Parallel Deep Boltzmann Machines

Edgar Chen, Taichi Akiyama

Summary

We implemented Deep Boltzmann Machines using OpenMPI on unix.andrew.cmu.edu. Originally we were going to use the Gates cluster machines and implement on GPU to compare and contrast. However, we came across many problems trying to get CUDA to work on python.

Background

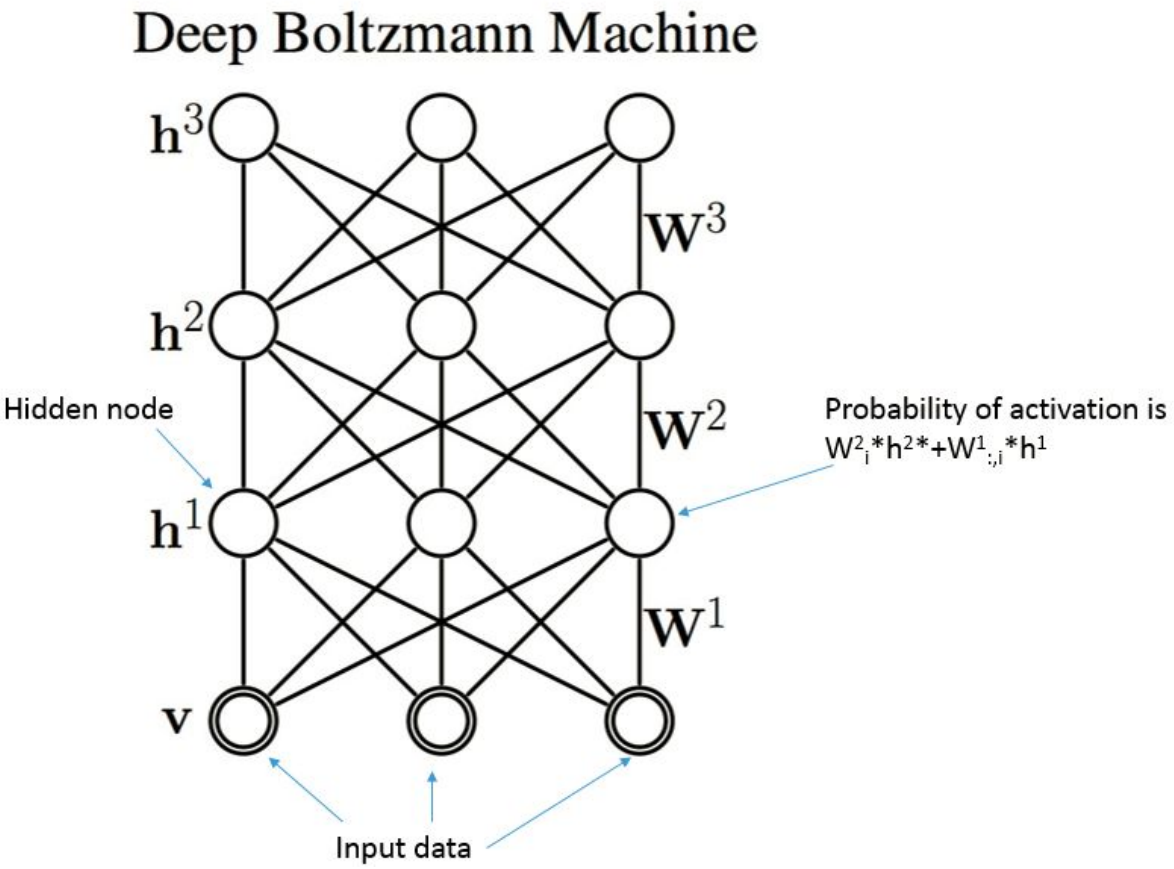
Deep Boltzmann Machines are a generative unsupervised learning model, which means that they take unlabeled input and learn structure about the input so they can generate samples which resemble the original input. They're also well known to be incredibly slow and difficult to train, and they are very sensitive to the amount of batching (data-parallelism), learning rate (a constant which affects how fast we make changes to the network) and other parameters used to train the network.

