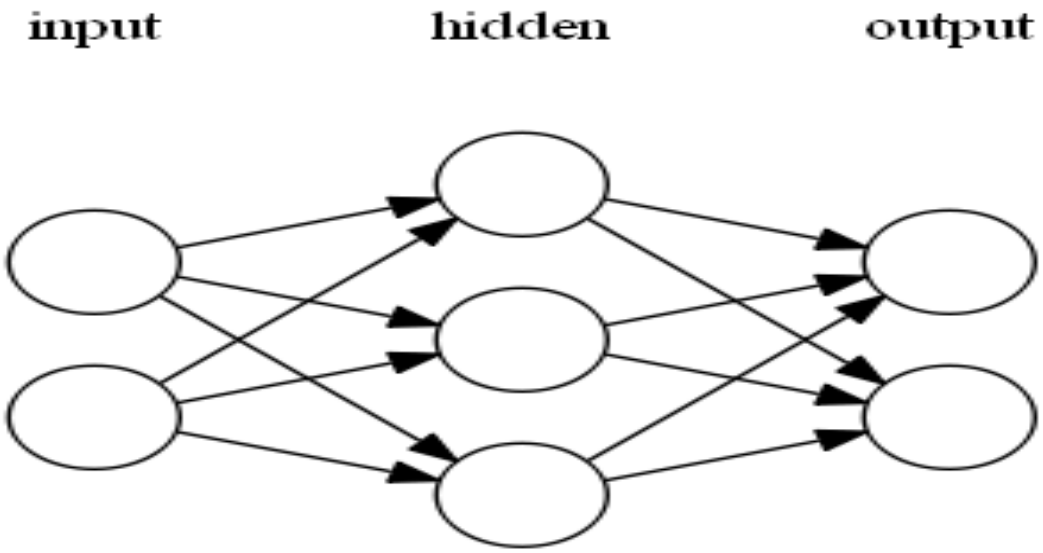
Parallelizing Open Source Back-propagation code

Bruce Darcy

This project has involved me working with open source neural networks code, to parallelize the training of a neural network. The source code is written in C, and comes from the GitHub repository https://github.com/codeplea/genann. The project includes an example on IRS data to train the network that I used as an example for testing my optimizations. Attached to this project is the source code with my parallelizations done in OpenMP.

In the Genann code, there is a function that performs one iteration of training with one example. Back propagation involves two phases, the first where the network is run forward to determine the current output for an example, then the second where error is back-propagated through the network, and the weights of the connections between neurons/nodes are updated based on how much they contributed to the error. The code supports up to 4 hidden layers each containing as many nodes as you want. Here is a visualization of a simple neural network with 2 inputs, 3 hidden nodes, and 2 output nodes from the project repository.



My first parallelization was to use Open MP to parallelize steps in back-propagation that were taking the most time. I worked with a network of 4 hidden layers, and node numbers in each layer ranging from 1000 to 2000, because with smaller networks the fork join overhead outweighs the parallelism because you aren't doing very many iterations in each step. For the IRS data set, networks this big are completely overkill, but my focus is more to demonstrate how this parallelization would help if you had a problem that did require extremely big neural networks.

To run the example I used, run the example4 executable after compiling the project with the make file. The main part of the code that is used in everything comes from the genann.c file. The example is only performing one iteration of training for a big network, so ignore the output that the accuracy is low; the network is just not getting trained very much.

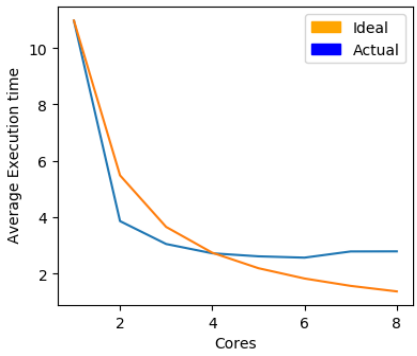
I ran the code through the callgrind tool from Valgrind, and first found that calculating the outputs of the hidden layers in forward propagation was taking a lot of time. That step consists of triply nested loops, the outer that iterates over each hidden layer, the middle that iterates over each node, and the

inner which sums the weights of all inputs coming into a certain node. Because each layer's input is dependent on the layer before its output, the outer loop is inherently not parallelizable. But the inner two are completely. The code however needed to be rewritten to allow me to parallelize it, because it was individually incrementing each pointer to the data it uses every iteration. I rewrote the inner loop to make it parallelizable and ran it, but the overhead was way too high and the program got slower. Then I rewrote both loops, and managed to get some speedup for large problem sizes, but I wasn't getting very much speedup, so I looked further into callgrind and found two more phases where most of the computation was being done.

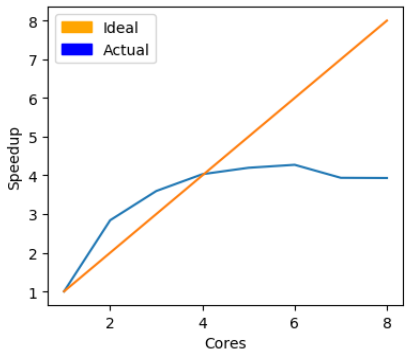
One was calculating the deltas, the amount each node was contributing to the overall error in back propagation. I first used an Open MP parallel for with a reduce, adding up the deltas for the inner most loop of the phase, where all the deltas get added up for a given node to find how much they total to. It ran faster, and I tried making this parallel over all the loops, but for some reason it didn't run as fast. I suspect the reduce is the cause, and that it is somehow adding up all these values in a way that is more efficient enough that it makes up for spawning a lot more threads since the fork join is deeper in the loop structure.

