

Classifying Handwritten Digits with the Artificial Neural Network and Genetic Algorithm in a Distributed Environment

Zach Firth* Stefan Novak†

May 9, 2010

Abstract

This study explores the integration of the genetic algorithm and artificial neural network in performing classification of handwritten digits in a distributed environment. Message Passing Interface (MPI) is used in the modeling of the environment in which many configurations of the artificial neural network are initiated and OPENMP is used to increase the runtime performance of the artificial neural network calculations. Although implementations of the serialized backpropagation algorithm converge on a solution more rapidly than the distributed implementation, the backpropagation algorithm has a tendency to converge into local minima, requiring additional tuning parameters to avoid such behavior. A superposition of both backpropagation and genetic algorithms are implemented to resolve this behavior.

Introduction

In today's society, people continually want things bigger, faster and cheaper. As a result, computational scientists have to be very good at finding optimal solutions to problems in the shortest amount of time. Many optimization problems of the past could be simply solved analytically, using a piece of paper or a simple serial program. However, many modern day problems are just too chaotic and complex to be solved in this fashion. As a result, parallel algorithms now play a vital role in solving many optimization problems. Typical optimization algorithms operate in a serial fashion in which one iteration must finish before the next begins. While this approach can work well for some problems, other problems are simply intractable without a major paradigm shift. Parallel optimization algorithms provide such a departure. A parallel optimization algorithm can either take advantage of multiple cores (i.e. OPENMP) or multiple

*zfrth@gmu.edu, George Mason University, Fairfax, VA, 22030

†snovak@gmu.edu, George Mason University, Fairfax, VA, 22030

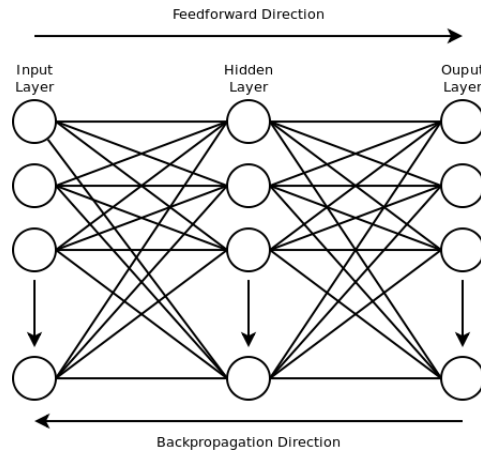


Figure 1: Three-layer, feed-forward artificial neural network used to process handwritten digit data.

computers in a distributed environment (i.e. MPI). By utilizing parallelism, one can perform many iterations simultaneously. This project explores the utility of parallelizing a genetic algorithm in order to optimize a program's ability to infer numbers from a set of handwriting samples. In order to accomplish this task, the genetic algorithm will optimize the weights of a feed-forward artificial neural network in conjunction with the backpropagation algorithm, a gradient-descent optimization technique for artificial neural networks. The data used to train the ANN was derived from the MNIST Database¹.

The MNIST database is a collection of handwritten character data originally developed by the National Institute of Standards and Technology (NIST) and post-processed and distributed by Yann LeCun² and Corinna Cortes³. The original data, comprised of the NIST Special Database 1 (SD-1) and Special Database 3 (SD-3), represent hand-written digits that have been digitized and then post-processed to increase the available resolution and translate the center of mass of each digit in order to increase the fidelity of the data. As a result, there are 60,000 training samples and 10,000 test samples, with a resolution of 28×28 pixels and 256 bit-depth, that can be used to test and verify the effectiveness of various machine learning algorithms. Although other methods have been shown to produce more successful results in character recognition when compared to the use of feed-forward artificial neural networks, such as the support vector machine [1] and the convolutional neural network [7], the use of the feed-forward artificial neural network to solve the given problem provides a unique opportunity to implement a distributed genetic algorithm.

