Fall 2012

GPU implementation of a deep learning network for image recognition tasks

Sean Patrick Parker
University of Iowa

GPU IMPLEMENTATION OF A DEEP LEARNING NETWORK FOR IMAGE RECOGNITION TASKS

by

Sean Patrick Parker

A thesis submitted in partial fulfillment of the requirements for the Master of Science degree in Electrical and Computer Engineering
in the Graduate College of The University of Iowa

December 2012

Thesis Supervisor: Associate Professor Tom Schnell

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

MASTER'S THESIS

This is to certify that the Master's thesis of

Sean Patrick Parker

has been approved by the Examining Committee for the thesis requirement for the Master of Science degree in Electrical and Computer Engineering at the December 2012 graduation.

Thesis Committee: _____
Tom Schnell, Thesis Supervisor

_____
Jon Kuhl

_____
Anton Kruger

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

State of the Problem and Solution Approach

Accurate automated object recognition is a highly sought after machine task for many industries ranging from military intelligence to manufacturing. The ability to visually monitor an environment without human observation potentially leads to increased productivity and security. For instance, images sent from unmanned aerial vehicles could be scanned to detect threats early and allow for proper safety precaution. If a task is focused enough and domain knowledge exists, hard coded systems can be built for a specific task. However, the complexity of visual input makes a general, multiclass system require very high computational power.

Fortunately the advent of general purpose computing on Graphics Processer Units could potentially lead to better solutions. Intuitively the use of GPUs makes sense. The hardware is designed to rapidly render visual images so using the same structure to do visual detection seems logical. In reality, the process of classification involves a very different set of calculations. GPUs also have several disadvantages when compared to CPUs such as weak branching and strict memory limitations. Despite these downfalls, a GPU implementation of such a system can still be accomplished and will provide a fast and accurate solution to the problem.

Overview of Machine Learning

Machine learning is a subset in the larger field of artificial intelligence that is focused on the ability of a system to recognize patterns contained in data. The applications of such systems vary greatly from simple polynomial curve

fitting to complex image or speech recognition. The implementation of such systems is equally variable with certain methods being more practical for certain tasks.

Types of Learning

The learning process can be separated distinctly into two categories: supervised and unsupervised. Supervised learning uses a data set that has associated labels given to each input. That is, the "right answer" is given to the system during learning so the parameters defining the system are adjusted towards this solution. Unsupervised learning only needs the empirical data itself to extract some useful information. Since no labels are needed, no human input is required which allows unsupervised learning to use much larger data sets or a continuous stream of input data.

Types of Problems

Supervised learning is used when a problem requires an output selection based on a given input vector. If the output is the selection of one of a finite number of categories the problem is called *classification*. If the output is a real number or continuous variables the problem is called *regression*. Unsupervised learning is used to find out less context specific information about data sets. For example the discovery of groups within data is called *clustering*. *Density estimation* is a determination of the distribution of data in input space. Lastly, projection of high-dimensional space can be used for *compression* or, if projected to two or three dimensional space, *visualization*. (Bishop, 2006)

Neural Networks

Neural networks are a class of machine learning systems that were originally inspired by the architecture of the brain. Although none of the models

are true to the information processing mechanisms of nature, they all borrow the basic idea: a system of interconnected units that can become activated due to some combination of input stimulation.

Artificial Neurons

The basic unit in a neural net is called the neuron. Mathematically the neuron can be thought of in two parts. The first is a linear combination of all the inputs:

$$a_j = \sum_{i=1}^{D} w_{ij}^{(1)} x_i + w_{0j}^{(1)}$$

a_j is known as the *activation* of a given neuron. x_i is the value of input i and is multiplied by a weight w_{ji}[1]. The second part involves taking the activation value and passing it through a differentiable, nonlinear *activation function*:

$$z_j = h(a_j)$$

Two of the more common choices for the activation function are hyperbolic tangent and logistic sigmoidal. (Bishop, 2006)

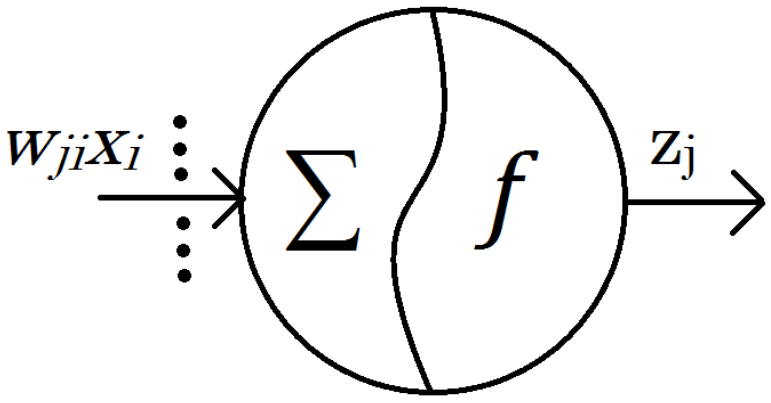

Figure 1: An Artificial Neuron

[1] The term w_{ji} refers to the weight to neuron j from neuron i. w_{j0} is not multiplied by any input and is known as the *bias* for node j.

Multilayer Perceptron

The most general among these neural networks is a feed forward network called the multilayer perceptron. The multilayer perceptron is a supervised learning system used for classification and contains three layers[2]: input, hidden, and output. Units within a given layer have no connections to other units within the same layer but are fully connected to each unit in adjacent layers. The "feed forward" aspect of the network comes from the fact that the output from each previous layer is used as the input to the next.

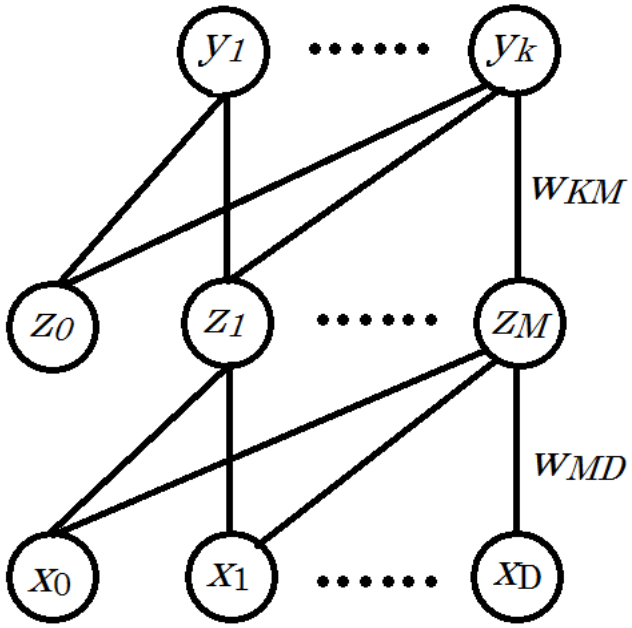

Figure 2: Multilayer Perceptron Network

[2] Some literature regards the weights themselves as a layer and would consider the basic multilayer perceptron a two-layer network.

The bottom layer is input, either direct or preprocessed, from the dataset and has a size determined by the data[3]. The middle layer is called the hidden layer, the size of which can be selected. The top layer is called the output layer with size determined by the number of classes. Given an input vector, each neuron in the hidden layer will calculate z_j which will be used as input to each neuron in the output layer. Ideally, only 1 of K output values will be nonzero which, in the case of classification, identifies the correct class for the input data. Decision uncertainty given a particular input vector can be seen by output values that are not exactly 0 or 1.[4]

Network Training

Before a network can accurately interpret data it must first go through a training phase. In this step the parameters, the weights and bias', are adjusted to fit the data. Since the goal of the system is classification, an error function is used that reflects the accuracy of an output given an input. Specifically, given an N labeled input vectors:

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{n=1}^{N} \|\boldsymbol{y}(\boldsymbol{x}_n, \boldsymbol{w}) - \boldsymbol{t_n}\|^2$$

The systems resulting output is the vector $\mathbf{y}$ and $\mathbf{t}$ is the true value from the associated label. In order to minimize the error the weights are then adjusted to descend along the gradient. Using a technique called *back propagation* the difference in the output compared to the label is sent backwards through the network to calculate the change in weights that corresponds to the gradient descend of the error function.

---

[3] Or the result of the preprocessing

[4] Assuming the activation function for the final layer is sigmoidal or similar function.

$$\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} + \Delta\boldsymbol{w}^{(\tau)}$$

In general, there are two ways that learning stops. Either training occurs over the maximum number of *epochs*[5] or convergence is reached. Convergence is said to have occurred when the change in error over a subset from one weight update to the next is smaller than some set value θ.

Basis Functions

The classification problem can be thought of as dividing a k-dimensional input space into regions belonging to a single class. The most basic machine learning algorithms are basic methods to move a decision line to best describe the data. This approach only works if the data is linearly separable. To classify data that is separable in some non-linear way basis functions, denoted $\varphi_j(x)$, are used to transform the input. For most methods these are fixed nonlinear functions combined in a linear fashion. This means these basis functions must be selected ahead of time which is difficult and generally requires some intuition of knowledge of the data. The basic structure of a neural network acts a linear combination of basis functions each of which itself is a nonlinear combination of linear inputs. The weights represent the coefficients on these linear inputs and therefore are adjustable and constantly updated during learning. Neural networks can therefore transform input using adaptive basis functions that are initially unknown. (Bishop, 2006)

Energy-Based Models

Consider the task of classification. The prediction from the model should give a probability for each class based on the input vector.

$$P(\boldsymbol{y}|\boldsymbol{x})$$

[5] An *epoch* is a complete pass over the entire training data set.

The general task of learning then becomes finding the set of model parameters that, given the data set, maximizes the conditional probability of all the answers.

$$\prod_{i=1}^{P} \boldsymbol{P}(\boldsymbol{y^i}|\boldsymbol{x^i}, \boldsymbol{w})$$

The parameters that maximize this product are equivalent to those that minimize the negative log[6] of it.

$$-\mathbf{log}\prod_{i=1}^{P} \boldsymbol{P}(\boldsymbol{y^i}|\boldsymbol{x^i}, \boldsymbol{w}) = \sum_{i=1}^{P} -\mathbf{log}\,\mathbf{P}(\boldsymbol{y^i}|\boldsymbol{x^i}, \boldsymbol{w})$$

The issue with using direct probabilities is that the calculation requires some normalization with respect to all configurations of a system. If a system large or complex this normalization task becomes intractable, especially when it the calculation is required after each parameter update. Instead, functions that can be calculated in a simpler fashion are used to define the system and comparison of states can give a learning gradient.

Energy functions are defined for a system with respect to that systems state variables and parameters. The essence of learning in such a model is to tune parameters to have low energy in configurations brought about by training data and to have high energy in other states. If classification is the end result the task becomes: given the configuration of input variables, find the configuration of output variables with the lowest energy. The specific learning algorithm is dependent on the model and classification is generally done by calculating probability from the energy distribution. (LeCun, et. al 2006)

[6] The logarithm function is monotonically increasing.

CHAPTER 2
BOLTZMANN MACHINE

General Boltzmann Machine

In the early 1980s Geoffrey Hinton and Terry Sejnowski developed the original concept of a Boltzmann machine. Borrowing from the field of thermodynamics, their research expanded upon learning systems of the day[7]to add a stochastic approach which allowed for an escape from local minima during the learning process.(Hinton and Sejnowski, 1983)

Structure

The Boltzmann machine is "a network of symmetrically coupled stochastic binary units." (Salakhutdinov and Hinton, 2009) The system has a layer of *visible* binary units and a layer of *hidden* binary units. Each unit has a bidirectional connection to each other unit in the system. The purpose of this machine is to be able to estimate the expectations that two connected units would both be on. This is done by allowing the machine to settle into near equilibrium distribution using both random initialization and data driven visible units. The general idea is to adjust the weights so that the random initializations settle into states similar to those which exist in the environment[8]. "The network will be said to have a perfect model of the environment if it achieves exactly the same probability distribution over these 2^v states when it is running freely at thermal equilibrium *with no environmental input*." (Hinton and Sejnowski, 1983)Although the system can never be a perfect model, regularities in the data can be captured to give a close estimation of the true probability.

