Skin Cancer Image Classification with Parallelized Deep Artificial Neural Network

Matthew Liddy
Department of Computer Science
College of Science and Engineering
Texas Christian University
Fort Worth, Texas 76129
Email: matthew.p.liddy@tcu.edu

Sellars Levy
Department of Computer Science
College of Science and Engineering
Texas Christian University
Fort Worth, Texas 76129
Email: s.n.levy@tcu.edu

Westen Riley
Department of Computer Science
College of Science and Engineering
Texas Christian University
Fort Worth, Texas 76129
Email: westen.t.riley@tcu.edu

***Abstract*—Deep neural networks can often take hours, or even days of training time to converge to an acceptable minima. This semester's research project investigates the application of high-performance parallelization techniques in deep multi-layer perceptron artificial neural networks for binary image classification. The problem our artificial neural network is attempting to solve is that of classifying skin cancer images, such as carcinoma and melanoma, as malignant or benign. Skin cancer arises from the development of abnormal cells that, if malignant, have the ability to invade or spread to other parts of the body. Such cancer usually arises in the form of a raised area on the surface of the skin. However, visually identifying if a raised area is indeed malignant is not a straightforward task as it depends on a variety of abstract and subjective characteristics, including asymmetry, border irregularity, color, diameter, and evolution. Due to these subjective factors, doctors often struggle to distinguish between malignant and benign visually. To better visually classify these categories, we can implement an artificial neural network. Specifically, a state of the art deep multi-layer perceptron artificial neural network that will classify an image as either malignant and benign. These networks are systems with interconnected nodes that work much like neurons in the human brain. Utilizing algorithms, such as gradient descent, we can train these networks to recognize hidden patterns and correlations in raw data, cluster and classify it, and — over time — continuously learn and improve. The downside to implementing such a neural network is that it will require a large number of nodes, increasing the computational complexity and training time. Our solution to this problem is to utilize parallelization techniques and technologies, including CUDA, MPI, and OpenMP, to ultimately improve the training time of our neural network.**

I. Introduction

Skin cancer is a disease that affects one in five people, and it is not so easy to detect. It generally appears in the form of raised skin, such as a mole, and can be almost indistinguishable from a completely harmless, benign mole. Because of the various subjective characteristics that distinguish it as malignant, dermatologists often are incapable of visually classify moles as malignant or benign and normally use visual inspection to determine if a biopsy is needed. A biopsy is currently the most effective and accurate way to classify skin cancer as malignant or benign. However, biopsies can take time to run and get results, allowing the disease to potentially grow and develop further. Skin cancer detected in the early stages can be treated with up to a 95% survival rate. Yet, if that same cancer is left unnoticed and untreated long enough to develop to its second stage, that person's survival rate just dropped from 95% to 23%. It is for this reason that early detection of skin cancer is crucial in lowering the mortality rate associated with skin cancer. To remedy this, many studies have been training neural networks to classify skin cancer as malignant or benign to create an easily accessible tool to detect skin cancer.

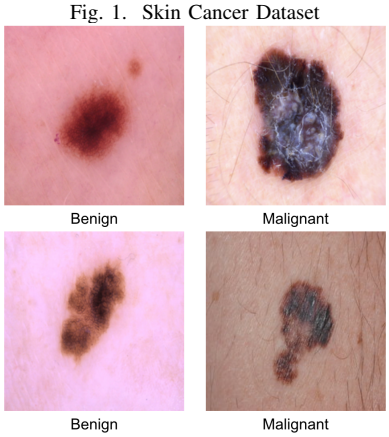
Training neural networks is a tedious and time-consuming task, particularly when it comes to image classification. A single image can have thousands of pixels, and when split up by Red-Green-Blue values, the size effectively triples. Having a large number of input pixels can cause a sequential stochastic gradient descent algorithm to become cumbersome in time, spending hundreds of hours to potentially converge on an accurate minimum. No matter how well optimized sequential code is written, it is almost always beaten by code written in parallel.