The other phase that takes a lot of time is updating all the weights going into a node based on the error delta and the learning rate. That consists of two loops, iterating over each hidden node, and each weight for that node. I rewrote and parallelized these loops using an OMP parallel for on the outer loop.

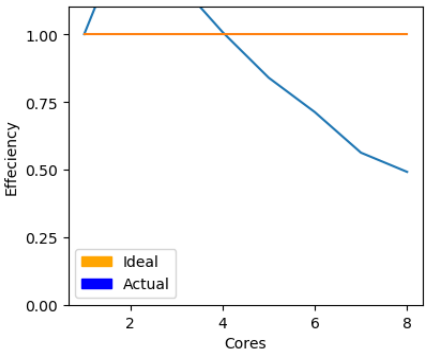
Here are the results I got from running the network with 4 hidden layers of 1500 nodes each on 1 run a though all 150 samples in the data-set. For each possible number of threads 1 through 8, the training of the network run 7 times and the average of all 7 was taken and then added to the graph.



1	10.966025142857143
2	3.8616132857142857
3	3.050185571428572
4	2.721011428571429
5	2.6135938571428565
6	2.5665732857142856
7	2.7858001428571435
8	2.789366714285715



1	1.0
2	2.8397522826599526
3	3.5951993365836894
4	4.030128292630681
5	4.195764813606138
6	4.272632776120111
7	3.9364005242710203
8	3.9313673195764296



1	1.0
2	1.4198761413299763
3	1.1983997788612297
4	1.0075320731576702
5	0.8391529627212275
6	0.7121054626866852
7	0.5623429320387172
8	0.4914209149470537

I have no idea why the speedup from 1 threads to 2 threads is so large, it theoretically shouldn't be possible for it to be bigger than 2. This data was produced by setting the environment variable OMP_NUM_THREADS from to 1 then 2. I got a similar speedup difference with an untouched version of this neural networks code to my parallelized code running two threads. All files are compiled using the -O3 flag. I tried to research why this might be happening, but didn't seem to find much. One theory I have, is that the OMP reduce I used for one of the loops is allowing for some other optimization once I have more than one thread. Perhaps it is vectorizing the code better, or exploiting caches and memory channels better.

Other than that strange occurrence, the speedup after 2 threads looks pretty normal, with the speedup eventually approaching an asymptote as Amdahl's law says it should. We also see a slight jump down since I ran this program on a 6 core machine, the Bugatti machine in the CSB. The speedup peaks around being just over 4x faster for 6 cores which is a bit better than our average project in Open MP this semester.

Back-propagation, as you already know, is a highly parallelizable problem, and I wanted to take this code and add to it CUDA parallelization or MPI. I decided not to go with MPI, since the dependency between layers of the neural networks would mean that you would need to have massive network layer sizes to take advantage of it. Since there are two for loops looping over each neuron and each weight for it, it is possible to do that in many chunks by splitting the weights summing into small parts, and doing to for each neuron. I also learned about bunching training data to train a network on multiple samples at once from the old PA6 project from the last semester this class was offered.

I ran into a pretty big problem as I was trying to add CUDA kernels to the project. The source code for Genann has some parts that don't compile correctly on the NVCC compiler, meaning that a lot of code would have to be completely rewritten to allow me to add it to the project. I've looked into ways to allow Genann.c to be compiled with GCC and another file to contain functions for it to call that would make CUDA calls, but I wasn't able to get anything working.

If I were to write it though, the ideal way to do it would be to cudamalloc the Genann struct on the GPU on the initialization phase, then Genann run and train would both be device functions. If I were able to get genann.c to compile with NVCC, it wouldn't be too hard from there to implement a coarse grain parallelization of the problem by simply using pointers to the GPU data. If one were to continue optimizing this code, moving onto a GPU would be the best next step.

One interesting thing I encountered in doing this project was examining how the activation functions affected the performance of the code. In the forward propagation the activation the sigmoid activation function was taking up a lot of computation time. I was able to reduce that by calculating the activation for each node in parallel. The original developers even implemented an option to use a 4KB cache to make the sigmoid activation function faster. I've learned in the past the ReLu was able to give huge performance increases when people started using it for neural nets, and that makes sense because the activation function limits the amount of parallelism you can have during that phase to the number of nodes in the layer that the activation function is being called for, so having one that's a lot faster can speed things up quite a bit. This is particularly important for smaller network structures where you might be limited by 5 nodes in a hidden layer rather than 1500.

Another thing I would like to look more into is why the OMP reduce seems to bring so much performance gain, it's really something that surprised me when I ran into it in this project.

Sources

Wrinkle, Lewis Van. “Codeplea/Genann.” *GitHub*, 5 Sept. 2018, github.com/codeplea/genann.