---

[7]Most notably the system was very similar to that described by Hopfield.

[8] As presented by the training data

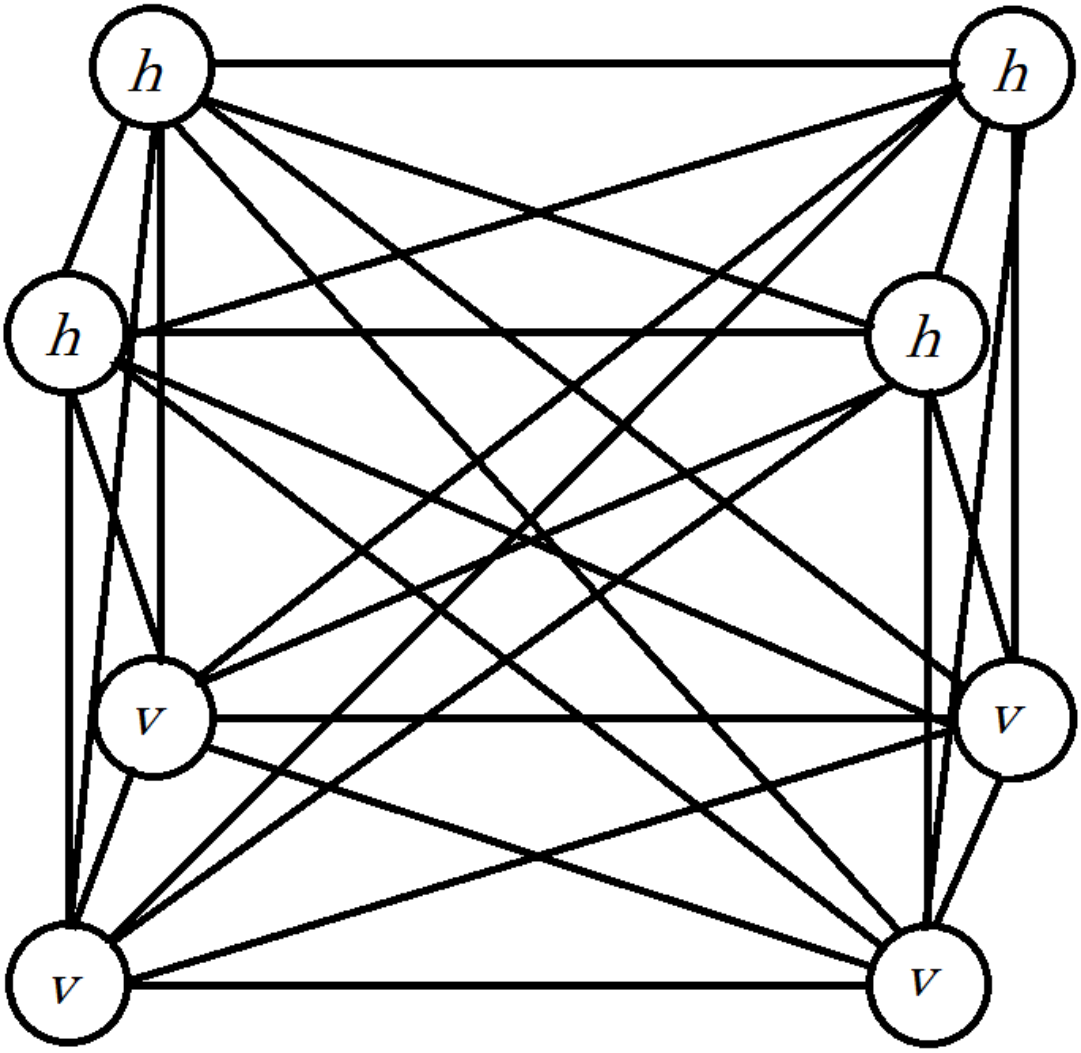

Figure 3: General Boltzmann Machine

Learning

Boltzmann machines fall into the category of unsupervised learning. Unlike supervised methods there is no correct answer to compare against and

adjust towards. Instead an energy function[9] (Salakhutdinov and Hinton, 2009) of the current state is used:

$$E(\boldsymbol{v}, \boldsymbol{h}; \theta) = -1/2\, \boldsymbol{v}^T \boldsymbol{L} \boldsymbol{v} - 1/2\, \boldsymbol{h}^T \boldsymbol{J} \boldsymbol{h} - 1/2\, \boldsymbol{v}^T \boldsymbol{W} \boldsymbol{h}$$

L, J, and **W**are, respectively, the visible-to-visible, hidden-to-hidden, and visible-to-hidden weights of the systems parameters θ. This energy function can then be used to calculate the probability the system assigns to a given set of visible units:

$$p(\boldsymbol{v}; \theta) = \frac{1}{Z(\theta)} \sum\nolimits_h e^{-E(\boldsymbol{v}, \boldsymbol{h}; \theta)}$$

Z(θ) is the *partition function* which normalizes the probability and is given by:

$$Z(\theta) = \sum\nolimits_v \sum\nolimits_h e^{-E(\boldsymbol{v}, \boldsymbol{h}; \theta)}$$

Note that the partition function involves calculating for all possible states of the system and changes if the parameters of the system are changed. The conditional probabilities do not require the partition function and are given as:

$$p\left(h_j = 1 \middle| \boldsymbol{v}, \boldsymbol{h}_{-j}\right) = \sigma\left(\sum\nolimits_{i=1}^{D} W_{ij} v_i + \sum\nolimits_{m=\frac{1}{j}}^{P} J_{jm} h_j\right)$$

$$p(v_i = 1 | \boldsymbol{h}, \boldsymbol{v}_{-i}) = \sigma\left(\sum\nolimits_{j=1}^{P} W_{ij} h_j + \sum\nolimits_{k=1/i}^{D} L_{ik} v_i\right)$$

The aim of learning with the system is to update the parameters of the system so that it models reality[10] as accurately as possible. This would mean that it should be equally likely in the data and in the model that any two units are on at the same time. Essentially the goal is for the model's expectation to be the same as the data-dependent expectation:

$$E_{p_{model}} = E_{p_{data}}$$

[9] Energy function given without bias terms

[10] More specifically, the representation of reality given by the data

$E_{p_{data}}$, more specifically, is the expectation with respect to the empirical distribution(Salakhutdinov and Hinton, 2009):

$$P_{data}(\boldsymbol{h}, \boldsymbol{v}; \theta) = p(\boldsymbol{h}|\boldsymbol{v}; \theta)P_{data}(\boldsymbol{v})$$

Therefore, to update each parameter so that the system ascends along the gradient of log-likelihood the weight parameters are updated according differences in the units they connect (Hinton and Sejnowski, 1983):

$$\Delta \boldsymbol{W} = E_{P_{data}}[\boldsymbol{v}\boldsymbol{h}^{\mathrm{T}}] - E_{P_{model}}[\boldsymbol{v}\boldsymbol{h}^{\mathrm{T}}]$$

$$\Delta \boldsymbol{L} = E_{P_{data}}[\boldsymbol{v}\boldsymbol{v}^{\mathrm{T}}] - E_{P_{model}}[\boldsymbol{v}\boldsymbol{v}^{\mathrm{T}}]$$

$$\Delta \boldsymbol{J} = E_{P_{data}}[\boldsymbol{h}\boldsymbol{h}^{\mathrm{T}}] - E_{P_{model}}[\boldsymbol{h}\boldsymbol{h}^{\mathrm{T}}]$$

Issues

The exact computations of the aforementioned expectations grow exponentially with the number of hidden units. Fortunately there exist a number of techniques that can be used to estimate the expectations and thereby allow an approximate gradient ascent. These techniques require calculating towards stable states of the systems. Since the machine is fully connected the units takes a very long time to reach a stationary distribution. This interdependence makes learning impractical. Even if one of the layers is known[11] the other layer still has free interdependent units so the state is not known in a single step. (Salakhutdinov and Hinton, 2009)

Restricted Boltzmann Machine

In the mid 1980's cognitive scientist Paul Smolensky developed a theory of information processing called Harmony Theory. It was similar to the idea of Boltzmann machine in that using statistical methods the system attempted to predict latent variables (environmental features) to become in harmony with the

[11] As is the case in the calculation of $p(\boldsymbol{h}|\boldsymbol{v}; \theta)$

true environment. In fact, in a paper published in 1986 Smolensky showed how a modified, parallelized Boltzmann machine could have exact inference with what is now referred to as a restricted Boltzmann machine. (Smolensky, 1986)

Structure

The restricted Boltzmann machine is a Boltzmann machine in which both **J** and **L** are set to 0. Simply put, there are no connections between units of the same layer but each unit is still fully connected to the units of the other layer. The advantage of this configuration can be seen with the reduced conditional probabilities. [12]

$$p(h_j = 1|\boldsymbol{v}) = \sigma(\sum_{i=1}^{D} W_{ij} v_i)$$

$$p(v_i = 1|\boldsymbol{h}) = \sigma(\sum_{j=1}^{P} W_{ij} h_j)$$

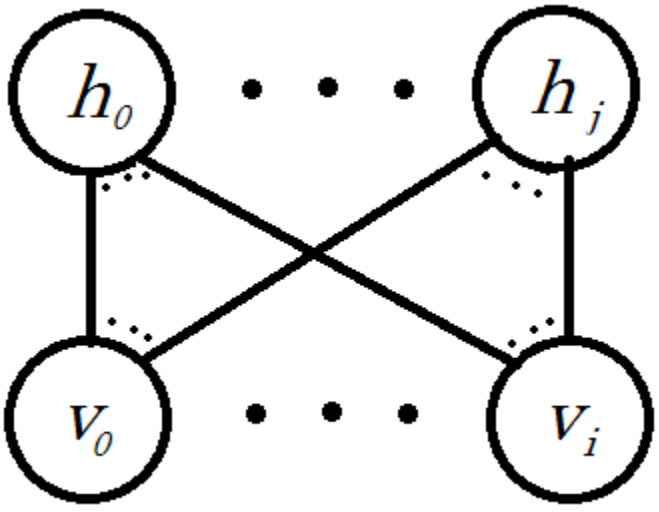

Figure 4: Restricted Boltzmann Machine

[12] The bias terms has been omitted from the conditional probabilities and the image of the restricted Boltzmann machine.

The energy function for a given configuration is now simplified[13] to:

$$E(\boldsymbol{v}, \boldsymbol{h}; \boldsymbol{W}) = -\frac{1}{2}\boldsymbol{v}^T \boldsymbol{W}\boldsymbol{h}$$

This simplification means the hidden state can be calculated in a single parallel, upwards pass given any visible vector. If the visible vectors are randomly sampled from the training data the data driven expectation becomes trivial to find. However, the models expectation does not have samples to draw from so some approximation method must be used. (Salakhutdinov and Hinton, 2009)

Learning

The basis for learning with RBMs comes from a Markov Chain Monte Carlo method known as a Gibbs sampler. Markov chains are a sequence of random variable states that are transitioned through using known conditional probabilities. One requirement is that the outcome of each transition must solely depend on the previous state. To start a Markov chain, an initial state must be chosen. After a small number of transitions the state of the system is still highly influenced by the initial selection. After enough transitions, however, the state will be independent of the initial selection and will have reached a stationary distribution. Sampling from this stationary distribution is shown to be equivalent to sampling from the actual probabilities and a Monte Carlo approach can be used to approximate the model's expectation. A Gibbs sampler is a special case where the transitions only involve one random variable. It can easily be seen from the conditional probabilities of the RBM that each upwards or downwards pass satisfies that criteria. Since the Markov chain must then be constructed in half steps this method is called alternating Gibbs sampling. (Walsh, 2004)

[13] Bias terms omitted for clarity

Contrastive Divergence

Standard maximum-likelihood learning of parameters would be done by gradient ascent along the average log-likelihood.

$$\Delta \boldsymbol{W} = \frac{\partial L(\boldsymbol{W}|\boldsymbol{x})}{\partial \boldsymbol{W}}$$

Where

$$\frac{\partial L(\boldsymbol{W}|\boldsymbol{x})}{\partial \boldsymbol{W}} = -\langle \frac{\partial E(\boldsymbol{x}|\boldsymbol{W})}{\partial \boldsymbol{W}} \rangle_0 + \langle \frac{\partial E(\boldsymbol{x}|\boldsymbol{W})}{\partial \boldsymbol{W}} \rangle_\infty$$

The first part of the right hand side can easily be calculated using training data. As mentioned in the discussion of General Boltzmann machines, calculation of the second term involves use of the partition function[14] and cannot be done efficiently. MCMC is a great way to sample from complex, unknown distributions. However, running the chain sufficiently long enough to get a good sample is very slow and equilibrium is hard to guarantee.

