Accuracy and Performance of Parallel Neural Network as Number of Processes Scales

Noah Carver {ncarver1@umbc.edu}
Computer Science & Electrical Engineering Department, UMBC

Abstract— **It is common practice to batch-parallelize neural network training. To study potential detriments of this on accuracy, a CNN was created and tested on the MNIST Handwritten Digits dataset. Surprisingly, the accuracy of the network increased as it scaled, likely due to increased accuracy when batch-updating parameters.**

I. Introduction

When training a Neural Network, a set of labeled data is used to train and verify the training. A common technique to make best use of training data and still adequately test the resulting network is k-fold Cross-validation. This technique runs training multiple times, each time, selecting a different testing set from amongst the training data. As one can imagine, this technique is time-consuming.
In fact, much of deep learning can be time-consuming, so many methods of parallelization have been developed for it. Methods using CUDA architectures to parallelize the basic tensor operations[1] are particularly popular.
Discussed here will be parallelization at the CPU Level in the style of Batch Parallelization.

II. Background

Before considering the changes made to parallelize, It is important to look at the original algorithm seen in Algorithm 1.
The data is first fed through the Neural network, and a result is obtained. That result is compared to the target result to determine an output gradient, which is then back propagated to calculate gradients for each layer. Then the model is updated using gradient descent.

[1]Unfortunately, I do not have a computer with CUDA cores.

Algorithm 1: Sequential CNN Learning

```
1 forall input,target ∈ Data Points do
2     Feed Forward($input$) → $output$, $layerInputs$
3     Compute Loss($output$, $target$) → $loss$, $outputGradient$
4     Back Propagate to update model gradients ($layerInputs$, $outputGradient$)
5     Update model using Stochastic Gradient Descent.
```

III. Method

By breaking the training data into 'mini-batches', with the size of each being equivalent to the number of processes available, we are able to perform updates to the model in bursts, potentially resulting in almost linear speedup!
However, note that the update to the model gradients isn't protected in any way. This introduces a race condition where if one process begins that step while another process is on it, the second process's update may be overwritten.

Algorithm 2: Parallel CNN Learning

```
1 forall batch ∈ MiniBatches do
2     Parallel forall input,target ∈ batch do
3         Feed Forward($input$) → $output$, $layerInputs$
4         Compute Loss($output$, $target$) → $loss$, $outputGradient$
5         Back Propagate to update model gradients ($layerInputs$, $outputGradient$)
6     Update model using Stochastic Gradient Descent.
```

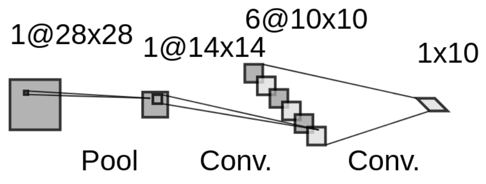


Fig. 1. Convolutional Neural Network Used in Testing

However, this race condition is ignored, as the model should react and converge anyway. These sort of 'mistakes' are even introduced by Stochastic Gradient Descent as a way of avoiding local minima.

Code was written in C++ and parallelized using OpenMP. It was trained and tested using the MNIST Handwritten Digits Database. [2] Tests were performed on my personal computer (Ryzen5 2600x, Radeon RX570[2]).

IV. Results

Num Threads	Accuracy(%)	Time	Speedup (approx.)
1	26.0	69.3580	-
2	68.4	41.5767	1.7
3	74.8	35.7616	1.9
4	77.8	31.8090	2.2
5	79.8	28.3113	2.4
6	80.8	26.2636	2.6
7	83.4	27.8849	2.5
8	86.2	27.3898	2.5
9	87.8	26.3562	2.6
10	88.8	24.2820	2.8
11	89.6	24.2981	2.8
12	91.8	24.7290	2.8

TABLE I

Test Results

V. Discussion

The speedup is as expected. The Ryzen 2600x has 6 cores and 12 threads, so the fall off at 7-8 threads reflects that and the linear speedup is exactly as expected from 2-6 threads, as seen in figure 2. The increase in accuracy was surprising. Assuming no major changes to the algorithm, the accuracy should remain fairly constant. However,

[2]As mentioned, the GPU was not used for this experiment

Fig. 2. Speedup as NumThreads Increases

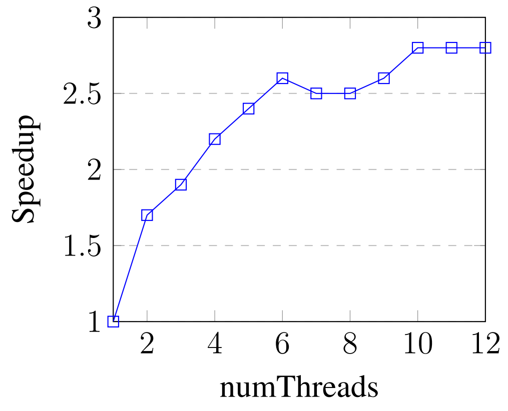
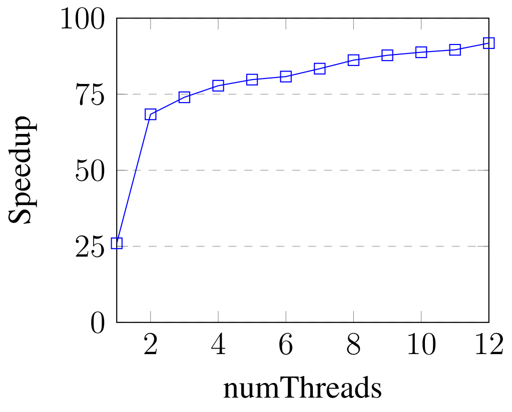


Fig. 3. Accuracy as NumThreads Increases



as seen in figure 3, It instead steadily increases, even past the 6-core inflection point! To explain this behavior, consider again Algorithm 2, specifically line 6, and remember that this algorithm accumulates all of the updates from the mini-batch and then performs one, more accurate update, instead of up to 12 less accurate updates. This 'zeroing in' allows the network to make less, more accurate steps!

It appears that not only are Neural Networks exceedingly parallel, their training can actually benefit greatly from being run in parallel!

References

[1] Freddie Åström and Rasit Koker. A parallel neural network approach to prediction of parkinsons disease. *Expert systems with applications*, 38(10):12470–12474, 2011.

[2] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

[3] Y. Takefuji. *Neural Network Parallel Computing*. The Springer International Series in Engineering and Computer Science. Springer US, 2012.