There are many ways to write accurate and fast neural networks. However, for this semester's research project, we had limitations for what we were allowed to use. We were not allowed to use significant libraries or tools such as Google's tensor flow to create our artificial neural network. We were also specified to use C++ as the language of choice, meaning that the versatility of python was not an option for our team. Thus our team

was left to create from scratch our own artificial neural network along with the parallelization to speed it up.

Our task for this semester project was to parallelize image classification using an artificial neural network. The training set we had access to was 3600 images of malignant or benign melanoma skin cancer in which we split up into training, validation, and testing sets (see figure 1). Our neural network is stochastic, meaning

Fig. 1. Skin Cancer Dataset



that forward propagation, backpropagation, and weight updating will be run individually on every image in our dataset. This is a very slow process but is a surefire way to converge to a minimum. It is slow due to the fact that for every image, we need to do matrix-multiplication on every layer. The most time-consuming layers to do this operation on is the first to the second layer in our network. The first layer contains $150,528$ nodes, which is derived from $224 \times 224 \times 3$, or the image size multiplied by 3 to include RGB, as stated above. The second layer contains 256 nodes, one for each of the red-green-blue values, so we do matrix-multiplication with two vectors of size $150,528$ and 256, leading to massive computations for every image in our dataset. This network training is indeed a very slow process.

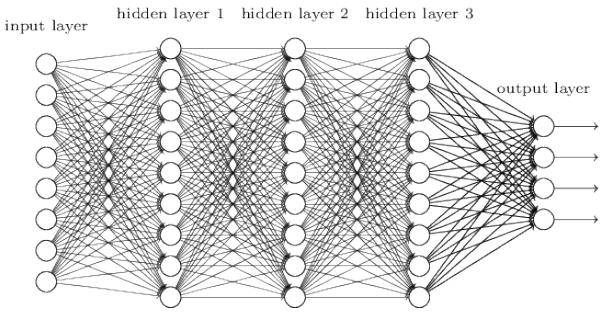
What our artificial neural network is trying to do is visually learn the difference between benign and malignant melanoma. To the untrained eye, they may look similar, if not identical. However, with a trained eye, one will be able to tell that benign skin cancer is usually smaller, visually darker, more round, and less sporadic. So with each image it is trained on, the neural network takes a step closer to understanding how to spot the difference between two types of cancer visually.

Our main target for parallelization was the matrix-multiplication operations since they consumed the majority of time for each pass. The solution was simple: using CUDA to do the matrix-multiplication for us. Instead of using the few cores on a CPU, CUDA utilizes the thousands of smaller cores on a GPU to do the calculations in parallel. This change generates a significant speedup in our program. Before CUDA, each pass took well around 30 seconds, whereas after we implemented CUDA, each pass takes around 5 seconds. Other methods for parallelization were using OpenMP to parallel for-loops during initialization and MPI message passing for matrix transposition.

Even though we have parallelized our neural network, it still took a significant amount of time to run through all 50 epochs. Since image classification is trying to condense an input of $150,528$ into 1, it can take a large number of epochs until it converges. In our case, we had 2109 images in our training set, each of which took around 5 seconds for one pass. This meant that if we wanted to run through 50 epochs, it would take a little less then a week, which seems like a long time. However, if we had a speedup of 6, i.e, before parallelization, it took 30 seconds per each pass and training the model through 50 epochs would have taken over a month to run. This is why parallelization is so important. When used correctly, parallelization can save significant amounts of time for everyday programs. Instead of taking a month to wait for our neural network to converge, we only had to wait a few days.

II. Background and Related Research

Artificial Neural Networks have long been utilized for classification as well as regression. There are many different network variations, but we have chosen a deep multi-layer perceptron artificial neural network with fully connected layers.