Contrastive divergence attempts to follow the gradient of an entirely different function. The Kullback-Leibler divergence is as follows:

$$KL(p_0||p_\infty) = \sum_{\boldsymbol{x}} p_0(\boldsymbol{x}) \log \frac{p_0(\boldsymbol{x})}{p(\boldsymbol{x}|\boldsymbol{W})}$$

It is possible, depending on the system[15], to minimize this divergence itself, however, in order to leave the initial distribution, p_0, unaltered it makes more sense to use difference between two divergences. If the calculation of p_∞ is intractable using the difference also allows the expectations from the equilibrium distribution to cancel out. (Hinton, 2002)

$$\mathrm{CD_n} = \mathrm{KL}(p_0||p_\infty) - \mathrm{KL}(p_n||p_\infty)$$

The minimization of the contrastive divergence therefore becomes:

$$\Delta \boldsymbol{W} = \langle \frac{\partial E(\boldsymbol{x}|\boldsymbol{W})}{\partial \boldsymbol{W}} \rangle_0 - \langle \frac{\partial E(\boldsymbol{x}|\boldsymbol{W})}{\partial \boldsymbol{W}} \rangle_n + \partial \varphi$$

---

[14] The sum of an exponential number of terms

[15] Namely, if p(x | W) is tractable

The third term is gradient of the reconstruction distribution with respect to the parameters multiplied by the gradient of the KL divergence with respect to the reconstruction distribution. It is "problematic to compute, but extensive simulations show that it can safely be ignored because it is small and it seldom opposes the resultant of the other two terms." (Hinton, 2002)

It should be noted that the Kullback-Leibler divergence is not symmetric[16] and provides a biased estimate of the maximum likelihood learning. However, the bias has been shown to be small and a number of successful applications have proven it to be an effective and efficient learning signal. It should also be noted that since the third term of the difference is not used the learning algorithm isn't actually following any function. For a Restricted Boltzmann machine the parameter updates using contrastive divergence become:

$$\Delta \boldsymbol{W} = < \boldsymbol{vh} >_0 - < \boldsymbol{vh} >_n$$

Learning works even if the reconstruction used to estimate the models expectation is done in only a single step. This is denoted as CD_1. The trade off for more steps (CD_n) is an improvement of the models expectation estimate[17] for the time required to another up-down pass. Much of the common literature using contrastive divergence as a learning method uses CD_5. (Carreira-Perpinan and Hinton, 2005)

Persistent CD

The cost of running more steps of Gibbs Sampling to get a better approximation of the likelihood gradient is large because all of those steps need to be performed for each data vector. This is because each time an update occurs all of the Markov chains are reset to the data. If the chains acted continuously

[16] $KL(p_0||p_\infty) \neq KL(p_\infty||p_0)$
[17] And therefore a better approximation of the gradient

across mini-batches they would be much reach a state much closer to equilibrium. This allows for something that approaches CD_∞ for the calculation cost of CD_1[18].

The key consideration with PCD is the weight updates. If a Markov Chain is started under one model and then that model's parameters are changed the current chain is not guaranteed to be any closer to equilibrium on the new model than a data vector. Fortunately, if the changes in the parameters are small enough, the persistent chain is generally closer to equilibrium than a single step from the data vector. In fact, for an infinitesimally small learning rate the updates become exactly equivalent to CD_n. (Tielman, 2008)

Training Considerations

To assist with training, a number of common machine learning techniques can be applied to the process. These tweaks generally effect the way the weight values are updated and are not related to the structure of the machine itself. Many of these techniques involve a meta-parameter that can cause a non-trivial change in both the speed of learning and the accuracy of the system.

Learning Rate

A learning rate is standard with almost any learning system. Denoted by ε, the learning rate is used to scale the updates.

$$\boldsymbol{W} = \boldsymbol{W} + \varepsilon \Delta \boldsymbol{W}$$

The rate is needed since most gradient updates by themselves would have far too great an impact on the system. Since each update is only based upon a limited amount of information from the environment a radical shift in parameters would lead to a slightly unstable learning process where interdependent parameters

[18] Assuming that each chain is only updated one step per batch

often oscillate across correct values. If the learning rate is too small learning convergence will be slow and training will take longer than necessary. It is common practice to start with a larger learning rate that is decreased towards 0 throughout training. (Bishop, 2006)

Mini-Batches

Another common technique used in machine learning is the idea of splitting up the dataset into small subsets and updating parameters only after the gradient has been calculated from all samples in the subset. It is also common practice to divide the total gradient by the number of samples in the mini-batch so that the learning rate is not dependant on mini-batch size. Depending on the system, some literature suggest a mini-batch consisting of one example of each class whereas others use upwards of 100. If the examples of the batch can be learned in parallel then a larger mini-batch size can reduce training time. (Hinton, 2010)

Fantasy Particles

Determining the number of fantasy particles to use is something specific to stochastic methods using sampling. In the RBM, when the calculation from visible to hidden takes place the calculation for an up-pass results in the probability of a hidden unit being active given the visible units. Since this hidden layer consists of binary units the probabilities are each compared against a random number 0.0-1.0. If the probability is greater than the randomly generated number the binary unit is activated.

Ideally, to accurately sample the subspace of the model a very large number of these possible hidden configurations should be sampled. Calculation time is the clear compromise as each fantasy particle needs to go through all the

steps required for the chosen alternating Gibbs steps. (Salakhutdinov and Hinton, 2009)

Sparsity

"Discriminative performance is improved by using binary features that are only rarely active." (Nair and Hinton, 2009) This statement seems to hold true most of the time. Rarely active features tend to code for more specific patterns in the visible layer and are also easier to interpret as input into another RBM.

If an RBM is designed with a specific probability of each hidden unit being active, p, then the common error measure used is the cross entropy between the desired and actual probability.

$$p \log q + (1 - p) \log (1 - q)$$

Where q is an estimate of the current probability. With respect to the RBM training algorithm it makes the most sense to estimate q from the current mini-batch[19]. The derivative if the error measure cleanly becomes:

$$p - q$$

Also, to improve the quality of the estimate a running estimate can be used with a chosen decay factor.

$$q_{new} = [decay] * q_{old} + [1 - decay] * q_{current}$$

Ensuring sparsity also ensures that if a hidden unit's probability drops below the target it will be pulled back up. This prevents a hidden unit from becoming useless to the system. (Hinton, 2010)

Momentum

Momentum is a technique used both to increase the speed of learning and to help dampen oscillations. Oscillations occur when parameters swing back and

[19] The quality of the estimate is therefore directly related to the size of the mini-batch

forth across the optimum value due to larger than ideal changes thereby leading to suboptimal solutions. The motivation comes from the fact that many objective functions contain long, narrow and fairly straight ravines. Ideally, once these ravines are roughly defined in weight space the learning should be fairly consistent along the floor. Normally this gradient is followed in a stepwise fashion and is limited by the learning rate. If instead the weight is updated by a velocity then the learning can happen more quickly with a smaller learning rate while still benefitting from smaller oscillations. Parameters are updated as follows:

$$\Delta\theta_i(t) = v_i(t) = \alpha v_i(t-1) - \varepsilon \frac{dE}{d\theta_i}(t)$$

α is the parameter used to control how fast the velocity decays. Generally it is a good idea to keep this small[20] for the initial learning and then increase the value afterwards. (Hinton, 2010)

[20] 0.5 seems to work in most cases

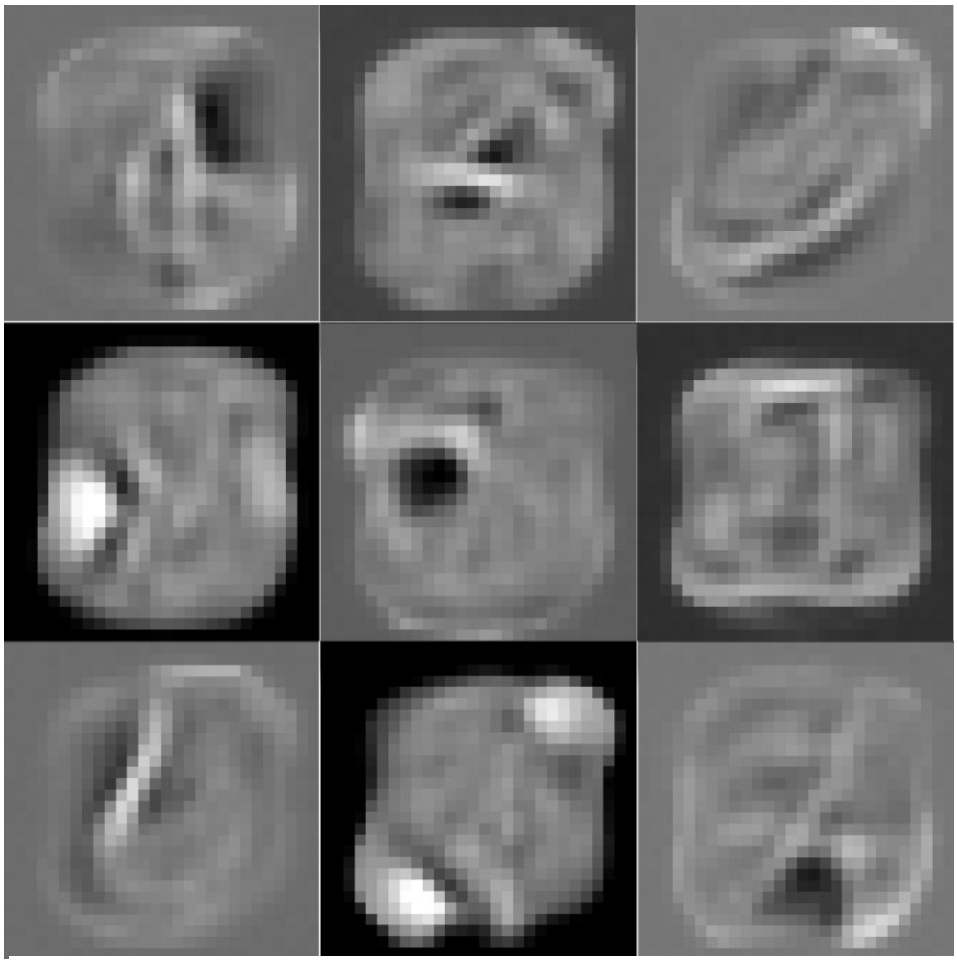

Figure 5: Projections of the Weights from a Single Hidden Unit to the Visual Layer. This RBM was Trained Using the MNIST Dataset and a Sparsity Target of 0.1

CHAPTER 3

DEEP BELIEF NETWORKS

A Restricted Boltzmann machine, by itself, acts as a feature detector that represents common patterns in the data using a binary level of hidden layer memory. It is a purely generative model. If the hidden layer from an RBM were then used as the "visual" layer for another RBM that second level RBM would essentially be learning features of the features. These second order features could help model an even better representation of the input space and hence a better generative model. In theory these RBMs could be stacked indefinitely and *n*th order features could be models. In practice little is gained after about three layers. When a system consists of pre-trained RBMs that are stacked in such a fashion it is called a deep belief net.

