Neural Net memory (CUDA) management

# Overview

The idea here is to map out what memory will be moved on and off the device. Let’s start with a list of the arrays/matrices.

A note on the sizes. The weights multiply nodes and the biases are added. So, I am using affine transforms similar to the way translational information is added to a 3-D pipeline. This generally adds and extra element to each of the arrays, which in turns adds rows and columns to the matrices.

# Training set Images

These are in 28 by 28 matrices flattened to a 784 +1 length array. Each has an associated label telling the intended value of the digit shown in the image. The image label is used to create a vector of length 10 whose elements are all zero except for the digit matching the label. That will be 1. Note that technically, you don’t really need that array. You could just do your least squares fit using the label.

If I have 60,000 training sets and the images use one byte per pixel, they represent 60,000\*785 bytes = 48 Meg. I think I currently use double to hold the 10 length vector. That still makes 8 \*10 \* 60,000. That is still nearly 5 Meg. I could cut that down to simply 60,000 bytes.

I want the training set images to get loaded to the device one time only. I also do not need to copy them back to the host.

# Weight and bias matrix

There is a single set of weights and biases matrices (flattened) that holds the currently trained network. For my case of 784 +1 input array, two hidden layers of length 16 +1 each and the final output layer of size 10 you get a first layer matrix of 17 by 785, a second layer matrix of 17 by 17, and a third layer matrix of 17 by 10. This gives a total of 13,804 numbers, if they are doubles, that gives 110,432 bytes per set. There is only one weights and bias set of matrices.

# Gradient matrix

The gradient matrix is the same size as the weight and bias matrix. You need one for each thread running in parallel. If you are running 1000 threads, you need 100 Meg to hold them. In your CPU solution, you are using plus equals on the same gradient matrix over different training images. Here, each thread will have its own. So, it will get an absolute value. That means that you should not have to initialize them. You do however, need to reduce them. If you are cutting a training set of 60,000 images into sets of 1000, you will do this 60 times. It may be fine to do 2000 threads, which would cut it down to 30 trips. So, expect to make memory for 2000 gradient matrices sets and move them to the device once. I will then just keep reusing them. After each 2000, I’ll run a reduction and += them into a single weighting matrix. Note that copying, adding, or += one matrix into another can be done in parallel.

# Layer nodes

The layer nodes (excluding the first layer, which is always the input image) need to be updated before its corresponding gradient matrices are calculated. You do this using the current weights and bias matrices. If you do all of them at once, you need 60,000\*(17+17+10) = 2,640,000 numbers. If they are doubles, you need 21 Meg.

# Conclusions

Make one time device memory for (all of them flattened):

* The updated layer nodes
* 2000 gradient matrices sets
* Entire training set
* One weights and biases matrices set
* A single reduced gradient set.

Break the training set into 30 loops. The kernels will be:

Update all 60,000 layer nodes.

For 30 loops:

Calculate next 2000 gradient matrices sets

Reduce them to a single gradient set and += with the set that will hold the gradient results

When all 30 loops are done, add the resulting gradient to the existing weight and biases matrix.

Repeat.