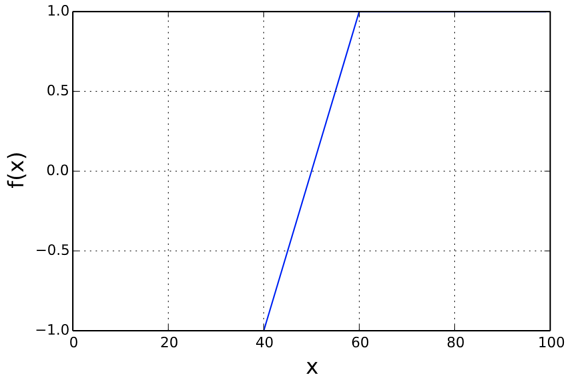
Similar to Ercal F. et al. [1], we selected this structure as it is a general-purpose connection pattern and makes no assumptions about the features in the data. However, a side effect of this structure is that it results in a large number of weights. Approximately, the product of the

size of the current layer and the bias multiplied by the size of the next layer $(|L_i| + 1) \times |L_{i+1}|$.

A large number of weights θ becomes computationally expensive when we apply gradient descent. Traditional stochastic gradient descent algorithms begin with feed-forward propagation. This algorithm is used to calculate the output vector Y from a given input vector X. This computation entails calculating the output for each neuron in the network $\sum_{i=1}^{n} x_i \theta_i$. The activation function a using hard tanh is then applied to this output.

Hard Tanh

$$a_j^i = f(x_j^i) = \max(-1, \min(1, x_j^i))$$



Instead of calculating these outputs individually for each layer, we vectorized our solution so we can calculate any output o for each layer in a single matrix multiplication step.

After the network computes the output value $\hat{y}$ it is compared to the target value y using the mean squared error cost function $J = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$. With the loss computed, the network's weights need to be updated to improve the network for the next pass. This weight update computation is completed with the use of the backpropagation algorithm. The goal of this algorithm is to compute the partial derivatives $\frac{\partial J}{\partial \theta}$ of the cost function J with respect to any weight θ (including bias of each node) in the network. Instead of computing the change in weight $\Delta\theta$ for each weight individually we utilized matrix multiplication as well as a matrix transpose function to simplify the calculation to $\Delta\theta = \frac{\partial J(X,\theta^t)}{\partial \theta} \times \alpha$ where α is the learning rate.

A common problem with deep artificial neural networks is the exploding gradient problem. We overcame this problem with a technique outlined by R. Pascanu et al. [2] called gradient clipping (see algorithm 1).

Algorithm 1 Pseudo-code for norm clipping the gradients whenever they explode

$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$
if $\|\hat{\mathbf{g}}\| \geq threshold$ **then**
  $\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|}\hat{\mathbf{g}}$
end if

After solving these minor challenges, we now had a completely vectorized solution for gradient descent. The benefit of vectorizing the gradient descent algorithm is that we can take advantage of modern parallelism in the Central Processing Unit (CPU) and the Graphics Processing Unit (GPU). Likewise, with the recent advances in technology, multicore processors and powerful GPUs are affordable to everyone. Parallelization is the approach of designing a computer program or system to process data in parallel. Artificial Neural Network Parallelization has been researched before—to much success. For example, In Nickolls et al. [3] and Che et al. [4], the authors explore the use of CPU-based parallel approaches such as MPI, Pthreads, and OpenMP. They also experiment with streaming processor cores interconnected to external DRAM partitions. Jang et al. [5] described a Multi-Layer Perceptron Network using CUDA and OpenMP to classify text. Similarly, F. Nasse et al. [6] created a face detection Convolutional Neural Network and observed the benefits of GPU acceleration.

This project will attempt to combine the tools and techniques utilized in the above literature as well as produce similar results.

III. Parallelization Technique

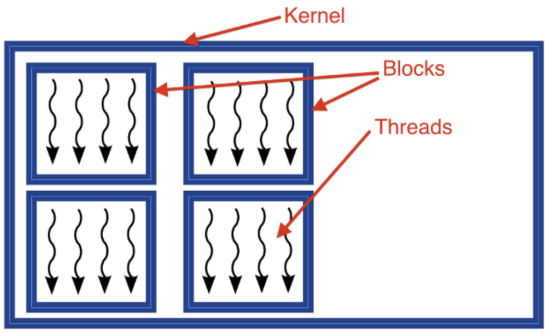
Initially, our neural network was designed to run sequentially; however, for obvious reasons, this causes the neural network to spend days or even weeks to train itself. This is unacceptable and way too slow, so we used a variety of parallelization techniques to speed up our computations and training. The three main parallelization techniques we implemented are CUDA, OpenMP, and MPI.