Theory

Assume a multilayer generative model using maximum likelihood learning. The energy of this system for a given configuration is defined as:

$$E(\boldsymbol{v^0}, \boldsymbol{h^0}) = -[\log p(\boldsymbol{h^0}) + \log p(\boldsymbol{v^0}|\boldsymbol{h^0})]$$

The model's approximation of the hidden units given a data vector is noted by:

$$Q(\boldsymbol{h^0}|\boldsymbol{v^0})$$

Neal and Hinton (1998) have shown that such a model has the negative log probability of a single data-vector that is bounded by the expected[21] energy minus entropy.

$$\log p(\boldsymbol{v^0}) \geq \sum_h Q(\boldsymbol{h^0}|\boldsymbol{v^0})[\log p(\boldsymbol{h^0}) + \log p(\boldsymbol{v^0}|\boldsymbol{h^0})] - \sum_h Q(\boldsymbol{h^0}|\boldsymbol{v^0}) \log Q(\boldsymbol{h^0}|\boldsymbol{v^0})$$

[21] Expected under the approximating distribution

If the parameters defining the upwards pass from $\mathbf{v}^0$ to $\mathbf{h}^0$ are frozen then the approximate distribution becomes the true posterior for the model. The bound on the log probability then becomes an equality.

The next level up will use $\mathbf{h}^0$ as input so they are now maximizing the log probability of a dataset based off $Q(\mathbf{h}^0 \mid \mathbf{v}^0)$. Learning the higher level therefore tightens the bound on log $p(\mathbf{v}^0)$ but, since the bound on the lowest level (and hence the data distribution presented to the higher level RBM) is an equality, it can never fall below the value learned at the lower level. This guarantee is not valid for a model that is minimizing contrastive divergence but empirical evidence shows such models generally improve with additional layers. (Hinton *et al.*, 2006)

Discriminative Fine-tuning

A deep belief net can very easily be transformed into a classification system. The top level of the deep belief net is simply treated as an input into a neural net. Standard error back propagation is used down the entire network. The learning rate during this training should be fairly small as to ensure no major changes are made to the features learned by the deep belief net. (Hinton, 2006)

Generative Fine-tuning

Labeled data can be useful for a generative model. If the model can learn about specific groups it will further differentiate features defining the specific classes. This models the energy landscape to keep transitions between dissimilar items of a single class low and helps separate items with similar features but belonging to different classes.

To accomplish this task the top level RBM is trained differently. The input during training should be, as before, the hidden units in the layer below but each

example is now augmented with k additional units using a 1-of-k encoding scheme to activate the appropriate binary unit. This top RBM is now models the joint distribution of the features and the label.

Back-Fitting

The greedily pre-trained layers of the generative model are not optimal with respect to the network as a whole. To fine tune the network as a whole the weights at all levels except for the top must be untied. This allows one set of weights to be optimized for inference and the other for the models generation.

The back-fitting process starts with an up-pass given a training vector. Each hidden unit is stochastically selected using the "recognition" weights. Each generative set of weights then produces it's confabulation of the layer below. In essence, the generative model at each layer is saying "this is the input I believe produced my current state." The generative parameters can then be updated using contrastive divergence. Note that since the weight is now generative only the update is now attempts, given a hidden state, to minimize the difference between what it represents and what the model thinks it represents.

$$\Delta \boldsymbol{W} = h_0 * (v_0 - \widehat{v_o})$$

After the up-pass, the top level RBM is run the Gibbs sampling for a few iterations and then is used to generate each layer in a downward pass. The top level RBM is updated using the standard learning rule. After the downwards pass each "recognition" weight is then used to infer what data most likely caused the current state and the weights are updated. Again, since the weight is now only concerned with recognition the updates are calculated slightly differently. (Hinton *et al.*, 2006)

$$\Delta \boldsymbol{W} = v_x * (h_x - \widehat{h_x})$$

Using this method along with treating the top level as a joint distribution allows the model to generate well. However, alone it is less effective at classifying than standard back prop and neural net. If a system is first back-fitted and then trained for discrimination it generally will do slightly better at classification tasks than back prop alone. (Hinton, 2006)

CHAPTER 4

IMPLEMENTATION

CUDA Overview

The Compute Unified Device Architecture was developed by NVIDIA to allowed general purpose use of the parallel processing power of graphic processing units. GPUs, as the name implies, are constructed to do highly parallelized floating point calculations in order to render scenes and images. The device architecture is designed to only run one set of instructions over all the cores but operating on different data locations. This enables high speeds at the expense of flow control. Caches are also small when compared to CPUs so care needs to be taken when designing applications. (NVIDIA, 2011) All code was developed on the Ubuntu operating system using Nsight Eclipse Edition. NVCC must be used to compile all code.

Kernels

CUDA programs are written in C and defined inside code blocks called *kernels*. Kernels are launched with specified thread dimensions. A *block* defines an array of threads in up to 3-dimensions. A *grid* defines an array of blocks in up to 3-dimensions. A *block* can contain at most 1024 threads. From a hardware standpoint the arrangement of threads matters little. A single block must execute on a single core but everything else is purely virtual.

Inside of a kernel CUDA supplies built in variables to allow a specific thread to know its exact location. The location should not be used to branch to a different instruction set[22] but instead should be used to execute on, and write to, different locations in memory. (NVIDIA, 2011)

[22] Reduced flow control makes this very, very slow.

Memory

The largest memory space available to the GPU is the global memory. It physically lives on the GPU board so data from the host must first be transferred over. This is by far the slowest memory operation so applications should limit the amount of data transferred and transfer the data in large batches. Memory accesses to the global store are fairly slow as well. CUDA provides a number of techniques to help hide access latency. First, if an instruction from one thread is waiting on a memory access the core will go ahead and shift priority to other waiting threads. This requires no additional work from the programmer. Second, global memory accesses should be *coalesced.* Physically, memory transfers occur in set sizes[23] so if a single thread requests 1-byte from global memory an entire block much be brought to the core in order to use. If in-order threads in a block are set to use in-order global memory locations then the access is considered to be *coalesced.* One transfer can then be used by multiple threads thus maximizing bandwidth.

The two other major types of memory used by CUDA are local and shared. Local, as the name implies, is a small memory store that is available only to a given thread. Generally this requires no additional consideration in program design. The second type is shared memory. Shared memory access is much faster than global memory but it can only be used by threads within the same block. Shared memory must be loaded and accessed by a thread and the common technique is a coalesced transfer from global memory and then repeated use by threads in a block. This is only necessary if different threads need to use the same piece of memory. If only a single thread will need the global memory it makes more sense to keep it local. (NVIDIA, 2011)

---

[23] 32, 64, and 128-bytes

RBM Calculations

The highest design priority for the system was speed during the RBM calculations. The calculations happen so often that any small increase in time will be magnified intensely. The memory design was crucial as memory transfers, especially those from the host to GPU, account for a significant portion of time in many CUDA applications. The basic algorithm for a single update is as follows:

1. Load mini batch data to the GPU
2. Stochastic hidden state calculation from training vector
3. Visible reconstruction from hidden state
4. Hidden reconstruction from visible reconstruction
5. Update weights and biases

It should be noted that steps 3 and 4 are dependent on the type of contrastive divergence used. The algorithm given is for CD_1 and if more steps of Gibbs sampling are used then these steps will be repeated n times. Furthermore if persistent Markov chains are used then the data access for steps 3 and 4 will be separate[24] from the stored hidden units of step 2.

Memory Storage and Access

Essentially the system has two inherit opportunities for parallelism. The first is the simultaneous update of all the units in a layer. This is afforded by the RBMs structure and the fact that each hidden unit is independent[25] from each other. The second parallelism is a common one in machine learning and stems from the idea that each data vector from the training mini batch is independent of each other.

---

[24] Except for an initialization pass

[25] In the sense of a single update

The design challenge with such a system as that each kernel must operate on a separate location in memory but only a single reference may be passed to all the kernels. It therefore becomes imperative that the data is stored in global memory in such a way that thread identifiers can be used to locate the appropriate source and store.

Mini batch loading

GPU storage of the mini batch is perhaps the most trivial of the design tasks. First a training item is selected at random using the systems *rand*() function and stored in host memory. The next item is selected and stored immediately after the previous item using a pointer offset. The entire mini batch is stored this way on the host and then a single host to device transfer occurs. The entire mini batch resides on the GPU, available via single pointer to the first value of the first item, with each subsequent example available using pointer offsets.

Upward pass

Recall from the discussion of neurons that each unit has a probability, given a visible vector, of:

$$P(h_j|\boldsymbol{v}) = \sigma(b_j + \sum_{\boldsymbol{v}} v_i * w_{ij})$$

Where σ is the logistic sigmodial function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

If the state of the hidden unit needs to be a stochastically selected binary value then it's state is determined by:

$$h_j = P(h_j|\boldsymbol{v}) > rand(0 - 1)$$

The natural division of labor is for each thread to calculate the state of a single hidden unit. The block dimensions therefore would be one dimensional

and equal to the number of hidden units. Each thread could then realize it's intended hidden unit using the built in CUDA variable.

h_idx = threadIdx.x

However, due to the CUDA limitation of maximum threads/block being 1024 this would not work for most tasks. Fortunately, the calculation can be divided over multiple blocks so the grid becomes a one dimensional array of size BLOCKS_LAYER and each block has hidden size divided by BLOCKS_LAYER threads. Each thread then realizes it's intended hidden unit via:

h_idx = (blockIdx.x * blockDim.x) + threadIdx.x

Similarly the grid's Z[26] dimension is reserved for identifying the intended training example in the mini batch. The intended training example is:

visible[blockIdx.z * VISIBLE_SIZE]

Lastly, each thread must know where to store the calculated hidden unit. The total offset is given:

hidden[blockIdx.z * VISIBLE_SIZE + h_idx]

This representation of memory space through thread dimensions made kernel definitions and implementation concise and readable.

The majority of the calculation time during an up pass involves the summation over all the visible units into a given hidden unit. Moreover, the global memory access to the weight values takes a considerable amount of time and will be greatly slowed if done incorrectly. The weight array is stored row by row with each row representing all connections from a given visible unit. During an up pass, each thread must access the *i*th item from each row. This sounds counterintuitive but actually is the correct way to coalesce the global memory

[26] The Y-dimension was originally used to define which fantasy particle was accessed. However, during tested it was discovered that launching multiple fantasy particles/data vector did not improve the model and greatly slowed performance.

access. The summation take the form of a for loop over the visible units and each weight access is given by:

w[i*HIDDEN_SIZE + h_idx]

This allows each thread to request access to a given element in the same row during each iteration of the loop which the compiler recognizes and executes using a single global memory access to a portion of the row. The speed of each up pass is greatly slowed[27] if the weights are stored in the transposed fashion and accessed in order by each individual thread.

Once the summation is calculated each thread must transform the result by the sigmoid function. The actual calculation is sped up by the use of a built in approximation function.

$$1/(1 + \_\_expf(-x))$$

The use of this function has been compared to both the actual calculation and other approximating techniques. It's speed and accuracy offers the best solution and it's use has been successfully documented in a machine learning application. (Ly, et al., 2008)

Lastly, for stochastic binary calculations of the hidden layer, the probabilities must be compared to random numbers. The CUDA library provides a nicely parallelized implementation of the Mersenne Twister random number generator. An array of floats the size of the hidden layer times the mini batch size is randomly initialized between 0 and 1 using the *curand* function. This array location is then passed to the threads for comparison.

Downward pass

The downward pass is set up structurally the same as the upward pass but the threads are set to update one visible unit each. The block dimensions are

[27] Approximately x50 slower during testing

set to the size of the visible layer. One consideration is that if the normal weight matrix is used then the update scheme would take the format of:

w[v_idx * HIDDEN_SIZE + i]

This corresponds to non coalesced global memory accesses and causes slowdown. To alleviate the problem a duplicate weight matrix is transposed and stored. Since and RBM has identical up-down weights by definition the weight matrix must be transposed and stored after every update. Fortunately the SDK provides a matrix transpose function that uses shared memory and coalesced global accesses and can be done in negligible time. A slight modification is required to make the NVIDIA code work on non-square matrixes and exact details can be seen in the code in appendix B.