¹The MNIST Database: <http://yann.lecun.com/exdb/mnist/>

²Courant Institute, New York University

³Google Labs, New York

Literature Review

Genetic algorithms use the theory of biological evolution to arrive at an optimal solution to a given problem. The genetic algorithm begins by randomly creating a set of solutions known as a population in which each solution is known as a chromosome and each component of the solution is known as a trait [2]. After the population has been created, a fitness function is used to evaluate the fitness of each chromosome within the population. The fitness function quantitatively calculates the quality of each solution. The next step in the genetic algorithm involves the selection of parents. During this process, chromosomes are chosen depending on their fitness function value. Imagine a dartboard where the size of the arc defining the point value is dependent on the value of the fitness function. Mating pairs are chosen by randomly “throwing two darts” at the board. The chromosomes in which the darts land become the mating pair. After the mating pair has been selected, they may swap parts of their genetic material (traits) to form a new generation (crossover), or simply move those parents on to the next generation. Next, a small number of chromosomes in the population are chosen for mutation of a random number of traits in its chromosome. The process of selection, crossover and mutation is then done for many generations until the fitness function has reached an optimal value or a pre-defined number of generations have been completed [2].

The artificial neural network is a general algorithm that can be utilized to approximate a real-value or discrete-value target function [3]. With advancements in computing technology, artificial neural networks have been successfully used in a variety of various implementations, such as the Autonomous Land Vehicle in a Neural Network (ALVINN) [5] and in document character recognition[7]. In this implementation, a three-layer feed-forward artificial neural network is constructed and trained to classify digits from the MNIST database. The first layer in the system, the input layer, consists of values that correspond to the pixels of the digitized image of the digit to be classified. The last layer in the system, the output layer, consists of ten nodes used to classify the digit with the given input—the rank of the node with the highest value corresponds classification type for that sample. A simplified version of this system is shown in Figure 1.

Results

The digit recognition problem was implemented through a small population of 22 members on a single node machine. Figure 2 visualizes the evolution over the course of 100 generations. Although a maximum success rate of 25% was achieved, higher success rates can be gained when running the genetic algorithm for longer durations.



Figure 2: The fitness function shown for a population with 22 members over 100 generations. The color shown for each instance corresponds to a success rate shown in the legend. The success rate was calculated by validating the artificial neural network configuration with the MNIST test data.

Local Speed-ups from OPENMP

As previously mentioned, the greatest bottleneck in the code was the repeated calls to the `CALCULATESUCCESSRATE()` function which takes the current artificial neural network configuration and calculates the classification success rate given the test data set of 10,000 MNIST images. When this code is parallelized with the use of `OPENMP`, a linear speedup is observed, as shown in Figure 3. The test cases used to assess the code speedups used the same configuration as the data shown in Figure 2.

Technical Challenges

There were several technical challenges that were faced in the course of this study, ranging from the integration of the MNIST data set to implementing MPI into the genetic algorithm code.

Interfacing MNIST with the artificial neural network implementation

In order to properly integrate the processing of MNIST data as an artificial neural network with the functionality of the genetic algorithm, several design decisions were made to ensure efficient integration. The artificial neural network was constructed as a structure within the C program. This allows the object to contain pointers to various positions within the MNIST database. When multiple networks are executed on a single system, it is vital that the training and test data sets be loaded a single time and reused as often as possible since the complete data set occupies approximately 425 megabytes in memory. Therefore, instead of creating multiple instances of a neural network, which results in unnecessary duplication of memory, a single instance can be used in which data