The first technique used, CUDA, is for matrix multiplication. The GPU has thousands of slower processors on it that can do multiplication computations at a slower rate in contrast to a CPU with a single, much faster processor, but, since there are thousands of them, the GPU can do thousands of these computations at the same time. As a result, for very large matrices, CUDA is able to perform a large number of computations necessary for matrix multiplication in a fraction of the time needed to do it sequentially. Also, in order to speed up access time, CUDA requires matrices to be represented in one

dimension and use the number of rows and columns to index into the ”matrix.”

The first thing needed to be done for CUDA matrix multiplication is to allocate memory. Since we already have allocated memory for the matrices to be multiplied, we need only allocate memory on the host for the resulting product matrix. However, we need to allocate memory on the device GPU for all of our matrices, both the two matrices being multiplied and the resulting product matrix. When the host calls the matrix multiplication function that is to be executed in parallel, it launches what is called a CUDA kernel.

The kernel has a particular job that it wants to complete as efficiently as possible, and it does so with the use of threads. The kernel's job is then divided up into a series of subtasks that need to be accomplished in order for the job to be completed. Each of these subtasks is divided among the threads in the kernel. These threads are then grouped into clusters called blocks, which are then further grouped in grids. Each of these blocks can hold up to 1024 threads, or 512 on older GPUs, and each kernel can launch in parallel as many blocks as the GPU can support.



The next technique we implemented in our neural net is MPI, or message passing interface. Initially, we used MPI for our matrix transpose function since it can be exhausting to do sequentially. Instead of having to swap the position of each element individually, we can use message passing to divide up the work in blocks between processors. However, we came to find that it wasn't the fastest parallelization technique for our matrix transposition function, which leads us to our final technique.

The last technique incorporated into our project was OpenMP. We have a variety of for-loops throughout our program for reading in data, updating values, etc., and it can take a while to iterate over every element sequentially. That's where OpenMP comes in to parallelize these tedious loops by dividing the work over multiple different threads. However, the main benefit we found with OpenMP was its use of shared memory allowed for a much faster matrix transposition function since we no longer needed as much communication and were able to vastly reduce the overhead. Instead of separating the data into blocks between processors, each thread is given its own row and transposes it into its own column in a new matrix so that no two processors should ever try and access the same memory location for its row in the input matrix or its column in the output (transposed) matrix. We experimented with a few different implementations by parallelizing various loops. However, after testing the OpenMP parallelization with both row parallelization and column parallelization, we found that there was not much difference in the execution time when the pragma omp statement was switched between the outer (row) for-loop and the inner-nested (column) for-loop. Also, since CUDA requires matrices to be represented in one dimension, both column and row vectors are stored the same way and do not need to be transposed before matrix multiplication.

IV. Summary of Results

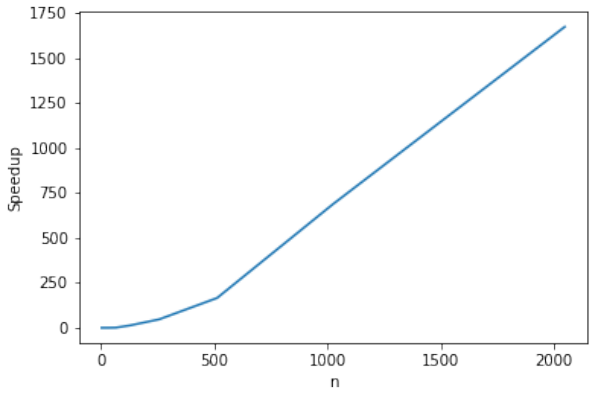
A. Computation Environment

For both the sequential version and the parallel accelerated version of the training algorithm we used the following hardware and software environment: The programs were compiled by the Nvidia CUDA NVCC 9.1.85 and run on 10-Core 3.3 GHz Intel i9-9820X with two Nvidia RTX 2080 Ti 11 GB with NVLink and 64 GB of DDR4 3000MHz.