Update

The weight updates require a difference of multiplications over the entire weight space.

$$\Delta W_{ij} = v_i(\text{data}) * \text{h}_\text{j}(\text{data}) - v_i(\text{model}) * \text{h}_\text{j}(\text{model})$$

Each weight must also be updated by these values over the entire training mini batch. The threads are allocated in the same was as an up pass[28] but now each block in the Y dimension of the grid is responsible for an entire row of the weight matrix. This corresponds to a number of blocks equal to the visible unit. This is possible since the limit for maximum blocks per grid is much larger[29] than the maximum threads per block.

Each thread then loops over the mini batch and calculates the average change in a single weight. The bias's are updated similarly, however they have

[28] Thread per hidden unit and grid X dimension responsible for splitting up the layer

[29] Around 65k

size equal to the layer they correspond to so the extra Y grid dimension is not needed.

Training Specifics

Some of the general machine learning improvements mentioned in chapter 2 are applied to the updates of the restricted Boltzmann machine. The initial values also are carefully chosen to help ensure faster learning.

Initial Values

The weights are randomly initialized according to a Gaussian distribution with 0 mean and 0.01 standard deviation. Each weight is chosen using the central limit theorem over 100 randomly selected samples. This distribution is similar to what the weights tends to learn after training. That is, most weights will be very small and close to zero and a small portion will have high values corresponding to some piece of a learned feature. Small initial weights tend to be easier to learn and, from a practical standpoint, help negate the possibility of overflow error. (Hinton, 2010)

The bias of each visible unit is initialized after the training data is loaded. The initial bias directly corresponds to how often a pixel is active in the training data.

$$Bias_{v_i} = \log(\frac{P(v_i)}{1 - P(v_i)})$$

The bias of each hidden unit was initialized to -2.0. This roughly corresponds to using the above equation but assuming that the probability in the training data of each hidden unit is 0.1. This initial value corresponds loosely to a sparsity target of 0.1 and if a different target is selected the value be initialized differently.

Improvements

A sparsity target modification is applied to the weight and hidden bias updates. The weight updates calculate q over the training batch and update the estimation with a decay factor of 0.95. The penalty is set to 0.0001 and does not seem to overpower the main objective learning on any of the data sets.

A simple weight decay was applied in the form of:

$$Penalty = 0.0005 * \varepsilon * weight$$

This form of regularization is very simple and has little to no effect on very small weights. The use of such a simple term has been shown to improve generalization of a system and improve performance when working with unseen data examples. It also helps to prevent any overflow errors that might occur. (Moody, 1995)

The weight updates are also calculated using a velocity value. The velocity decays according to a momentum, α, that is slowly increased as follows:

Epoch 1: α=0.5

Epoch 2: α=0.6

Epoch 3: α=0.7

Epoch 4: α=0.8

Epoch 5-n: α=0.9

Monitoring Learning

A number of techniques are employed to check the progress of the system during training. As suggested by Hinton (2010) many of the system statistics can be displayed using histograms to give an overview of the parameter distribution. Visually displaying the layers themselves often helped as well to give more definitive example of the layers generative accuracy. A simple graphics utility was developed to create histograms. The histograms are color coded and dashes

are used to indicate approximate values along the horizontal axis. These visual representations can be found in appendix A.

In addition the probability estimate of the hidden units is printed after each epoch during the training of the RBM. This is used to detrmine the effectiveness of the sparsity target. During RBM training the free energy of the system over the validation data and an equivelent subset of the training data is calculated. Although this value is not, by itself, very useful the comparision between the value over the validation data and the training data helps to monitor overfitting. The free energy of the system is "the energy that a single configuration would need to have in order to have the same probability as all the configurations that contain **v.** The easiest way to calculate the free energy is the following equation. (Hinton, 2010)

$$F(\mathbf{v}) = -\sum_i v_i a_i - \sum_j \log\left(1 + e^{bj + \Sigma_i v_i w_{ij}}\right)$$

While training a neural network the average sum of squares error on the validation data is used to measure progress. Since the validation data is never used during RBM pre-training or back propagation it provided a nice prediction of how the system would perform on a larger set of unseen data. The error gradually reduces until the parameters begin to over fit to the training data. Once this occurs the classification will be less accurate on unseen data making for a less accurate general recognition model.(Bishop, 2006)

Architecture

Designing a system that utilizes both CUDA C kernels and the object oriented approach of C++ presents a number of difficulties in terms of system architecture. The primary objective the machine learning system was speed. The number of calculations and iterations that occur during training means a slight compromise in the speed of single kernel execution can greatly increase total

training time. The second objective of the system, as with any object oriented approach, is to be adaptable and reusable.

System Structure

One of the limiting factors of CUDA is that kernels cannot be defined nor executed[30] from within a class. This is due to the fact that the kernels must be defined with the __global__ directive. As a result all calls must be made from within static functions inside the main .cu file instead of within the class. Fortunately, classes have full access to device memory. Classes can store pointers to device locations, allocate memory on the device, and copy between the device and host.

As a result of these considerations the system was designed as follows. Classes are made for the subcomponents that all systems use. These subunits hold all the system information and can present the data contained within. Higher level classes are built that arrange these subcomponents in the way that define the intended learning system. These high level classes are coupled with .cu files that defined the kernels and the major processes that occur during learning. Lastly, a trainer class is used to control the examples seen and the learning flow.

[30] Execution can technically happen, but not in a clean, object oriented way.

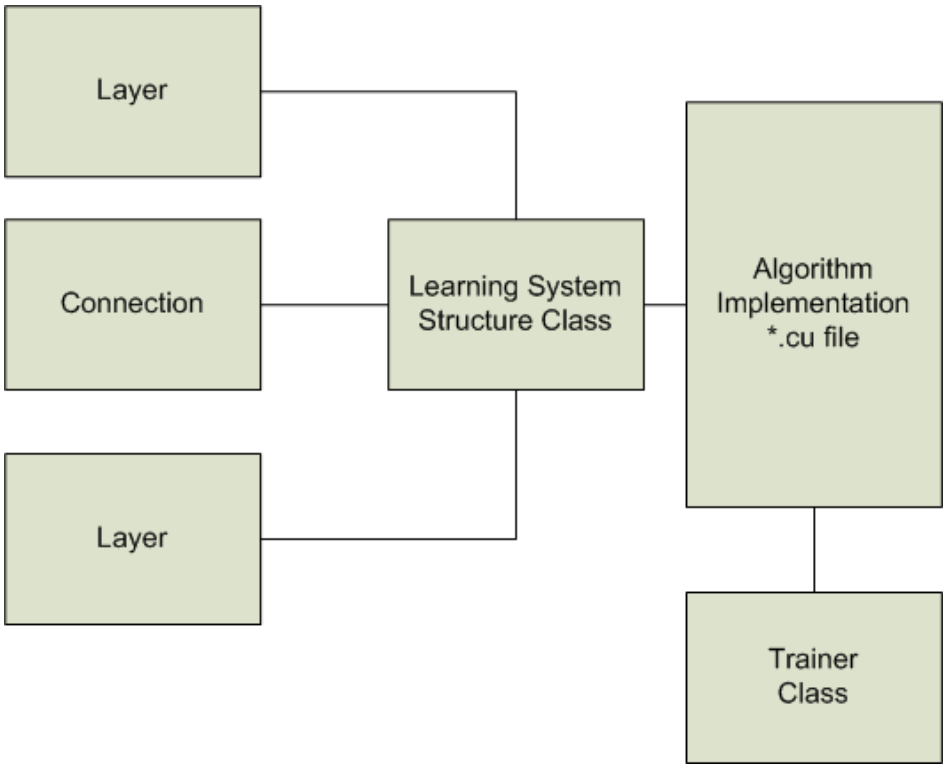

Figure 6: Block Diagram of General System Architecture

Subcomponents

Layer

The Layer class represents the states of groups of neurons. Each layer class contains an array of floats equal to the intended size of the layer. It also stores an additional utility array that can be used by different learning system. This allows, for example, a DBN to save the layer state calculated during the up pass to be used to calculate updates that also require the layer state generated by generative weights. Both these arrays are stored on the device and a single copy exists on the host to store calculated device values in order to view the

layer. This display functionality is implemented using static functions in a namespace defined in the learnUtil.cpp class found in appendix B.

Connection

The connection class contains all the relevant values used to transition between layers. At the core this involves weights and bias's but the class also contains some values that are only necessary during learning. Copies of the parameter values exist on both the device and host. Most of the values can be displayed using histograms which are drawn after the active values on the device are copied to the host.

Trainer

The trainer is a reusable class that is used to disseminate training examples to the system. The data files are directly loaded by the trainer and stored in host memory. The entire data set is never store on the device due to limited memory. The trainer has an array of examples, equal to the size of the mini batch, stored on the device and contains all the control logic to randomly select and load batches. The class also counts how many training examples have been seen during the epoch as well as the total number of epochs trainer over. Display functionality also exists to show training examples as well as to visually show which examples are loaded into a mini batch. This functionality and memory structure applies to labels as well for supervised training methods.

Learning Systems

Currently, three learning systems are implemented. The restricted Boltzmann machine contains the simplest structure containing just two layers and the connection between them. The deep belief net contains 3 layers and the

corresponding connections. The neural network contains an additional top level with the *k* binary units where *k* is the number of classes in the data.

Each system contains, a number of pass through function to the subcomponents to get system facts and device memory location. The function calls act as a wrapper that give the utility layer more meaning with respect to its use in the system. In addition, each system class allows the individual subcomponents to be loaded and saved with a single call.

Algorithm Implantations

The *.cu files associated with a machine define the kernels and modes of the learning systems. The program makes used of #defines to etch out the system configuration to be used for training. This, of course, requires recompilation each time the system changes. It might seem this could be avoided by a configuration file that can be edited in plain text and read in by the compiled program. However, through testing this caused some issues with a slowdown in a number of the CUDA kernels. The slowdown occurs when a loop condition is dependent on a variable that is passed in. Regardless of whether that variable is stored on the device or host this occurs. If the variable is stored in a #define the compiler knows the exact value during compilation and can optimize. Most likely this comes in the form of loop unrolling but it also could provide some guarantee of iterations that is needed to truly coalesce memory accesses.

In general, a good practice when working with CUDA is to have all data that needs to be accessed within a kernel stored on device memory. The transfer time between the host and device is the slowest of all memory accesses so it should be avoided at all costs. However, in the case of the weight updates a few

variables[31] are passed in by value without any measurable slowdown. Latency hiding, which is performed by allowing ready threads to run while waiting on slow memory access, accounts for the similar timing. This most likely would not be the case for faster calculations such as the up pass.

Aside from these considerations the rest of the implementation was fairly straight forwards and the object-oriented component structure allowed for clean, simple code. As is the case with all code, there is most likely room for improvement and most definitely an even more concise way to design portions of the system. Agile development techniques were used and the system was developed over several iterations.

[31] Momentum and learning rate

CHAPTER 5

DATASETS AND RESULTS

MNIST

The MNIST database is a large collection of handwritten digits 0 to 9. The dataset is a subset from the larger NIST database with some modifications to make it more standardized. The set contains 60,000 training examples and 10,000 test images. All of the digits have been size-normalized and centered[32]. The images are an 8-bit grayscale, unlike the original binary data, as a result of the normalization.

The lowest error rate on the test set is 0.23% using a committee of 35 convolutional nets and additional techniques. This result also used elastic distortion on the data to, in essence, allow for more training examples to be seen. A 3-layer NN with 500 and 150 hidden units using the strict[33] MNIST data resulted in an error rate 2.95%. (LeCun, 1998)

This dataset was used for system verification and to see the benefits gained from different learning techniques. Only the original data was used and the systems were trained using a constant number of epochs and consistent learning rates across methods. The data was presented to the system as "pixel probability" meaning the grayscale value of each pixel was converted to between 0-1. The system contained a 784 visual layer, 512 hidden layer, 512 hidden layer, and a top layer of 10 softmax units.