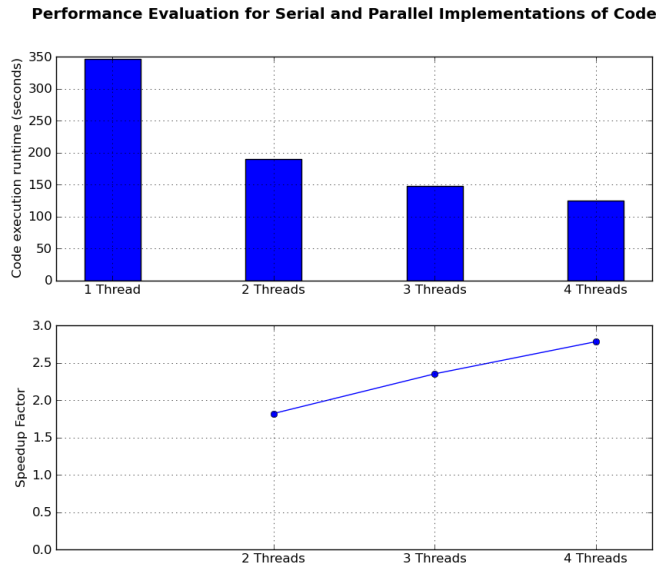


Figure 3: Code execution speedups due to using OPENMP to parallelize fitness calculations. The same configuration was used for each test, which consists of a population size of 22 members and a limit of 100 generations.

references can be swapped in and out. For example, the follow code segment steps through ten training cases, propagates the inputs through the network, applies the backpropagation algorithm to update the weights of the system, and repeats:

```

for (i=0; i<10; i++)
{
    network.systemInput = trainingData[i];
    network.targetOutput = trainingLabels[i];
    PropagateInputsForward(&network);
    PropagateErrorBackwards(&network);
}

```

Interfacing the artificial neural network implementation with the genetic algorithm

This agility in the network representation is applied to the weights of the neural network as well allowing for easy integration with the genetic algorithm. As a result, the previous code implementation is modified to update the weights of the network as the genetic algorithm steps through each member of the population and apply the backpropagation algorithm to evolve each member of the

population towards a minimum:

```
for ( i = 0; i < (ga.popSize); i++)
{
    network.weights = &myNewPopulation [ i * (g.numWeights) ];
    network.systemInput = trainingData [ trainingDataID ];
    network.targetOutput = trainingLabels [ trainingDataID ];
    PropagateInputsForward(&network);
    PropagateErrorBackwards(&network);
}
```

Testing

Several challenges arose during the implementation and testing process. The first challenge involved optimizing the algorithm to work efficiently in parallel. In order to do so, we first analysed the code to determine what loops were taking the longest. It quickly became apparent that the largest bottleneck in the problem was not the MPI version of the genetic algorithm, but in the serial computation of the fitness of each chromosome. As a result, OPENMP was used to divide the task of computing the fitness for each chromosome among multiple threads. The next challenge encountered was properly setting up the submit script to the gmice cluster. The "#PBS -l select=..." line had to be modified to provide n MPI nodes to each have m cores for the OPENMP threading. After trial and error the line was changed to "#PBS -l nodes=m:ppn=1:cpp=m" where m and n were as defined above.

Future Work

The feature set used in modeling the genetic algorithm in this study was relatively simple. The only two features utilized were crossover and mutation. However, there are many other potential additions that could be added in the future to make the algorithm more robust. The first possible addition could be the implementation of a tabu list to the genetic algorithm. Tabu-genetic algorithms operate in much the same way that a standard genetic algorithm operates, with the addition of ancestry to each chromosome. That is to say that each chromosome in the population has a list enumerating its last n mating partners. Members on this list are said to be tabu because they are too genetically similar to the chromosome. Tabu-genetic algorithms essentially prevent inbreeding within the population, thus resulting in a more diverse population of chromosomes[6]. The addition of a TGA to the current code could be made easily with little change to the current algorithm.

Dynamic parameter encoding is another potential enhancement that could be added to the algorithm. In the current implementation, parameters, such as the rate of mutation, crossover, and the size of the population are fixed at the beginning of the simulation. In a genetic algorithm with dynamic parameters,

these values are often a function of the results produced by the genetic algorithm. For example, if the standard deviation of the populations fitness was low, the genetic algorithm would recognize that the population has become trapped in a local optimum. This would likely result in the genetic algorithm boosting the rate of mutation to try and break free from the local optima. Additionally, if the fitness of the overall population is relatively high, the genetic algorithm may decrease the rate of mutation and increase the rate of crossover. This would allow the genetic algorithm to focus in on the global maxima [8].

Additionally, other forms of the artificial neural network have demonstrated increased performance, such as the polynomial neural network[4], which could be utilized to further the classification abilities of the system. Although other computational learning algorithms exist, it would be challenging to integrate their approach with the genetic algorithm.

