SOFT354 – Parallel Computation and Distributed Systems

10555972 | Plymouth University | Computer Science | 16th January 2020

CONTENTS

INTRODUCTION

In this report I am implementing both serial and parallel versions of a neural network to compare the performance differences when using back-propagation to train the network. To complete this I will first use C/C++ to write the serial implementation and then use the Compute Unified Device Architecture (CUDA) to implement the parallel version. CUDA is a parallel computing platform designed and created by NVIDIA and it allows for computing on NVIDIA graphics processing unit's (GPU's). Additionally, it works as an extension to the C programming language. This will allow the algorithm to run on the systems GPU. A modern GPU has many streaming cores that contains lots of threads which makes them very useful when computing intensive and repetitive algorithms.

Artificial neural networks (ANN) are one the of most popular tools used throughout machine learning. As their name suggests they were made to roughly copy what happens in our brains. Although the idea of neural networks has been around since the late 1940s (Hebb, 2002), it is only recently that they have become a big part of machine learning due to the backpropagation method. Backpropagation is one of the most efficient training algorithms due to its simplicity as it is just an application of the chain rule and gradient descent (Le Cun, 1988, pp.21-28). The algorithm is designed to minimise the error between the input values and the target values that are both fed through the algorithm. This is achieved by calculating the delta rule for each layer and adjusting all of the weights in the network accounting for the error. It is called backpropagation because the algorithm starts at the end of the neural network and works backwards, updating each layer's weights as it goes.

The application for ANN's today are vast, and they're being used in many different industries. This ranges from medical diagnosis, predicting stock on the financial market and object detection. One of the latest and popular applications of neural networks is autonomous driving. This is possible by using many cameras and sensors and continuously completing a feed forward pass of a trained network to work out what the next appropriate actions are to take. It would take thousands of hours of video footage with all possible scenarios to be able to train a network like this. ANN's are able to train using either supervised or unsupervised learning, for the purpose of this report I am going to use a supervised approach. I will be using a dataset that contains 4 features about 3 different types of 150 iris flowers. The features are: sepal length, sepal width, petal length and petal width. I will be using this as my input data and my target data will be the type is iris plant, which is either: Setosa, Versicolor or Virginica.

The algorithm used to backpropagate through a network and train it contains a lot of matrix mathematics, this means the system is having to run through large matrices and perform the dot product for lots of multiplications. This can take a very long time on a systems CPU as it can only compute one calculation at a time. The time taken to train a network is then also greatly increased by the number of iterations the backpropagation algorithm takes place, this can often be much greater than 1 million times (depending on the dataset size and the appropriate

training period). Due to these factors it makes a lot of sense to complete the matrix mathematics in parallel as many of the results from sections of a matrix multiplication do not depend on themselves.

The standard formula to complete matrix multiplication for any size is given by: $C_{ij} = \sum_{k=0}^{width(A-1)} \sum_{h=0}^{height(B-1)} A_{kj} B_{ih}$. For multiplying two matrices together of size $m \times a$ and $a \times n$ the total number of calculations required is: $m \times n \times (2a - 1)$, this then gives the complexity of $O(n^3)$. Parallelizing this algorithm does not change its complexity, however, it will reduce its total run time by a factor of $1/a$ (Kirk and Hwu, 2013, pp.111). This results in much faster training times for a neural network. This is extremely useful when training deep neural networks with many layers, on large complicated data sets. This would usually take a very long time but can be significantly reduced by implementing a parallel method.

IMPLEMENTATION

As previously stated, I will be using a dataset containing features on 150 different iris flowers. Figure 1 shows the 4 features of data I will be using as the inputs and the species of the plant that I will be using for the target values. Due to linear regression the network will output continuous values which means that instead of outputting the species name it will output a value between 0 and 1 that will correspond to a species.

sepal_length	sepal_width	petal_length	petal_width	species	network_output
6.9	3.2	5.7	2.3	virginica	0
4.9	3	1.4	0.2	setosa	0.5
5.1	2.5	3	1.1	versicolor	1

Figure 1. Example snippet of the iris plant dataset.

Figure 2 shows the design I will be using for the neural network I am going to implement. The network will have an input layer of 4 inputs, a hidden layer where the number of nodes can vary, and an output layer with 1 output. I will design the system so that then number of input nodes, hidden nodes and output nodes can all be changed. This will allow the network to be much more versatile and work with any dataset size. A feed forward pass of the network will consist of calculating the hidden layer output, running that through an activation function and then repeating that for the output layer.