---

[32] Centered with respect to a calculated center of mass of pixels

[33] No distortions of the data allowed

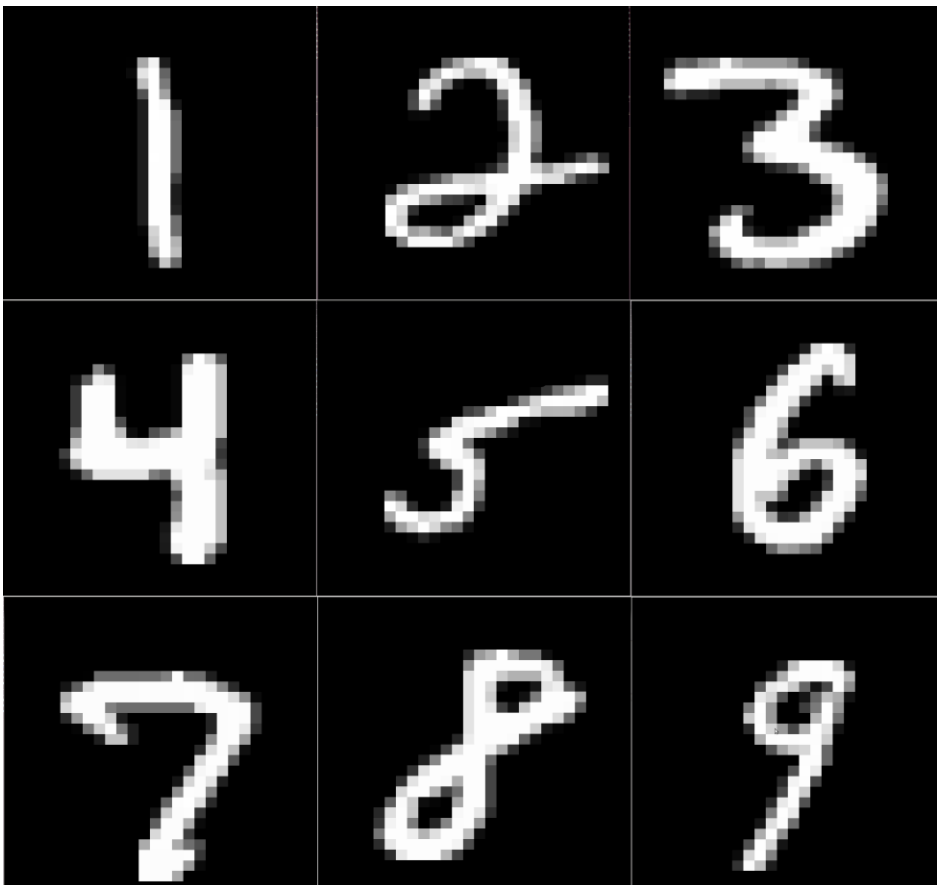

Figure 7: Training Examples from MNIST Dataset

Contrastive Divergence 1

The first run of the system was trained using CD_1. A standard number of epochs was chosen for each part of training in order to get a comparison with later versions. Most likely the system performance would have been improved from longer cycles as no over-fitting seemed to be occurring as evident in free energy comparisons on training and validation data[34].

[34] Only the free energy comparison graphs from the PCD run are shown.

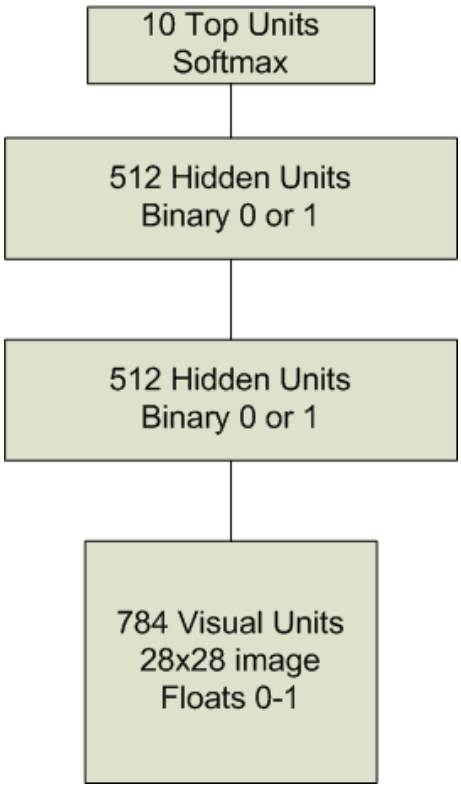

Figure 8: : System Configuration for MNIST

Table 1: Training Parameters for Level 1 RBM (CD1)

Epochs	50
Mini Batch Size	100
Gibbs Steps	1
Learning Rate	0.001
Learning Rate Decay	0

Table 2: Training Parameters for Level 2 RBM (CD1)

Epochs	50
Mini Batch Size	100
Gibbs Steps	1
Learning Rate	0.001
Learning Rate Decay	0

Table 3: Training Parameters for Neural Net (All)

Epochs	200
Mini Batch Size	20
Learning Rate	0.001
Learning Rate Decay	0.0001

Table 4: Classification Results Using CD1

Data Source	**Misclassifications**	**Misclassification Rate**
Training Set	1193	2.21%
Validation Set	151	2.52%
Test Set	273	2.73%

Contrastive Divergence 5

An identical system was used to train using CD_5. The training was around four times slower than CD_1. Classification results showed improvement around 0.2% on all sets. Again, it is believed the system suffered from no overfitting.

Table 5: Training Parameters for Level 1 RBM (CD5)

Epochs	50
Mini Batch Size	100
Gibbs Steps	5
Learning Rate	0.001
Learning Rate Decay	0

Table 6: Training Parameters for Level 2 RBM (CD5)

Epochs	50
Mini Batch Size	100
Gibbs Steps	5
Learning Rate	0.001
Learning Rate Decay	0

Table 7: Classification Results Using CD5

Data Source	Misclassifications	Misclassification Rate
Training Set	1054	1.95%
Validation Set	132	2.20%
Test Set	259	2.59%

Persistent Contrastive Divergence

The same system was trained using a continuous Markov chain across mini batches and epochs. This was the only change from the CD_n training runs. The free energy of the validation data never seems to rise to drastically compared to the training data. The actual value of the free energy is arbitrary and the second level RBM has higher values. The neural net error improve rapidly to start due to the top level weights roughly aligning with correct penultimate layer configurations. The progress becomes much more gradual as the weights begin to fine tune for classification.

Table 8: Training Parameters for Level 1 RBM (PCD)

Epochs	50
Mini Batch Size	100
Gibbs Steps	1 (Persistent Chain)
Learning Rate	0.001
Learning Rate Decay	0

Table 9: Training Parameters for Level 2 RBM (PCD)

Epochs	50
Mini Batch Size	100
Gibbs Steps	1 (Persistent Chain)
Learning Rate	0.001
Learning Rate Decay	0

Table 10: Classification Results for PCD

Data Source	**Misclassifications**	**Misclassification Rate**
Training Set	1088	2.01%
Validation Set	127	2.12%
Test Set	257	2.57%

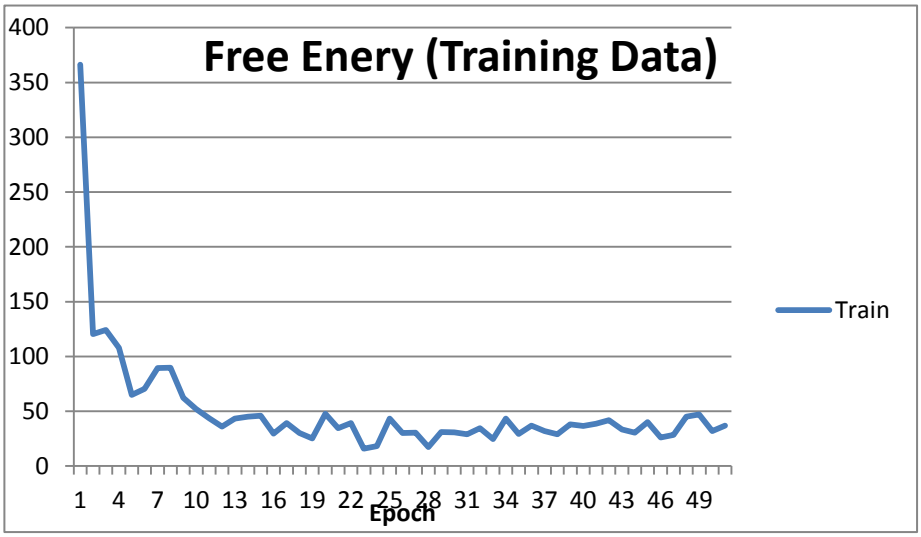

Figure 9: Free Energy over Training Data of Level 1 RBM During Learning Using PCD

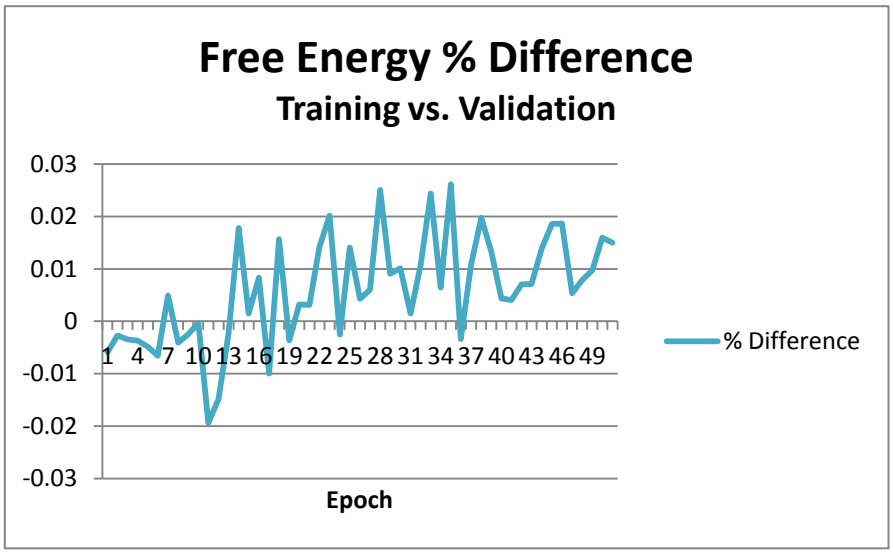

Figure 10: Percent Increase in Free Energy of Level 1 RBM on Validation Data Compared to Training Data During Learning Using PCD

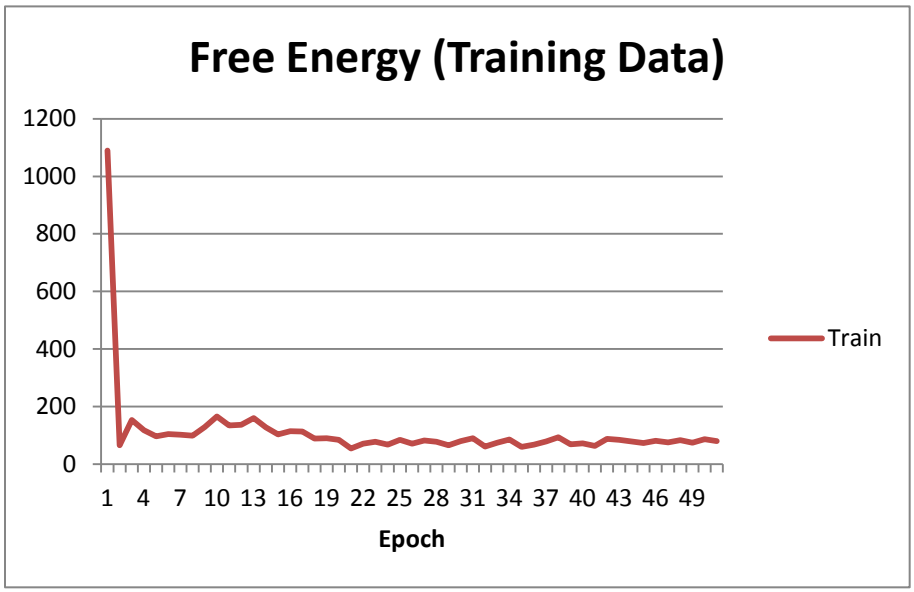

Figure 11: Free Energy over Training Data of Level 2 RBM During Learning Using PCD

