

# Parallelizing Neural Networks

Atharva Khandait  
160010020

Abhishek Kanthed  
150010031

Abhishek Singh  
183100006

Sumit Kumar  
13D100042

## I. INTRODUCTION

Neural networks are inherently parallel. We implemented neural network layers such as the Linear(fully connected), ReLU, Sigmoid and Cross entropy loss to harness the parallel computing power of a GPU. We put them together in a network and tested it on a simple dataset. We also implemented the layers on CPU for comparisons.

## II. CODE STRUCTURE

### A. Matrix Class

Most of the neural network boils down to tensor operations, for which we just need a matrix(2nd order tensor). We want the matrix to be available both on host(CPU) and the device(GPU) memory. Most operations will be performed on device but we need to initialize the matrix on host, just because its easier. Matrix class will manage memory allocation and make data transfer between host and device easier.

---

```
AllocateMemory()  
CopyHostToDevice()  
CopyDeviceToHost()  
cudaMalloc() called in 1)  
cudaMemcpy() called in 2) and 3) in the required  
direction .
```

---

This all is managed within the Matrix class so that we dont have to worry about it otherwise.

### B. Layers Classes

We implemented some commonly used neural network layers as classes. Each layer class has a Forward() and Backward() function where most of the work is done by CUDA kernels.

1) *Linear Layer*: In this layer, most of the computations are matrix multiplications. W is weights matrix, b is bias vector and A is input to this layer.

$$Z = WA + b$$

Backward: Update the parameters by gradient descent using the backpropagated error from next layer and pass on the error to previous layer. One kernel for the above equation, one for the below first equation and one for the second and third equations:

$$dA = \frac{dL}{dZ} \frac{dZ}{dA} = W^T \cdot dZ$$
$$dW = \frac{dL}{dZ} \frac{dZ}{dW} = \frac{1}{m} \cdot dZ \cdot A^T$$

$$db = \frac{dL}{dZ} \frac{dZ}{db} = \frac{1}{m} \sum_{i=0}^m dZ^{(i)}$$

Where  $dZ^{(i)}$  is  $i$ th column of  $dZ$  ( $i$ th input in a batch) and  $m$  is batch size. Most of computations in this layer are matrix multiplications. We didnt want race conditions because it slows the kernel down, so two threads cannot write to the same location. But, the threads can read from the same location simultaneously, because they wont change during computations. So, every thread calculates only a single element of the output matrix. We fix the number of threads per block to be a 16x16 grid and decide the number of blocks according to the size of the output matrix.

In the Backward() function, we need to do matrix multiplication with transpose of matrices. We dont have to transpose them, we can just manage the indices accordingly when multiplying. Example, for calculating  $W^T \cdot dZ$ , instead of taking a row of W and column of  $dZ$ , we take a column of W and a column of  $dZ$ . When we update the bias, we have to sum over all rows of  $dZ$ . We can do this using only one thread, or our other option is the atomicAdd() function. Both are undesirable and slow down the kernel as a lot of the threads need to wait. But, we can't avoid it here. In this case, we didnt use atomicAdd() and just used one thread to loop over a row.

2) *ReLU Layer*: Both activation layers we are implementing are very simple with element-wise operations.

$$ReLU(x) = \max(0, x)$$

3) *Sigmoid Layer*: Sigmoid layer should compute sigmoid function for every matrix element.

$$\sigma(x) = \frac{e^x}{1 + e^x}$$

Backward: According to the chain rule ( $dA$  is the error from next layer):

$$dZ = \frac{dL}{d\sigma(x)} \frac{d\sigma(x)}{dx} = dA \cdot \sigma(x) \cdot (1\sigma(x))$$

Where  $L$  is a cost function and  $dL/\sigma(x)$  is the error introduced by sigmoid layer.

