Deep Boltzmann Machines: Edgar Chen, Taichi Akiyama

Summary: We are going to implement Deep Boltzmann Machines with Parallel Tampering on a GPU.

Background: Deep Boltzmann Machines http://www.utstat.toronto.edu/~rsalakhu/papers/dbm.pdf are powerful generative models which are capable of doing unsupervised learning on large datasets. However, due to their expressivity, the training process is very much compute-bound. Our goal is to implement a Deep Boltzmann Machine training and inference program using modern-day GPUs. Boltzmann Machines can do very interesting things, and they don't require labels on the data to learn about the data, making them useful in settings where people have collected a lot of data but don't have the resources to label all of the data. They can learn structure about the data, which can then be used to classify similar datapoints or even generate new or similar data. They can learn, for instance, what a certain alphabet system's characters "look like" and generate plausible examples that would fit into the alphabet, or complete part of a character given the other half, which can be useful in filling in missing parts of data.

Challenge: http://15418.courses.cs.cmu.edu/fall2016/lecture/dnn/slide_048 is actually a pretty good example, since DBMs operate somewhat like neural networks in general (but use a lot more probabilistic sampling). There is a lot of memory movement required, and the parameters need to synchronize at each step. It's possible that this won't be challenging enough and we might want to do something like distribute the computation on latedays, but we think this is plenty challenging, given that the base code for a good DBM implementation is much more complicated than a neural network.

Resources: We will probably use a linear algebra library like Eigen, and maybe borrow some of CUDNN, a CUDA neural networks computation library, but probably not fully supplied neural network library like TensorFlow. http://www.utstat.toronto.edu/~rsalakhu/papers/dbm.pdf is the paper we will be using. We could benefit from access to a modern-day GPU, but we can probably just borrow one of the GHC machines for that.

Goals and Deliverables: Create a working DBM implementation with significant speedup over a naïve CPU version

Show off the different things that unsupervised learning can do: we'll probably be doing it on characters so we can do things like generate new characters that "look like" characters in the dataset, and generate closest representation in the dataset for a specified input.

Platform Choice: Obviously a CUDA program with a 1080

Schedule: Week of 11/7: Do research on how a Deep Boltzmann Machine operates and begin creating the sequential version of the machine.

Week of 11/14: Create working parallel version of program using CUDA and python integration. The algorithm can be divided into, roughly speaking, two halves: variational inference and Markov-chain Monte Carlo sampling to calculate the updates, and each of these we can select one of many algorithms (for MCMC sampling we might try parallel/simulated tempering, etc). We would like to have at least one, if not both, implemented in CUDA.

Week of 11/21: Do tuning on the parallel section to make it as fast as possible, and try out a selection of different algorithms on each of the two general halves of the algorithm to see what gives a good tradeoff between performance (training time) and accuracy (test set error).

Week of 11/28: Put together the deliverables (takes a nontrivial amount of time and coding), poster, and website. The deliverables will be basically sampling our generated model in different ways to produce character completions or character generation, showing that our model has learned the structure and form of the alphabet.

Milestones:

For our 75% goal, we plan on getting the parallelized version of the machine implemented with CUDA. Currently, we are writing out sequential code with Python, so we had to research how to implement CUDA programming in python. In our research, we found that we can use numbapro to run vectorized python code on the 1080. We also found that benchmarking will be very easy with the built-in profiler (or use nvprof) to see exactly in where in our code and in which device (cpu or gpu) takes the most time. There are a lot of resources available on how to use cuda on python, most of which is found on youtube (CUDAcast #10 and onward).

For our 100% goal, we would need to tune our CUDA implementation to achieve maximum speed-up. We can do this in several ways, including increasing/decreasing gridsize/blocksize each cuda function, network size, learning rate, etc.

For our 125% goal, we could implement different training algorithms and check their quality/performance against the one we wrote. Such algorithms include the Metropolis-Hastings algorithm, Gibbs sampling, Slice sampling, etc. All of these algorithms can be parallelized in some way, so we can implement crude versions of these algorithms to compare quality/speed.