B. Performance Analysis

Upon analysis of the CUDA Matrix Multiplication, we found significant improvements in overall speed. As documented in the following table, we found a noticeable difference between CPU and GPU multiplication of an $n \times n$ matrix. As a result, both our feed-forward and backpropagation vectorized algorithms were able to take advantage of this speedup. Overall, our parallelized network achieved a total speedup of 6.

n	CPU (ms)	GPU (ms)
2	0.001984	0.147584
4	0.00224	0.15856
8	0.001984	0.144384
16	0.003936	0.153248
32	0.01632	0.141248
64	0.122528	0.151328
128	1.384256	0.103712
256	10.148064	0.21632
512	163.643738	0.987296
1024	2849.583496	4.145984
2048	33529.48047	20.041183



V. Conclusions, Lessons Learned, and Future Work

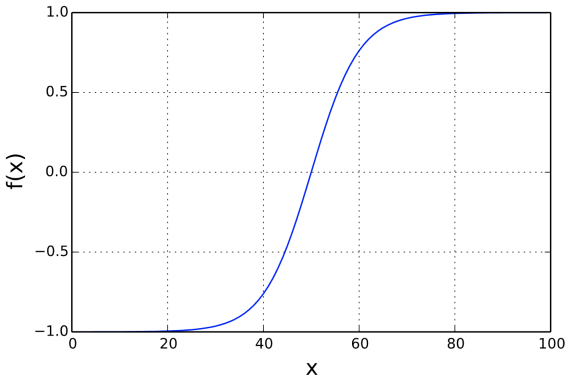
As we have seen, training a neural network to image classify is a very time-consuming task to perform. With the hundreds of thousands of pixels being simplified down into one output - it may take several days for even a parallelized neural network to converge. That said, it will still normally beat a sequential neural network. Given our limitations of not being able to use certain libraries, and having a limitation to use only the C language, we are proud to present our image classifying artificial neural network. With our parallelization techniques achieving a speedup of around 6, we can see how using parallel structures can impact the everyday world to make everything run faster. Cutting down our training time of over a month to just over six days is a fantastic achievement.

Through our research and experimentation, we were able to optimize our neural network and achieve a speedup of roughly six by implementing various parallelization techniques. As a result, our neural network was able to complete thirty epochs of the training data in only ninety hours, as opposed to the three hundred hours it would take to train the same neural network sequentially. The bulk of our speedup was achieved in our matrix multiplication function by using CUDA to perform a large number of computations in parallel on the GPU's thousands of processors. Another significant optimization we had came from using MPI for our matrix transposition. We were also able to reduce the number of matrix transpose function calls by implementing flattening and storing all of our two-dimensional matrices as one-dimensional vectors. We then use the number of rows and columns of the matrices to index into the one-dimensional vector, allowing us to represent the one-dimensional vector as a two-dimensional matrix.

The two main problems we encountered during our project were the vanishing gradient and the exploding gradient problems. These were mainly due to our activation functions, so we spent a good deal of time researching and testing out different activation functions to find out which one worked best for our neural network. Initially, we were using the tanh() activation function, but it kept giving too small a gradient for too large of numbers causing the gradient to become increasingly small and "vanishing." When this happens, the gradient is too small to make any significant impact on the weights, so consequently, the weights get stuck.

tanh

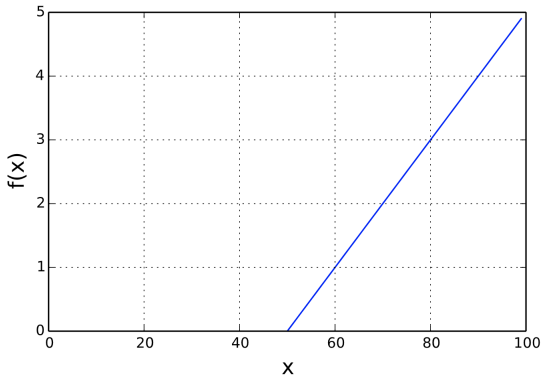
$$a_j^i = f(x_j^i) = \tanh(x_j^i)$$



To counter this, we switched our neural networks activation function to ReLU; however, this resulted in the exploding gradient problem where the values became increasingly large, causing the gradient to continuously grow bigger and bigger until it reaches infinity.