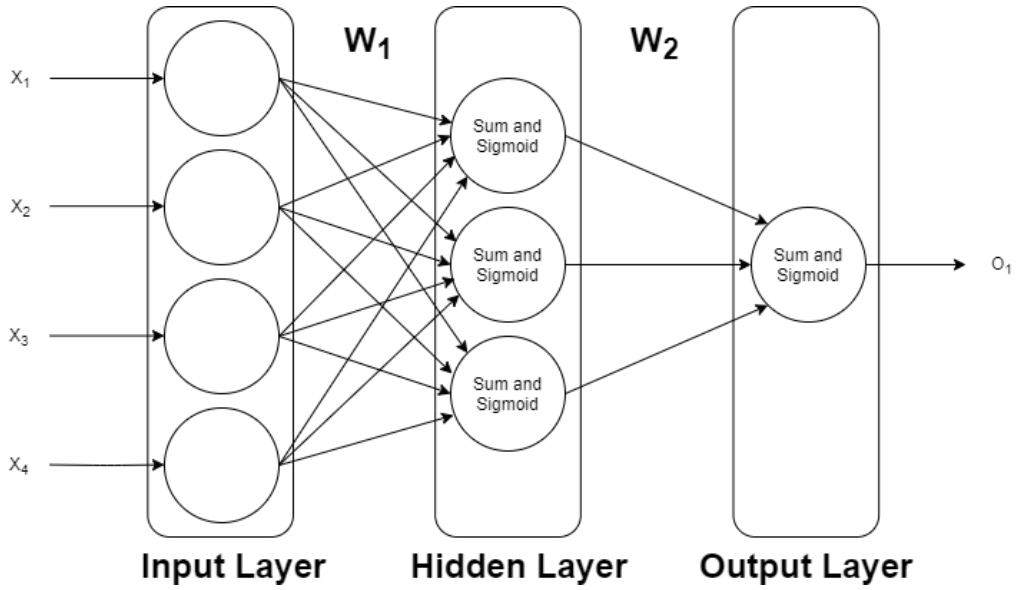

Figure 2. Diagram of implemented neural network with 4 inputs, a hidden layer, and 1 output.

The output of a layer is given by: $net = \sum weights * input$, this is then passed through an activation function. Activation functions are used to help map the output of the network to ensure that the network doesn't converge too quickly to 0 or to infinity. There are many different activation functions that can be used for neural networks but for this case I want continuous output values so I will be using a sigmoid function. The sigmoid of a value is given by: $sigmoid(x) = \frac{1}{1+e^{-x}}$. A sigmoid function tends to 1 as x tends to infinity and tends 0 as x tends to -infinity.

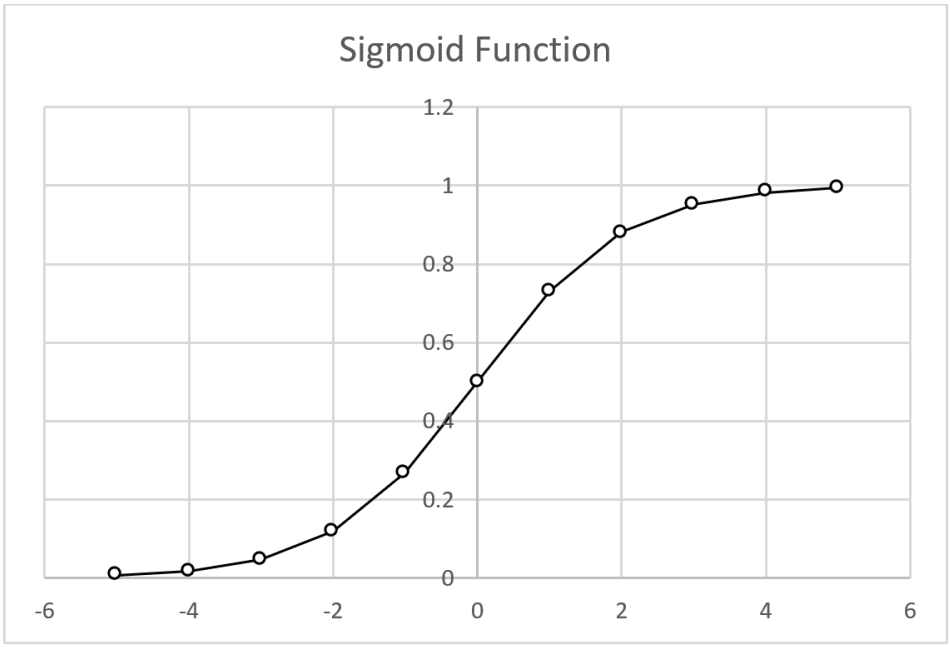

Figure 3. Sigmoid values between X-values of -5 and 5.