Conclusion

The combination of both local and global optimization schemes for solving an artificial neural network system can provide the largest increase of performance for computational learning problems. When transitioned to a high-performance computing environment, the dynamics of the genetic algorithm can achieve optimal performance levels due to the increased interactions with other instances of the problem. Since MPI was used to provide the framework for running the genetic algorithm in this environment, the scalability of the complexity of the genetic algorithm is limited by available distributed resources.

In addition, local code optimization becomes absolutely vital since many copies of the same program are running at any given time: the system will achieve an effective speedup that is proportional to the size of the system. When profiling the code, it becomes clear that the primary bottleneck consists of numerous calls to calculate the fitness of a system. The only acceptable way of doing this is by validating the artificial neural network configuration with the available test data—a relatively easy task to parallelize since the calculations can be modeled as a parallel reduction. As a result, OPENMP was able to provide a substantial speed-up in code with just a few, simple lines of preprocessor directives.

Since there are outstanding issues in evaluating the performance of the system in a distributed environment, it is difficult to assess how well the genetic algorithm can perform in the distributed environment. Before the system performance can be fully assessed, the configuration of the genetic algorithm needs to be optimized by the means of a parameter scan or Monte Carlo analysis—even a simple parameter scan would consist of well over 700 cases that need to be performed. Once the dynamics of the genetic algorithm and the learning rate of the artificial neural network can be balanced out, the code will be ready for production runs.

In retrospect, the problem that was chosen to demonstrate some of the concepts from the field of high-performance computing, the application of compu-

tational learning algorithms for recognizing hand-written digits, ended up being quite a daunting task. Integrating MPI and OPENMP into the code implementation was relatively trivial, but trying to demonstrate the effectiveness of those technologies was overshadowed by the complexity of the problem that was being solved. Perhaps it would have been more suitable to select a different global optimizer in place of the genetic algorithm given the time restraints on the study, but then again, modeling the dynamics of the genetic algorithm in a distributed environment was proved to be insightful and provided a fresh perspective on distributed computing—that is, modeling physical interactions in a distributed environment.

As computing systems become increasingly distributed and networked, new and innovative concepts have to be explored to fully utilize those technologies. Unfortunately, the rate at which technology is evolving makes it difficult to develop new coding paradigms and tools that can properly coexist with those technologies. In the case of computational learning, having a flexible code framework that can exist in a distributed system allows scientists to model increasingly complex problems in an intuitive manner. Once this framework has been developed, computational scientists can focus on new and exciting problems instead of being bogged down by the responsibilities of low-level code implementations.

As the authors of this study, we look forward to further developing these ideas and exploring alternative means for migrating computational learning problems into a distributed computational environment.

References

- [1] D. Decoste and B. Scholkopf. Training invariant support vector machines. *Machine Learning Journal*, 46(1–3), 2002.
- [2] James F. Smith III and Robert D. Rhyne II. Genetic algorithm based optimization of a fuzzy logic resource manager: Data mining and co-evolution. *Proceeding of the International Conference on Artificial Intelligence*, 1:421–428, 2000.
- [3] Tom M. Mitchell. *Machine Learning*. The McGraw-Hill Companies, Inc., 1997.
- [4] Sung-Kwun Oh and Witold Pedrycz. The design of self-organizing polynomial neural networks. *Information Science*, 141:237–258, 2008.
- [5] Dean A. Pomerleau. Alvin: An autonomous land vehicle in a neural network. Technical Report CMU-CS-89-107.
- [6] Chungnan Lee Shu-Ching Chen Sheng-Tun Li, Chuan-Kang Ting. Maintenance scheduling of oil storage tanks using tabu-based genetic algorithm. 2002.

- [7] Steinkraus-D. Simard, P. Y. and J. Platt. Best practice for convolutional neural networks applied to visual document analysis. *International Conference on Document Analysis and Recognition (ICDAR)*, pages 958–962, 2003.
- [8] D. B. Fogel Thomas Baeck and Z. Michalewicz. *Advanced Algorithms and Operations*. Taylor & Francis, 2000.