For both the activation layers, first we calculate index for current thread, then we check if this index is within matrix bounds. Every CUDA thread compute a single output. We have fixed the number of threads per block to 64 and decide the number of blocks accordingly. Both the activation functions

are element-wise so each thread just computes for one element of the output.

### C. Cross Entropy loss

We decided to use binary cross-entropy as cost function that returns gradient accordingly to network predictions and our target values. Binary cross-entropy is defined with following equation:

$$BCE = \frac{1}{m} \sum_{i=0}^m (y \log(\hat{y}) + (1-y) \log(1-\hat{y}))$$

And by calculating its derivative we compute gradient as:

$$\nabla BCE = \left( \frac{y}{\hat{y}} \frac{1-y}{1-\hat{y}} \right)$$

Where by  $\hat{y}$  we denote predicted values and by  $y$  the ground truth values. We use a built-in math function `logf` here.

### D. FFN Class

Feed forward network, this is the class where we will put all layers together to construct and train a network. We add new layers using `Add()` function. The pointers to all the layer objects are stored in a vector in this class. The most important functions here are `Forward()` and `Backward()` which will just call the `Forward()` and `Backward()` functions of all the layers sequentially. A `Train()` function has been implemented in this class which takes an instance of the `Dataset` class, learning rate, number of epochs and a boolean which tells whether to run the CPU only version of all functions.

We have implemented the CPU only functions for all layers as well.

## III. PERFORMANCE ANALYSIS

### A. Dataset

We create a simple dataset for training a model. We sample random 2D points and label them 0 if they lie in the 1st or 3rd quadrant and label them 1 if they lie in the 2nd or 4th quadrant. We train our model on this binary classification task. Our test loss drop to 0.106 and test accuracy reaches 1 (numbers may vary a little during each run).

### B. Profiling

We profile the CPU only code using `Callgrind` (`Valgrind`) and check the percent of total time the code spends in functions which can be parallelized.

Total percent of time serial code spends in functions which are almost completely parallelizable:

Linear: Backward: 48.67 Forward: 33.26 Total: 81.93

ReLU: Backward: 7.09 Forward: 7.06 Total: 14.15

Sigmoid: Backward: 1.03 Forward: 0.6 Total: 1.63

Grand total: **98.01%** of the code is parallelizable.

ReLU takes a lot longer than Sigmoid even when they basically both are simple element-wise operations. This is probably because of lot of if statements in ReLU.

### C. Theoretical limit using Amdahls Law

Amdahl's law can be formulated in the following way:

$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

where

- $S_{latency}$  is the theoretical speedup of the execution of the whole task
- $s$  is the speedup of the part of the task that benefits from improved system resources
- $p$  is the proportion of execution time that the part benefiting from improved resources originally occupied

Even if this is for processors and not GPU cores, we think we can at least get an upper limit of how much speedup we can get compared to CPU only code.

Using  $p = 0.9801$  and  $s = 768$  (number of CUDA cores on GTX 1050ti), Max speedup achievable is 45.45 times faster than the serial code.

We get **32.09** times faster than the serial code.

## IV. CONCLUSION

As we saw in the profiling, most of the computation of neural networks can be parallelized. Parallelizing neural networks can save us a huge amount of time and enable us to work with bigger models and larger datasets.

Hence, in recent years we have seen a huge growth in the field of deep learning as the GPUs have got faster and faster and we can also say that the GPUs have become better and better due to the huge demand for artificial intelligence hardware.

## REFERENCES

- [1] Kyoung-Su Oh and Keechul Jung, *GPU implementation of neural networks*.
- [2] Altaf Ahmad Huqqani, Erich Schikuta, Sichen Ye and Peng Chen, *Multi-core and GPU Parallelization of Neural Networks for Face Recognition*.
- [3] Vishakh Hegde and Sheema Usmani, *Parallel and Distributed Deep Learning*.
- [4] Ricardo Brito and Simon Fong, *GPU-enabled back-propagation artificial neural network for digit recognition in parallel*.