The Deep Boltzmann Machine carries two sets of state: One is the model parameters, ie the weight matrices and bias vectors. The weight matrices scale quadratically with the model size. There are multiple hidden layers of units which make up the network, and the probabilities of each hidden node inside the layer being activated is determined by the result of multiplying some of the matrices together and adding the bias. The other is the persistent Markov chains which we sample as part of doing the negative phase of training. These chains contain binary information about the state of the hidden units and data inside the chain, and we sample the chain using

Gibbs Sampling, by sampling each layer individually. Training proceeds as follows: we first use the network parameters and the data to generate the updates for the positive phase. Then we sample the persistent Markov chains k steps for the negative phase, then we add up the updates and modify the model parameters. Basically, we're pulling the network in the direction of the data, then away from the most probable points in the network, to keep a healthy average between the correct points and points likely to happen in our machine.

The first phase contains a lot of matrix multiplication, since we have to approximate the probabilities of the hidden layers when the data is already input. We have to multiply together decently large (728x400 x 400x400) matrices many times for convergence. The second phase consists of generating random points, which means we have to compute the probabilities by multiplying the matrices using the data that's currently stored in each chain, then generating random points using those probabilities. The more chains we use, the more accurate the final result will be. Matrix multiplication and random number generation can both benefit vastly from parallelization.

The workload is also data-parallel, and quite easily so, since the training can happen without needing the fully updated weights and biases. However, waiting too long to synchronize them can cause the network to converge suboptimally.



Approach

Although we tried to implement this machine with CUDA, we ran into many issues getting CUDA to work with python. In our initial research, we decided to use Anaconda, a python library package used for data analysis and machine learning, in order to parallelize our code. However, when we attempted to install the accelerate package on our local machine that ran a GTX 970m, our simple test code that we wrote did not work. For some reason, even after installing the numbapro package, python refused to import it. After we decided this would not work, we found a CUDA Matrix library written by Volodymyr Mnih which we believed we could use to parallelize our matrix multiplication portions of our code. However, installing on our local machine was

unsuccessful due to several environment/linking issues, so we opted to install this library on the gates machines. However, although installing on remote machines were successful, when we ran the built-in benchmarks provided by the library, several of the tests failed, most likely due to some floating point error. Although we have a cuda implementation of our code written, we could not test it at all.

Due to these issues, we used Python, NumPy, and OpenMPI on the unix.andrew.cmu.edu machines. We mapped each data point, or sometimes batches of data points to processes, computed the positive and negative phases on the processes, then used OpenMPI to communicate the updates to the weights and biases back together. Originally, our approach communicated the chains as well, but it turned out to be unnecessary; in fact, simulating more chains by simulating separate chains on each dataset was more helpful in terms of helping the algorithm converge to a good point. At one point, we realized that training was diverging when expanding large networks to many cores. We later realized that this was actually an artifact of not using enough chains, as we didn't properly seed the random number generator so all of the processes were using the same chains, which caused training to fail. We wrote the serial implementation ourselves, which took a very long time to write up and debug. Ultimately the project ended up being embarrassingly parallel, because the problem was inherently data-parallel and too easy to parallelize.

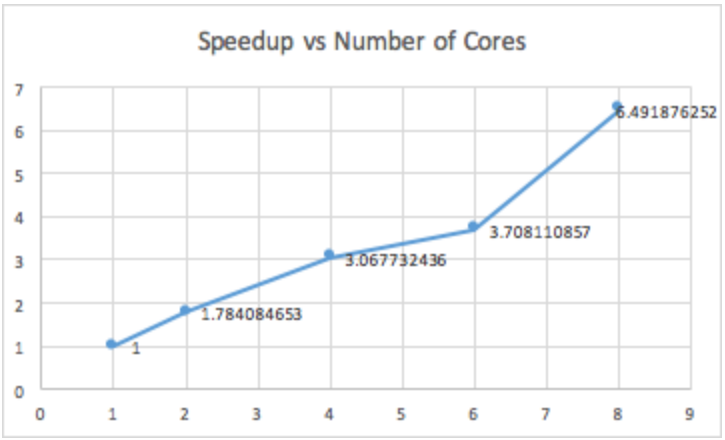
One interesting tradeoff that we were able to make while parallelizing was that instead of sampling from, say 100 chains per process, we could sample from 100 chains total with little penalty to performance. However, reducing the number of chains

in a single-core run caused the generalization power of the network to drop significantly. We conclude then that the chains don't need to be sampled too often, but there needs to be a sufficient number of chains for the algorithm to converge. Thus, the sequential version could be improved by sampling a subset of the chains for each training point instead of all of the chains.