The next step is to implement the backpropagation algorithm. A series of calculations are used to find the error and the error gradient to then update the weights using the learning rate. The learning rate I have used is 0.1 for this network, this was very much trial and error to find an appropriate rate that didn't converge too quickly or take too long. Below is a short breakdown of the values needed to be calculated for a full backpropagation for a 2-layer neural network.

$\delta_3 = (t - o)$	Calculate error from output layer, t is the target, and o is output value.
$\delta_2 = {W_2}^T \times \delta_3$	Calculate error from hidden layer.
$d = net(1 - net)$	Use derivative of sigmoid, net is output from hidden layer
$\delta_2 = \delta_2 \times d$	Multiply by sigmoid derivative.
$\frac{\delta e}{\delta W_1} = \delta^2 \times X^T$	Calculate error gradient of output layer with respect to weights.
$\frac{\delta e}{\delta W_2} = \delta^3 \times Y^T$	Calculate error gradient of hidden layer with respect to weights.
$W_1 = W_1 + \alpha(\frac{\delta e}{\delta W_1})$	Update weights connecting input to hidden layer, α is the learning rate.
$W_2 = W_2 + \alpha(\frac{\delta e}{\delta W_2})$	Update weights connecting hidden to output layer.

(Nielsen, 2019)

SERIAL IMPLEMENTATION

For the implementation of my neural network, I created a simple matrix library to perform all of the operations needed for my matrix mathematics. I am using a structure to store all of the required information about the matrices. This includes the height and width of matrix and also an array which stores all of the values of the matrix in row-major order. All of the code for this can be found in the file "Matrix.cpp". The serial function for the matrix multiplication is called 'matMult' (line 41) and performs the multiplication for two matrices which are passed into the function as arguments. The function then returns the matrix product as a new matrix as shown in the pseudo code below.

1. New Matrix C with height A and width B
2. For i = 0 to height
3. For j = 0 to width
4. $C_{ij} = 0$
5. For k = 0 to width
6. $C_{ij} \mathrel{+}= A_{ik} \times B_{kj}$
7. Return C

The matrix multiplication is the most complex and time-consuming calculation within my matrix library. Due to the complexity of matrix multiplication being $O(n^3)$, as the number of inputs, hidden nodes or outputs increase the time taken to complete the calculation also increase exponentially. To help solve this issue I am going to implement a parallel method using CUDA.

PARALLEL IMPLEMENTATION

To increase the performance of my neural network I am going to implement my matrix multiplication function in parallel to allow it to run across many threads on a GPU. The method behind doing this is called tiling, this is the idea of splitting the 2 matrices into small submatrices and each thread loads one element of each matrix into shared memory. Then the threads collaboratively load a tile of a given size for both matrices and the dot products is performed on both of these submatrices. The tile size also determines how many threads are executing at once, this should be the same as the size of the product matrix. If the size of the matrix is 32x32 then the tile size should be set to 32 to allow each thread to execute on an individual element. The stride must also be set for the matrices as our elements are stored in row-major order. The stride simply refers to the width of the matrix and shows the size of the step needed to get to the next column (Hochberg, 2012, pp. 24).