ReLU (Rectified Linear Unit)

$$a_j^i = f(x_j^i) = \max(0, x_j^i)$$



Another problem we encountered was the rand() function in the standard C library. Rand() is a pseudorandom number generator that generates an integer between zero and RAND_MAX. The problem lies in the fact that this function lacks a distribution engine. Therefore, if a smaller range is required and the modulus operator is used to with a new maximum value of m, and m is not divisible by RAND_MAX, then the distribution will not be uniform. To remedy this, we implemented mt19937 as our random number generator. This is a Mersenne Twister based on the prime $2^{19937} - 1$ and allows us to generate a real uniform distribution when shuffling the dataset as well as generating random weights.

Over the course of this project, we identified several approaches to improve both the parallelization and the performance of the model generated. One technique that we would like to implement in our future work is a process called mini-batching. We currently use a process called stochastic gradient descent. This, as described above, does forward propagation, backward propagation, and weight updating for each individual pass in our neural network. Another gradient descent algorithm is called batch gradient descent that updated all the weights after an entire epoch as passed. Stochastic is computationally cheaper and programmatically simpler than batch. Batch, however, is great for largely moving in the right direction of convergence, yet it computationally expensive since it has to store a whole dataset worth of information. This is where mini-batch comes into account. Mini-batch takes the best of both Stochastic and Batch gradient descent and combines them into one. Mini-batch is able to combine the power of batch, while also giving it the speed and cheap-to-compute algorithm of Stochastic. This method can also contribute to the parallelization of the network as we could run multiple training examples in feedforward in parallel and only be bound by a serial weight update when each mini-batch completes.

Moving forward, we could also implement a more aggressive optimization algorithm—Adam. This is an adaptive learning rate optimization algorithm that leverages momentum when updating weights to ultimately speed up convergence. Our problem would be a perfect candidate for this algorithm because it works well with non-stationary objectives and problems with very noisy and/or sparse gradients.

Going further, we could also implement dropout layers. This refers to randomly dropping out, or turning off, neurons in a neural network. This Google patented technique forces the neural network to learn more robust features that are useful in conjunction with many different random subsets of the other neurons. Although it results in more iterations to converge, it results in a more generalized model and less overfitting.

Another technique we identified was the use of convolution. This is the current state of the art technique for image classification that takes advantage of local spatial coherence. It uses convolutional layers, also known as filters, that extract the features from the input image but also preserves the relationship between pixels. This strategy models the way the human brain sees by essentially compressing the image but also preserving essential details. Convolution, in combination with max-pooling, results in a significantly reduced input vector. In turn, the network size is substantially reduced. This results in faster training times and a more generalized model. We did some preliminary testing of this strategy with Google's Vision AI and received a model accuracy of 88% in under 3 hours of training.

Ultimately, over the course of this semester project, we not only learned more about parallelization techniques but also neural networks. We believe that the synergy of these two domains is paramount as the field of machine learning grows. Also, we are not done here. Moving Forward, we will continue to improve on both accuracy as well as performance. Be on the lookout for our upcoming final research paper.

REFERENCES

[1] Ercal F., et al. *Neural network diagnosis of malignant melanoma from color images.*. IEEE Trans Biomed Eng, 1994.

[2] R. Pascanu et al. *On the difficulty of training recurrent neural networks.*. MLR Press, 1994.

[3] Nickolls et al. *NVIDIA Tesla: A Unified Graphics and Computing Architecture.*. IEEE Micro, 2008.

[4] S. Che et al. *A performance study of general-purpose applications on graphics processors using CUDA*. Journal of Parallel and Distributed Computing, 2008.
[5] H. Jang et al. *Neural Network Implementation Using CUDA and OpenMP*. Digital Image Computing: Techniques and Applications, 2008.
[6] F. Nasse et al. *Face detection using gpu-based convolutional neural networks, in: Computer Analysis of Images and Patterns*. Springer, 2009.