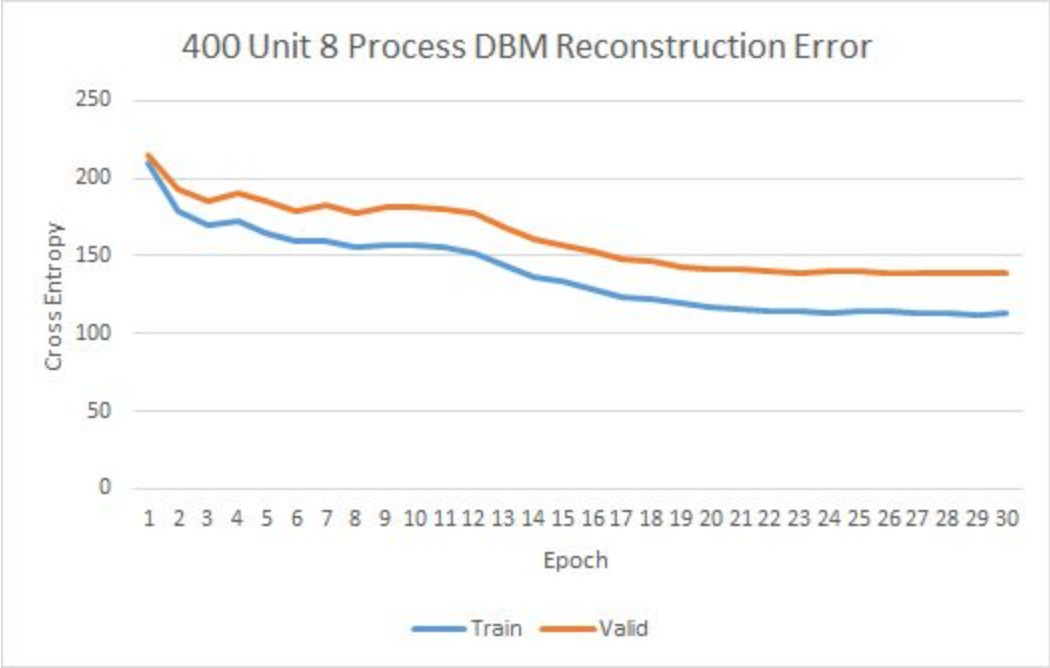
Results

We used the wall-clock time of training a single epoch (3000 training examples) to measure our efficiency. Training is the most important part to measure; we didn't measure testing for error after each iteration for example. The inputs were taken from MNIST, a handwriting dataset with 28x28 images of digits 0-9.