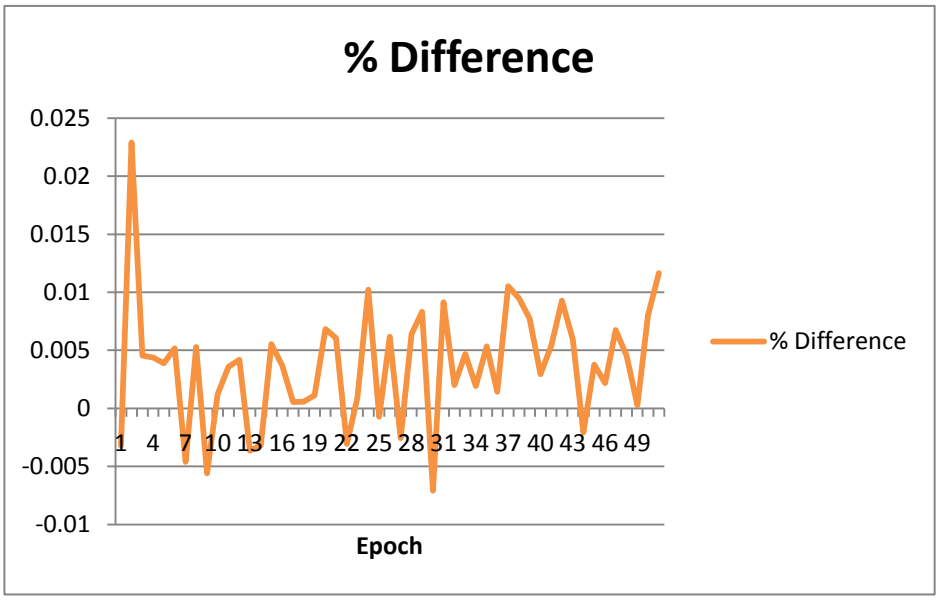

Figure 12: Percent Increase in Free Energy of Level 2 RBM on Validation Data Compared to Training Data During Learning Using PCD

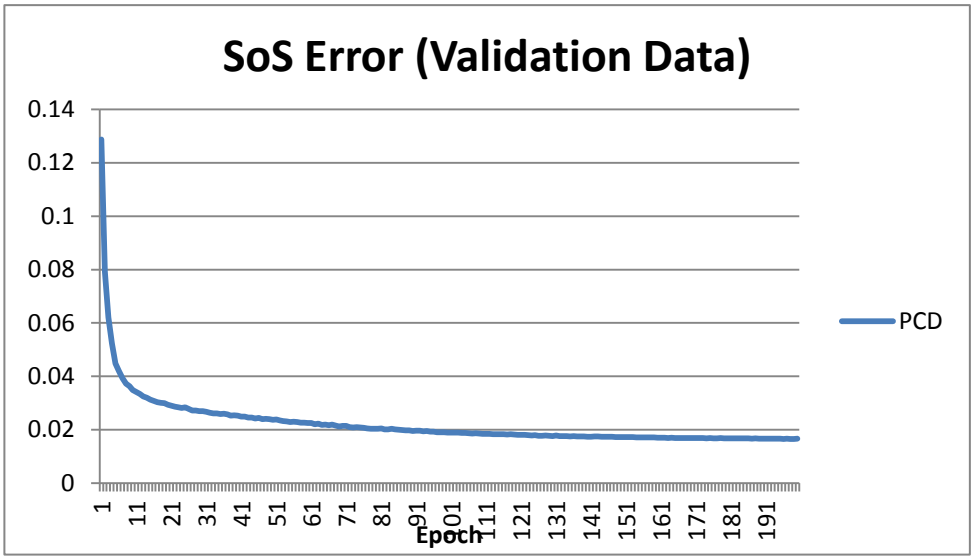

Figure 13: Sum of Squares Error on Validation Data During Neural Net Training

Global Fine-Tuning

The layers that were pre-trained using PCD were then fine tuned using the methods described in Chapter 3. The fine tuned layers were then used to initialize the neural network. Initial error rates were slightly higher most likely due to the random initialization of the top level weight. Steadily the globally tuned network outperformed the stacked RBMs across all data sets.

Table 11: Training Parameters for Global Fine-Tuning

Epochs	100
Mini Batch Size	100
Gibbs Steps	1
Learning Rate	0.001
Learning Rate Decay	0.0001

Table 12: Classification Results for Globally Fine-Tuned DBN

Data Source	**Misclassifications**	**Misclassification Rate**
Training Set	933	1.73%
Validation Set	117	1.95%
Test Set	242	2.42%

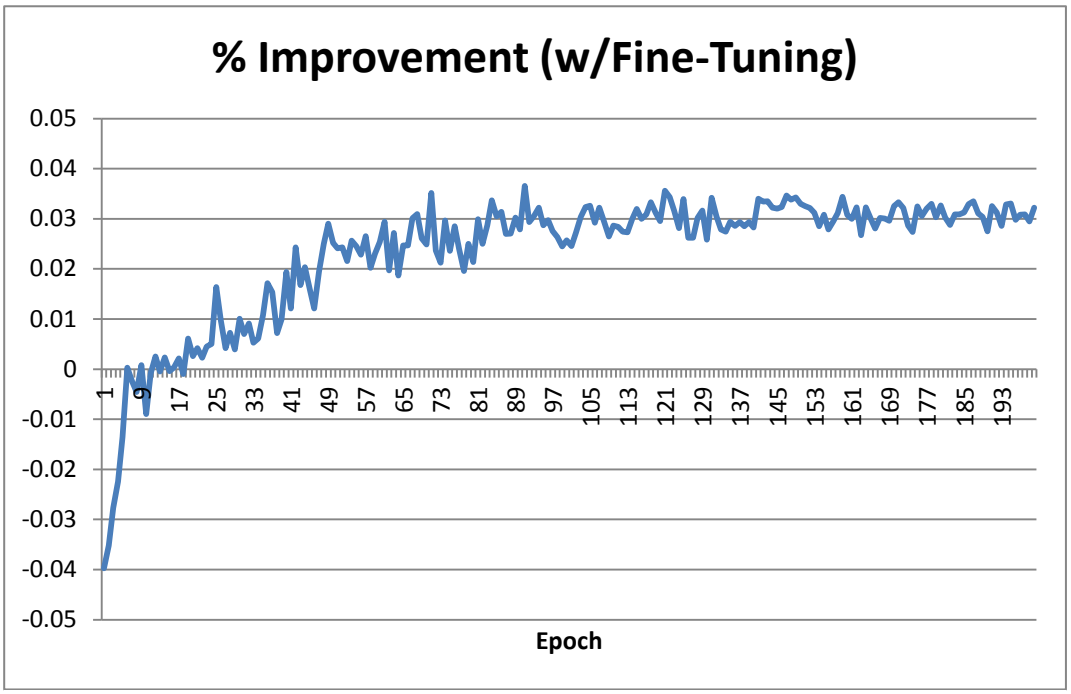

Figure 14: Validation Error Percent Improvement After Using Global Fine-Tuning

NORB

The NORB dataset was developed to provide a test for general 3D object recognition. 50 toys from 5 classification groups were photographed under various conditions. The categories are: four-legged animals, human figures, airplanes, trucks, and cars. Two cameras were used to provide stereo images under different lighting conditions, elevations and angles. The small NORB dataset contains fewer examples but each example is presented on a plain white background.

The data was presented to the system as floating point pixel "probabilities" between 0 and 1. To speed up training the second image of each stereo input was thrown out. This halves the size of the input and greatly speeds up training time. The RBM pretraining seemed to be ineffective as the

validation error started relatively higher after the first epoch and gradually declined. It's possible that the data was too complex or that raw pixel input was an improper method. Success as been had using raw pixel data on a similar learning devince known as a Deep Boltzmaan Machine. A higher learning rate used in that study may also have accounted for the improved pre-training and better classification results. (Salakhutdinov, 2009)

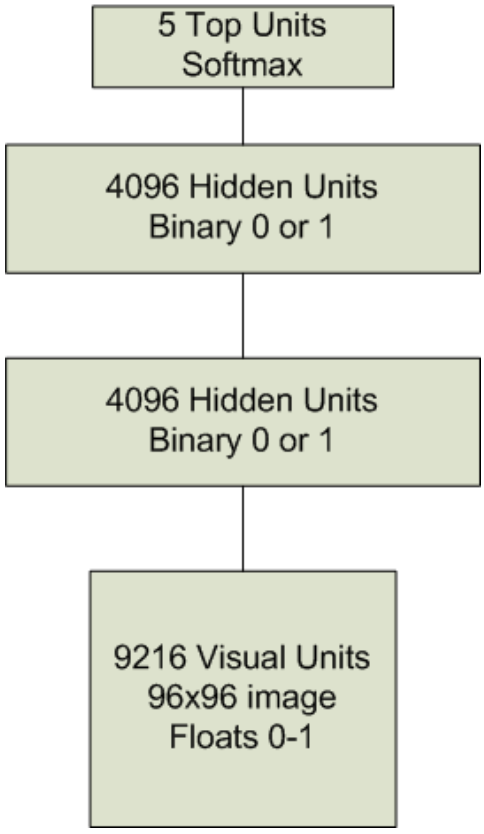

Figure 15: System Configuration for NORB

Table 13: Training Parameters for Level 1 RBM NORB

Epochs	500
Mini Batch Size	100
Gibbs Steps	1 (Persistent)
Learning Rate	0.001
Learning Rate Decay	0.0001 (last 100 epochs only)

Table 14: Training Parameters for Level 2 RBM NORB

Epochs	500
Mini Batch Size	100
Gibbs Steps	1 (Persistent)
Learning Rate	0.001
Learning Rate Decay	0.0001 (last 100 epochs only)

Table 15: Training Parameters for Global Fine-Tuning

Epochs	100
Mini Batch Size	100
Gibbs Steps	1
Learning Rate	0.001
Learning Rate Decay	0.0001

Table 16: Training Parameters for Neural Net

Epochs	200
Mini Batch Size	20
Learning Rate	0.0001
Learning Rate Decay	0.00005

Table 17: Classification Results on NORB Dataset

Data Source	**Misclassifications**	**Misclassification Rate**
Training Set	13683	23.6%
Validation Set	1527	24.0%
Test Set	15367	29.8%

Figure 16: Training Examples from NORB Dataset

CHAPTER 6
CONCLUSIONS

The system has clearly learned how to distinguish between the 5 classes. A single desktop GPU is capable of finding distinctions between 5 fairly similar types of objects. A deeper analysis of the results gives some insight into the actual applications of such a system and the original problem statement.

The advantage of harnessing the parallel power of GPUs was clearly shown in chapter 4. The final neural network is capable of doing 50 classifications in 192ms. This means, assuming a resolution of 96x96, the current system is capable of analyzing over 250 images a second. These images, of course, could also be image patches. This allows for much larger images to be analyzed if the system is trained on such data. For example, if the system was trained by slicing UAV images into equal sections it could analyze a single, large image by using slicing those same sized sections. Furthermore since a single desktop GPU is used it would be possible, and most likely cost effective, to have an array of such devices using the same pretrained parameters but analyzing different segments of the system.

The acquisition of training examples is another important consideration for a real life application. If a UAV needed to identify military vehicles, for example, data collection would be very straightforward. First, the initial RBM could be trained over image patches collected from a UAV, some containing such vehicles, with no manual input required. Subsequent layers and global fine-tuning would occur in the same manner presented in this paper. When it came time to train the neural net, a user would have to select image patches with objects of interest. This, presumably, would be a small percentage of the total number of patches since most of the image would not contain military vehicles.

