SOFT354 – Parallel Computation and Distributed Systems

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

Contents

[Introduction 2](#_Toc27847736)

[Implementation 3](#_Toc27847737)

[Serial Implementation 5](#_Toc27847738)

[Parallel Implementation 6](#_Toc27847739)

[Evaluation 6](#_Toc27847740)

[Conclusion 6](#_Toc27847741)

[References 6](#_Toc27847742)

# 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) and 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’s) 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 biological brains. Although the idea of neural networks has been around since the late 1940’s (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 call 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 camera’s and sensors and continuously completing a feed forward pass of a trained network to workout 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, 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 150 iris plants of 3 different types. 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 and 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 only can 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: . For multiplying two matrices together of size and the total number of calculations required is: , this then gives the complexity of . Parallelizing this algorithm does not change its complexity, however, it will reduce its total run time by a factor of (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.

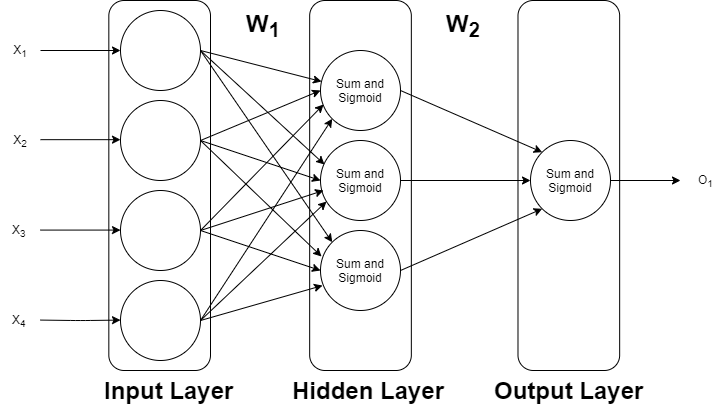
# Implementation

As previously stated, I will be using a dataset containing features about 150 different iris plants. 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 |

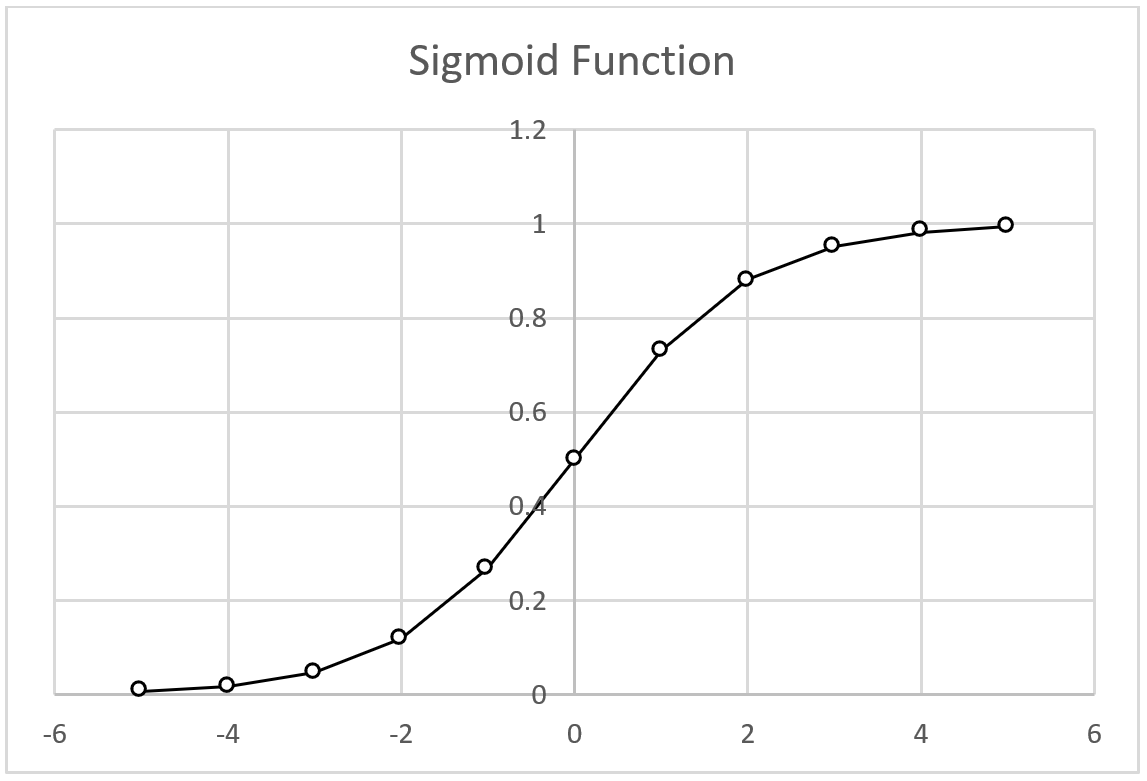
*Figure 1. Example snippet of the iris plant dataset.*

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: , 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: . A sigmoid function tends to 1 as tends to infinity and tends 0 as 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.

Calculate error from output layer, t is the target and o is output value.

Calculate error from hidden layer.

Use derivative of sigmoid, net is output from hidden layer

Multiply by sigmoid derivative.

Calculate error gradient of output layer with respect to weights.

Calculate error gradient of hidden layer with respect to weights.

Update weights connecting input to hidden layer, is the learning rate.

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 34) 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
5. For k = 0 to width
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 , 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. This is achieved by invoking a kernel (line 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 (defined at line ), it is necessary to take that grid size as an integer that is greater than or equal to: and .

* Shared memory is faster then global and local memory
  + Shared memory is allocated per thread block which means that each thread per block have access to the same shared memory.
    - This allows for
  + (Harris, 2019)
* How you moved from serial to parallel
  + What processes are still in serial?
    - What effect would if have if everything was in parallel
    - Why aren’t they in parallel?

# Evaluation

* Talk about tiling
  + How is it effective
  + Are there any better options out there
    - BACK UP WITH EVIDENCE
* The accuracy of the network

# Conclusion

* What did you find?
  + How much was the processes sped up?
* What could have been improved?
  + Could more things be paralyzed?

# 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].

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].