this by iterating through all n unsorted output values and inserting each into a binary minheap of fixed size k. The heap size is kept fixed by removing the root (smallest node) after each insertion. At the end of this process, the root of the heap is the k^{th} largest output value. The insertion and removal can be combined into a single operation which costs $O(\log k)$ time. The heap occupies $4 * k$ bytes of device shared memory.

Because fast on-chip memory (16 KB) is not large enough for our value of k ($4 * 18936 =$ 74 KB), we instead use p (5) passes of $\frac{k}{p}$-sized heaps. Each pass examines only values which are less than the threshold calculated by the preceeding pass. For example, the third pass examines only values below the 8% threshold and finds the top 4% threshold of this subset of values. This threshold therefore is the 12% threshold of the full list. The threshold calculated by the last pass is the desired 20% threshold.

The overall complexity of our algorithm is $O(np \log \frac{k}{p})$. Each top 20% calculation is performed independently for each candidate ANN. Therefore, the parallel performance of this step scales with hardware capabilities when there are enough candidates to occupy the entire device.

4.3. Computation of Number of Truly Positive Top 20% Instances

We wish to compute the number of truly positive instances in the top 20% of scores. Assuming the training instances are unsorted, we must read from memory each instance's true class and corresponding ANN output value and compare the output value to the

candidate's threshold. For n training instances, this requires $2 * n$ memory accesses. However, by pre-sorting the training data such that the n_p positive instances appear first in memory, we need only read the first n_p ANN outputs, knowing that they correspond to the positive instances. This reduces the number of memory accesses to n_p, which is significant particularly for training data with low positive rate (in our case, $2 * n = 189364$ while $n_p = 2080$). Sorting the training instances by class is a cheap operation which is performed only once during initialization and can be done by the host CPU.

Counting of positive instances above the candidate's threshold is performed independently, in parallel, for each candidate ANN. However, because there may not be enough candidates to fully occupy the GPU, we also split the counting process for each candidate into 256 independent sub-counts: the first sub-counter processes training instances 0, 256, 512, ..., the second sub-counter instances 1, 257, 513, ..., such that the sub-counts can be executed in parallel. The striped assignment of training instance indices to sub-counters is necessary to take advantage of the full memory bandwidth of the GPU. The host CPU then aggregates the sub-counts by summing, which is a very lightweight operation.

Finally, each count is divided by the number of positive instances to give a positive rate in the top 20%. Then, the ratio of this positive rate to the overall positive rate is the lift of each candidate. These operations are also comparatively lightweight and executed on the host CPU.

5. Results

We first present the computational performance of our algorithm for classifier training, followed by the estimated classification performance of the classifier designed for the UC San Diego data mining contest.

5.1. Experimental Setup

To evaluate the performance of our system, we compared execution time of a sequential algorithm on a commodity CPU and an optimized algorithm on commodity GPUs. The goal was to estimate the speedup of our algorithm with respect to sequential without using the CPU's multiprocessing capabilities. Each series represents one type of experimental hardware (Table 1). Each data point is an average over 10 runs of the algorithm with the same parameters.

Table 1. Experimental hardware. Free RAM is OS-reported available memory before running each experiment.

Legend Label	Hardware	Clock	Free RAM	OS
x86	Intel Core 2 Q9450 CPU	2.66 GHz	7.5 GB	Linux 2.6 (64-bit)
GTX260	NVidia GTX260 GPU	1.24 GHz	N/A	Windows 7 (64-bit)
GTX275	NVidia GTX275 GPU	1.4 GHz	N/A	Linux 2.6 (64-bit)

5.2. Performance Results

In most cases, the GPU implementation resulted in approximately an order of magnitude speedup over a serial implementation of equivalent code running on a modern x86 desktop processor (Figure 2). For population sizes below 10, observed speedup decreases and no

speedup is observed in the single-candidate case. This is due to GPU kernel invocation overhead and data copying dominating execution time. However, in these cases, both implementations are still quite fast (20 ms per generation).