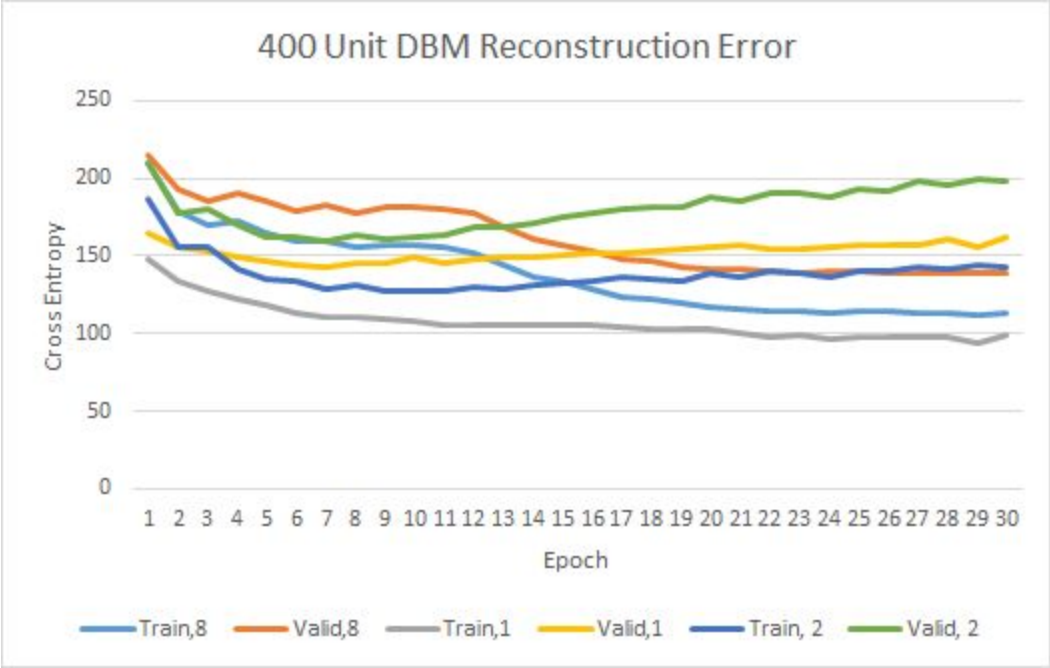
The speedup we achieved were as follows (100x100 hidden units):



We observe almost linear speedup as we increase the number of cores, except for the anomaly at eight cores. We believe this is to be the case because we sync less often as we increased the number of cores. We experimented with syncing less often, but this impacted the performance of the network.