This parallel implementation is achieved by invoking a kernel (line 127 in the parallel solution) and defining the number of blocks and grids. Due to the fact that the matrix dimensions may not be a multiple of the block size (line 16), it is necessary to take that grid size as an integer that is greater than or equal to: $\frac{Matrix\ Width}{Block\ Width}$ and $\frac{Matrix\ Height}{Block\ Height}$. Within the kernel shared memory is used to store the submatrices of A and B. Shared memory is used because it is almost 100x faster than global memory. In addition to this, the shared memory is allocated per thread block which means that each thread per block has access to the same shared memory (Harris, 2019). This means the total number of global accesses can be halved which results in the total traffic being reduced by $\frac{1}{n}$. Reducing the number of calls to global memory also increases the Compute to Global Memory Access (GCMA) ratio and increases the performance. The GCMA is the ratio between the number of floating-point operations by the number of memory accesses. This is a performance bottleneck as the calculation throughput is limited by the speed that the input data can be loaded from global memory. This means that the lower the GCMA value the lower the performance of the program and reducing the number of global memory accesses can increase the performance. Within my global kernel I perform 2 memory accesses, 1 floating point multiplication and 1 floating point addition. This is shown in the code below (line 85) and makes the GCMA ratio equal to 1.

```
for (int j = 0; j < BLOCK_SIZE; ++j)
       Cvalue += As[row][j] * Bs[j][col];
```

I have only implemented a parallel version of the matrix multiplication function; however, it would be possible to implement other parallel functions for most of the matrix calculations. This would possibly show a small increase in performance but not a noticeable difference due to other matrix calculations being a lot simpler and also not being used as often during the backpropagation algorithm.

EVALUATION

After implementing and training the neural network using the backpropagation algorithm, I then started running feedforward passes on the network to some results on how accurate the network was. For these tests I am training the network on each data set (150 different plants) and completing 1000 iterations with the network having 5 hidden nodes. As shown in figure 4 the average error percentage across 5 tests was 6.56%.

Target	Output	Error Percentage
0.5	0.558155	11.63%
1	1.012723	1.27%
1	0.875857	12.41%
0	0.012671	1.27%
0.5	0.531149	6.23%

Figure 4. Tests on neural network showing target, output and error percentage. Error percentage average is 6.56%.

For the dataset that I am using the input and output matrices are very small (input is 4x1 and output is 1x1), this means that I don't need a large number of hidden nodes to get good accuracy from the network. Due to this the matrices are very small and will not benefit from using a parallel function as this requires more copying of data and can even slow the process down. In this case I am going to I am going to increase the number of hidden nodes which will in turn increase the matrices size. Although this isn't necessary for this dataset to be able to get good accuracy, for other datasets with large amounts of inputs or outputs, a larger number of hidden nodes would also be needed. When choosing the appropriate tile size for the program I used NVIDIA Nsight to be able to determine to most appropriate tile size to get the best performance. In this case around a tile size of 48 had the best performance, which means a block has 48x48 threads.

Hidden Nodes	Serial Time (ms)	Parallel Time (ms)	Percentage Decrease	Speed Up
500,000	153	121	20.90%	1.26
1,000,000	296	173	41.50%	1.71
2,500,000	847	337	60.20%	2.51
5,000,000	1207	542	55.10%	2.23
7,500,000	2765	794	71.20%	3.48
10,000,000	2928	998	65.90%	2.93

Figure 5. Time taken to complete one iteration of the backpropagation algorithm.

Figure 5 shows the amount of time taken to complete one iteration of the backpropagation method using both the serial and parallel implementation of the matrix multiplication function. I ran each test 3 times and took the average for each number of hidden nodes. The parallel implementation reduces the algorithms time drastically. Using the averages of this data I have

used this formula: $S = \frac{Ts}{Tp}$ to calculate the speed up of the parallel algorithm compared to the serial. As figure 6 shows more clearly, the percentage decrease increases as the number of nodes increase. This means that the time saved is exponentially increasing until about 8,000,000 nodes. Furthermore, these time decreases show that this can be easily scaled to a much bigger dataset or network and be able to see the improvements of using a parallel function.