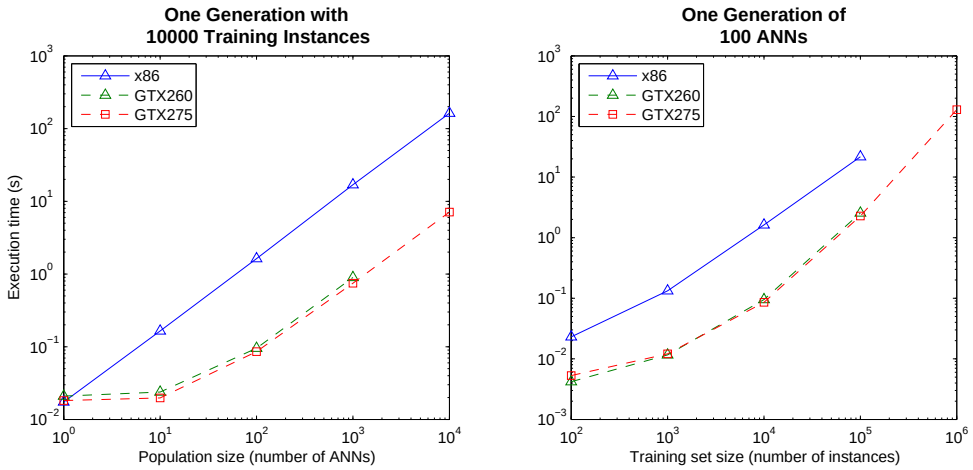

Figure 2. Training performance as a function of population size (left) and training set size (right). Note logarithmic axes.

5.3. Classification Performance

The classifier was trained over 12000 generations; the fitness metric during this training was lift at 20% calculated on a training set which consisted of 70% of the data set, chosen at random during initialization. After each generation, the candidate ANN with the highest fitness value was used to classify 20 random subsets of the hold-out data set (each consisting of exactly one half of the hold-out data set), and the average of the lift at 20% of these classifications was recorded. Figure 3 demonstrates experimental results across 12,000 generations of population 50. The final estimated lift is 4.51 out of a maximum lift of 5.

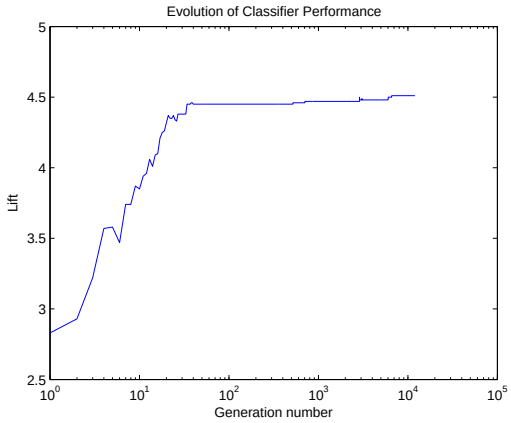

Figure 3. Evolution of classifier lift. Note logarithmic generation number axis.

6. Conclusion

We have presented an implementation of GA training of feedforward ANN classifiers for the CUDA platform for GPU programming. By carefully designing memory organization, algorithm computational load and memory access patterns, we have obtained a 10-fold speedup compared to a conventional sequential CPU implementation. A multipass approach was required in the selection step of classifier evaluation to compensate for the limited amount of fast on-chip GPU memory. Our method scales across population and training set sizes and is expected to be useful in other data intensive machine learning and data mining tasks.

7. Future Work

Montana and Lawrence [1] have demonstrated that a brief period of hill climbing after training with a genetic algorithm may increase the accuracy of classifier results. Thus, we could improve the accuracy of our methods by incorporating hill-climbing methods, such as the parallel backpropagation method described by Oei, Friedland and Janin [2].

In addition, parallelization methods of genetic algorithms involving multiple populations [8] may interact favourably with data mining applications. Seperate populations may be trained on different subsets of the training data, allowing for a more thorough search of the solution space which the training set presents.

[1] Montana D J and Davis L 1989 *Proceedings of the eleventh international joint conference on artificial Intelligence* vol 123 (Citeseer) pp 762–767 URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.77.3838&rep=rep1&type=pdf`

[2] Oei C, Friedland G and Janin A 2009 URL `http://www.icsi.berkeley.edu/pubs/techreports/TR-09-008.pdf`

[3] Halfhill T 2008 *Microprocessor Report*

[4] Diego U S 2009 UC San Diego Data Mining Contest URL `http://mill.ucsd.edu/`

[5] Nieminen P 2008 Multilayer Perceptron Neural Networks URL `http://users.jyu.fi/~nieminen/dm2008mlp/dm_mlp.pdf`

[6] Han J and Kamber M 2006 *Data mining: concepts and techniques* (Morgan Kaufmann) chap 2.4.2 ISBN 1558609016

[7] Andersen R 2008 *Modern Methods for Robust Regression* (SAGE) chap 2 ISBN 1412940729

[8] Alba E and Troya J M 1999 *Complexity* **4** 31–52 ISSN 10762787