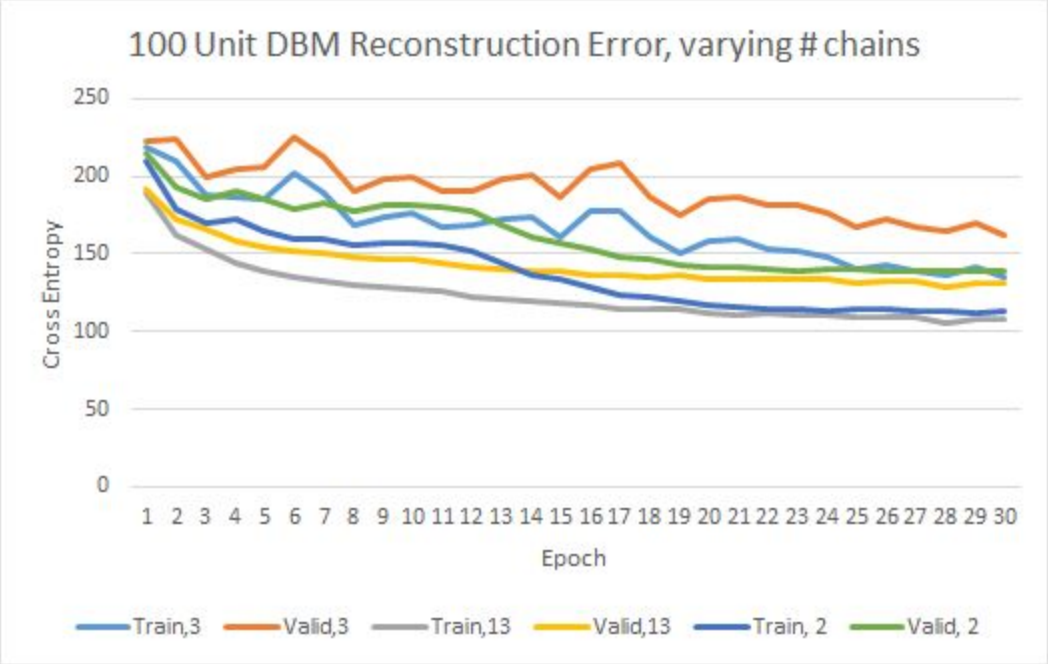


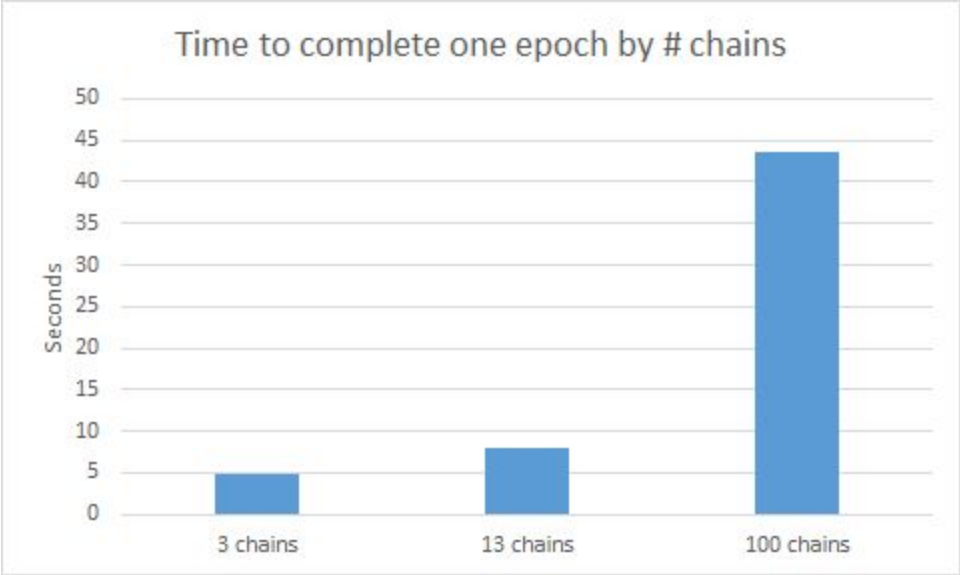
We also want to compare the performances of networks trained with different numbers of cores. In the next figure, we can see that the 8 core version did fairly well compared to the 1 core and 2 core versions. It seems that the smaller core counts are more vulnerable to overfitting (where the training error goes down but the validation error starts to go up), possibly because they are simulating less chains over a long period of time, and the negative phase is the regularizing mechanism in the algorithm to prevent overfitting.



On the 400 unit case, we noticed that sampling the chains takes about 5 times longer than the positive phase (0.11s/sample versus 0.02s/sample at 100 chains). And synchronization, which only took 0.005s/sample, a very short amount of time; we severely overestimated the communication cost (communication between processors has high latency but not high bandwidth, so passing many parameters every once in a while wasn't too bad). Sampling the chains has many steps, it seems that taking the sigmoid f(x)=1/(1+exp(-x)) took about 40% (0.045s/sample) while taking the dot products took 60% (0.78s/sample), and generating random samples took a negligible amount of time. One thing we could do is reduce the float precision on this part since the results don't need to be too precise here, as we're just generating random samples. On the other parts though, the math needs to be much more precise since we're using the actual values to update, and not sampling markov chains.

Of course, then we would be missing out on the most obvious way to save time: to reduce the number of chains! It turns out that reducing the number of chains by a lot actually works, to a degree. Reducing the number of chains by too much (below 100, the standard used in the paper for the original) causes divergence, however, as we can see from the massive spikes in error which occur frequently in the 3 chain case. But the time savings are massive, and the training still works, so this tells us that we can sample less often from the chains in the serial version and the algorithm will still work, and in the parallel version that we can reduce the number of chains per core. The following experiment was conducted on 8 processes.





Deliverables

Our network is, of course, capable of generating things that it thinks are handwritten character samples. Here are some examples, and a video of it sampling in action: https://dl.dropboxusercontent.com/u/59046488/out400x400.mp4

1 epoch 5 epochs 30 epochs

After 2000 steps of sampling

Conclusion

The project to parallelize Deep Boltzmann Machines in OpenMPI was successful, albeit too successful. We tried our best to increase the efficiency of the algorithm in

other ways, and we were able to successfully reduce the training time significantly by parallelizing the chain sampling portion of the algorithm.

Citations

http://www.jmlr.org/proceedings/papers/v5/salakhutdinov09a/salakhutdinov09a.pdf

http://www.jmlr.org/proceedings/papers/v9/salakhutdinov10a/salakhutdinov10a.pdf