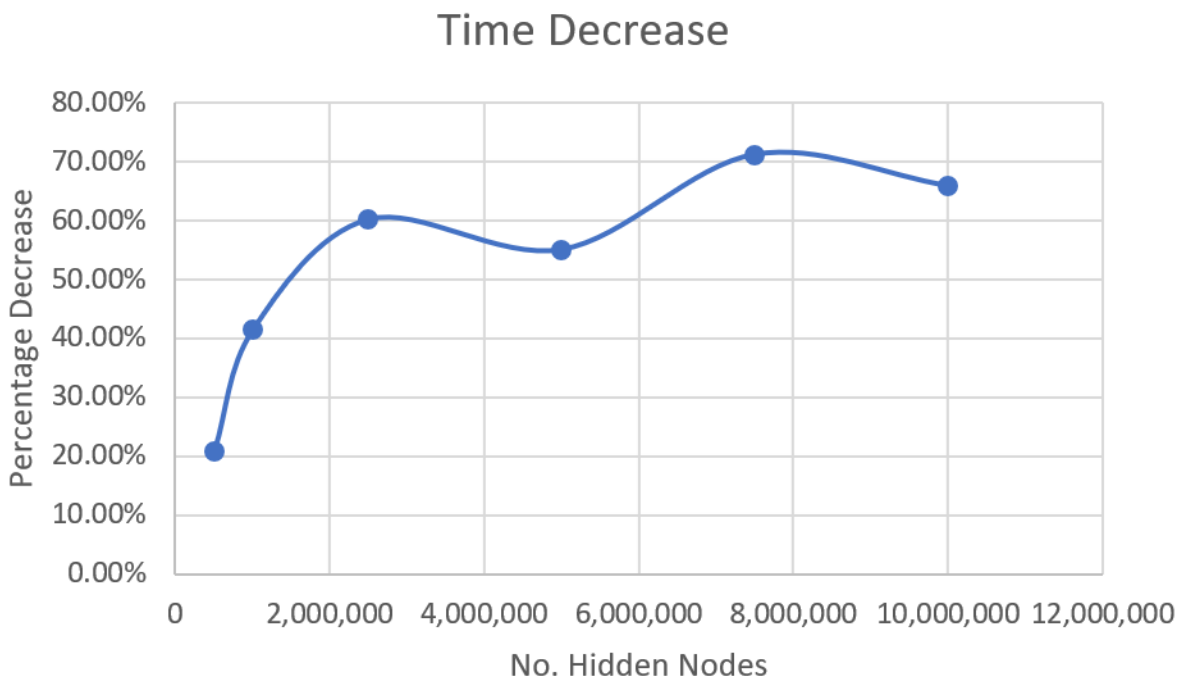

Figure 6. Time percentage decrease plotted against number of hidden nodes.

Tiling is a very effective method to use when implementing a parallel function for matrix multiplication as it allows the calculation to be split evenly across a large number of threads.

CONCLUSION

In conclusion, the results show that implementing a parallel function for the matrix multiplication operation significantly improves performance when using large matrices. This is very useful when the network has a large number of inputs, outputs or hidden nodes and the matrices being used would be very large. For example, when implementing a convolutional neural network. However, if the matrices are relatively small, using a parallel function can show very little improvement and sometimes even cause a decrease in performance.

To further improve the performance of this algorithm it would be possible to implement parallel functions for the entire matrix library. This would further speed up matrix calculations

for large matrices when doing any matrix operation. Another way of improving the performance would have been to take a different approach and use a method called data parallelization and works by splitting the training data into smaller batches. It would then train the network on these smaller batches in parallel and take the average values to be able to update that weights. This is then repeated for a number of repetitions to train the network (Huang, 2013).

Overall paralyzing the matrix multiplication function improves the backpropagation algorithm and should always be considered when implementing a neural network.

REFERENCES

Kirk, D. and Hwu, W. (2013). Programming massively parallel processors. Burlington, MA: Morgan Kaufmann Publishers.

Hebb, D. (2002). The organization of behavior. Mahwah, N.J.: L. Erlbaum Associates.

Le Cun, Y. (1988). A Theoretical Framework for Back-propagation. Proc. 1988 Connectionist Models Summer School, (June 17-26).

Nielsen, M. (2019). Neural Networks and Deep Learning. [online] Neuralnetworksanddeeplearning.com. Available at: http://neuralnetworksanddeeplearning.com/chap2.html [Accessed 21 Dec. 2019].

Hochberg, R. (2012). Matrix Multiplication with CUDA — A basic introduction to the CUDA programming model. [online] p.24. Available at: http://shodor.org/media/content/petascale/materials/UPModules/matrixMultiplication/moduleDocument.pdf [Accessed 22 Dec. 2019].

Harris, M. (2019). Using Shared Memory in CUDA C/C++. [online] NVIDIA Developer Blog. Available at: https://devblogs.nvidia.com/using-shared-memory-cuda-cc/ [Accessed 21 Dec. 2019].

Huang, Z. (2013). Accelerating recurrent neural network training via two stage classes and parallelization. [online] Available at: https://ieeexplore.ieee.org/document/6707751 [Accessed 23 Dec. 2019].