It is also important to consider the causes of misclassification in the test sets. Domain specific difficulties account for a large portion of the error. Overall, animals had the highest miss rate. This, most likely is due to animals having the most variation in shape as the toys ranged from lions to elephants. As shown in the per class error, cars had a high miss rate when compared to trucks. Trucks had the lowest miss rate of all the classes, and a deeper looks reveals many of the misses for cars are due to the system classifying it as a truck. The system has some bias towards trucks which is partially due to the random subspace of initialization and would most likely be slightly alleviated but training the neural net step for longer.

Non-domain specific bias's should also be explored. The NORB dataset provides additional information about each training and test example. Included in this information is camera angle used. Both degrees above horizon and azimuth are given. As can be seen in figures 18 and 19 the camera angle seems to have no noticeable effect on misclassification rate. The best elevation seems to be around 50 degrees above the horizon, but is not significantly better than the lowest or highest elevations used. Azimuth is especially irrelevant because it relies on the way the object is pointed. Light level is also given. As expected the mid range levels were the easiest to classified. The darkest setting performed the worst by about 30% over the other levels. This could potentially be alleviated in a real system by performing more unsupervised training under poor lighting conditions. Of course, humans would also perform worse under poor lighting conditions so this result is not entirely unexpected.

The results show that a GPU implementation of a deep net provides a fast solution to general object recognition. No innate properties of the system make it impractical for most industry uses. However, it should also be noted that no innate properties of the system make it particularly good at finding patterns in

images as opposed to other input spaces. Specifically, nothing in the system makes use of the structure of the input space that comes with using images. Possible improvements could be used by integrating other techniques used for image recognition into the restricted Boltzmann machine or deep net. As is, the system performs fine on image tasks and may yet prove more useful in other domains.

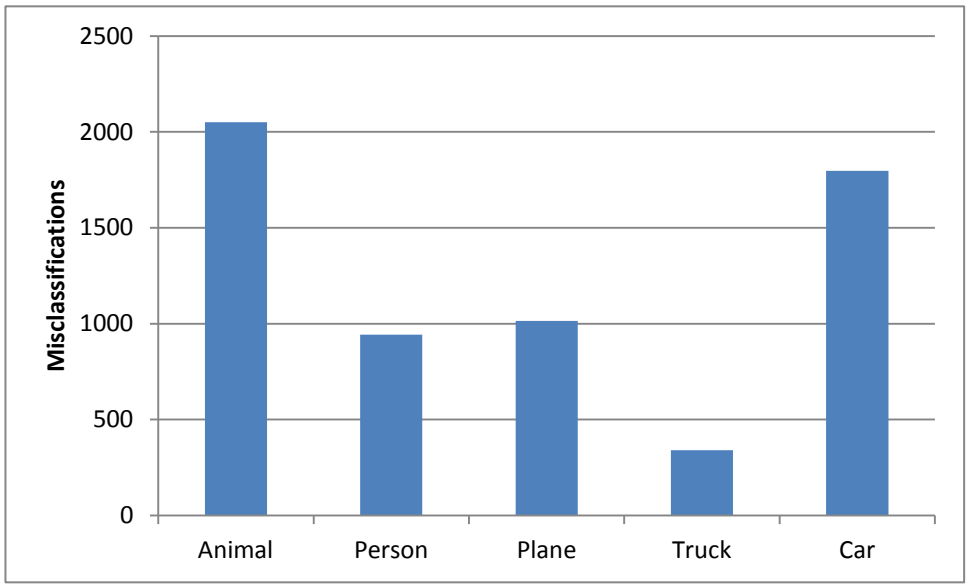

Figure 17: Miss Rate per Class

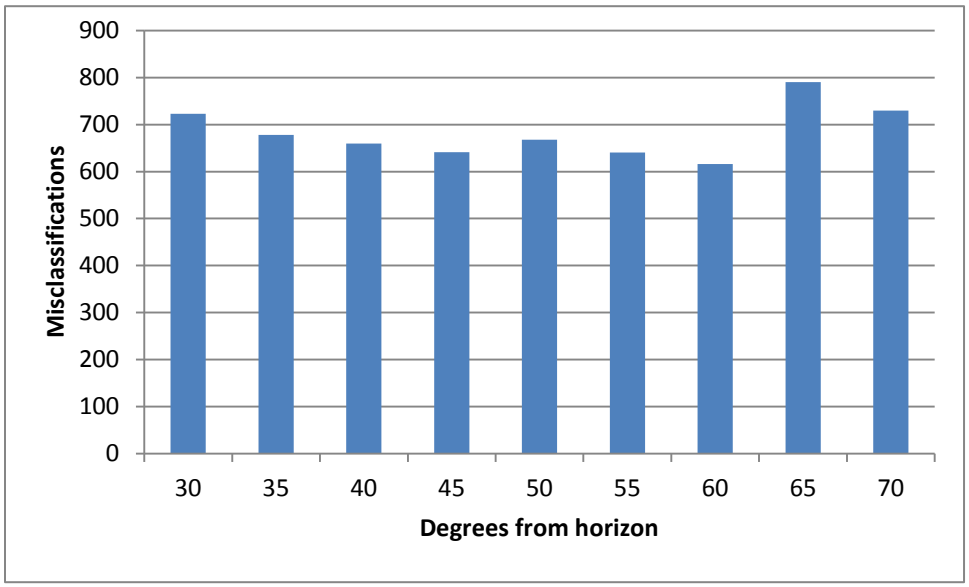

Figure 18: Miss Rate per Elevation

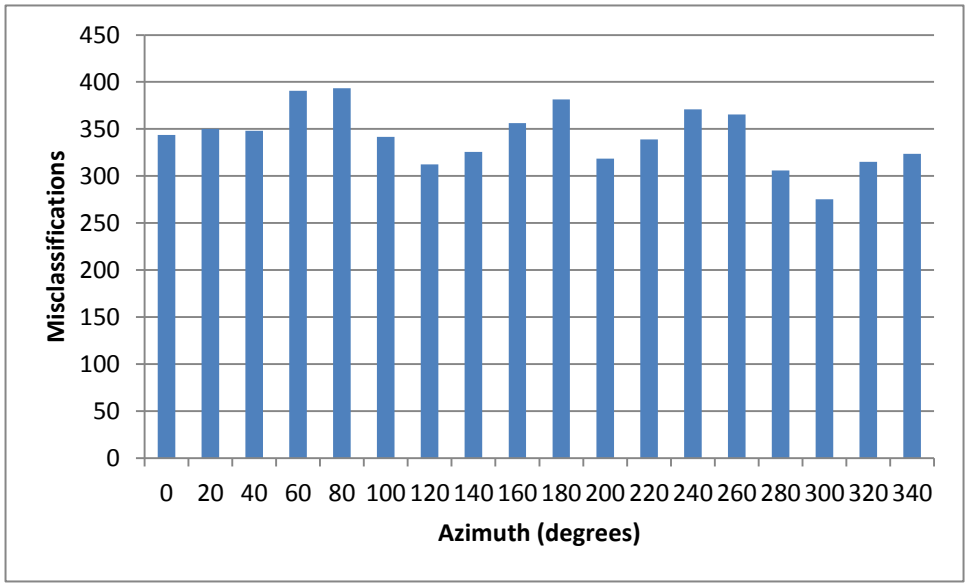

Figure 19: Miss Rate per Azimuth

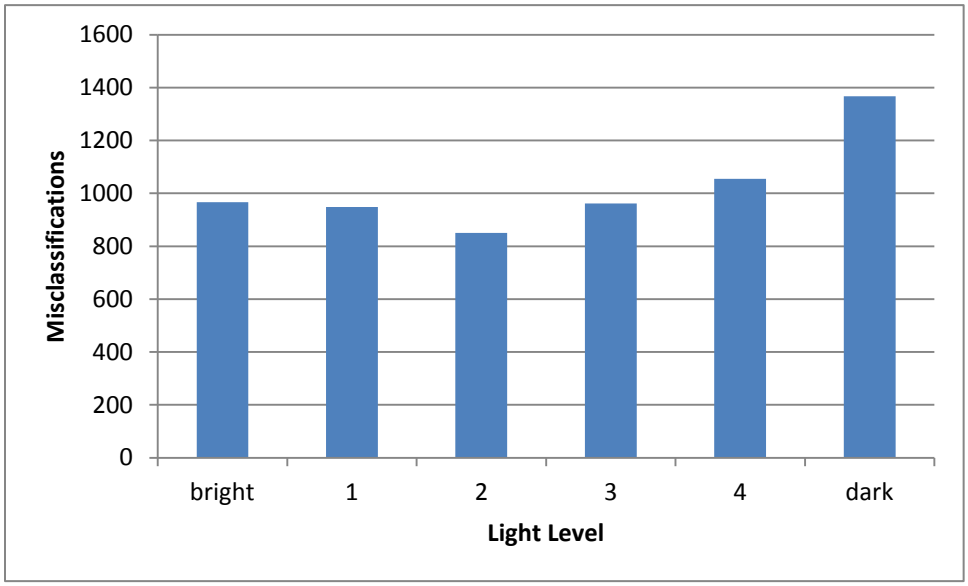

Figure 20: Miss Rate per Lighting Level

REFERENCES

Bishop, C. M. (2006). *Pattern recognition and machine learning* (Vol. 4, No. 4). New York: springer.

Carreira-Perpinan, M. A., & Hinton, G. E. (2005, January). On contrastive divergence learning. In *Artificial Intelligence and Statistics* (Vol. 2005, p. 17).

Hinton, G. (2010). A practical guide to training restricted Boltzmann machines.*Momentum*, *9*, 1.

Hinton, G. E. (2007). To recognize shapes, first learn to generate images.*Progress in brain research*, *165*, 535-547.

Hinton, G. E., Osindero, S., & Teh, Y. W. (2006). A fast learning algorithm for deep belief nets. *Neural computation*, *18*(7), 1527-1554.

Hinton, G. E. (2002). Training products of experts by minimizing contrastive divergence. *Neural computation*, *14*(8), 1771-1800.

Hinton, G. E., & Sejnowski, T. J. (1983, June). Optimal perceptual inference. In*Proceedings of the IEEE conference on Computer Vision and Pattern Recognition* (pp. 448-453). Piscataway, NJ: IEEE.

LeCun, Y., Chopra, S., Hadsell, R., Ranzato, M., & Huang, F. (2006). A tutorial on energy-based learning. *Predicting Structured Data*

LeCun, Y., Huang, F. J., & Bottou, L. (2004, June). Learning methods for generic object recognition with invariance to pose and lighting. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on* (Vol. 2, pp. II-97). IEEE.

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278-2324.
Ly, D. L., Paprotski, V., & Yen, D. (2008). *Neural networks on gpus: Restricted boltzmann machines*. Technical Report, Department of Electrical and Computer Engineering, University of Toronto.

Moody, J. E., Hanson, S. J., Krogh, A., & Hertz, J. A. (1995). A simple weight decay can improve generalization. *Advances in neural information processing systems*, *4*, 950-957.

NVIDIA. (2011, November) CUDA C Programming Guide.(Version 4.1)

Salakhutdinov, R., & Hinton, G. E. (2009). Deep boltzmann machines. In *Proceedings of the international conference on artificial intelligence and statistics* (Vol. 5, No. 2, pp. 448-455). Cambridge, MA: MIT Press.

Smolensky, P. (1986). Information processing in dynamical systems: Foundations of harmony theory.

Tieleman, T. (2008, July). Training restricted Boltzmann machines using approximations to the likelihood gradient. In *Proceedings of the 25th international conference on Machine learning* (pp. 1064-1071). ACM.

Walsh, B. (2004). Markov chain Monte Carlo and Gibbs Sampling.

APPENDIX A

VISUAL SYSTEM DISPLAYS

The histograms are all centered around 0.0 which is denoted by the largest vertical dash. The second largest dashes represent 1.0's and the smallest dashes denote 0.1.

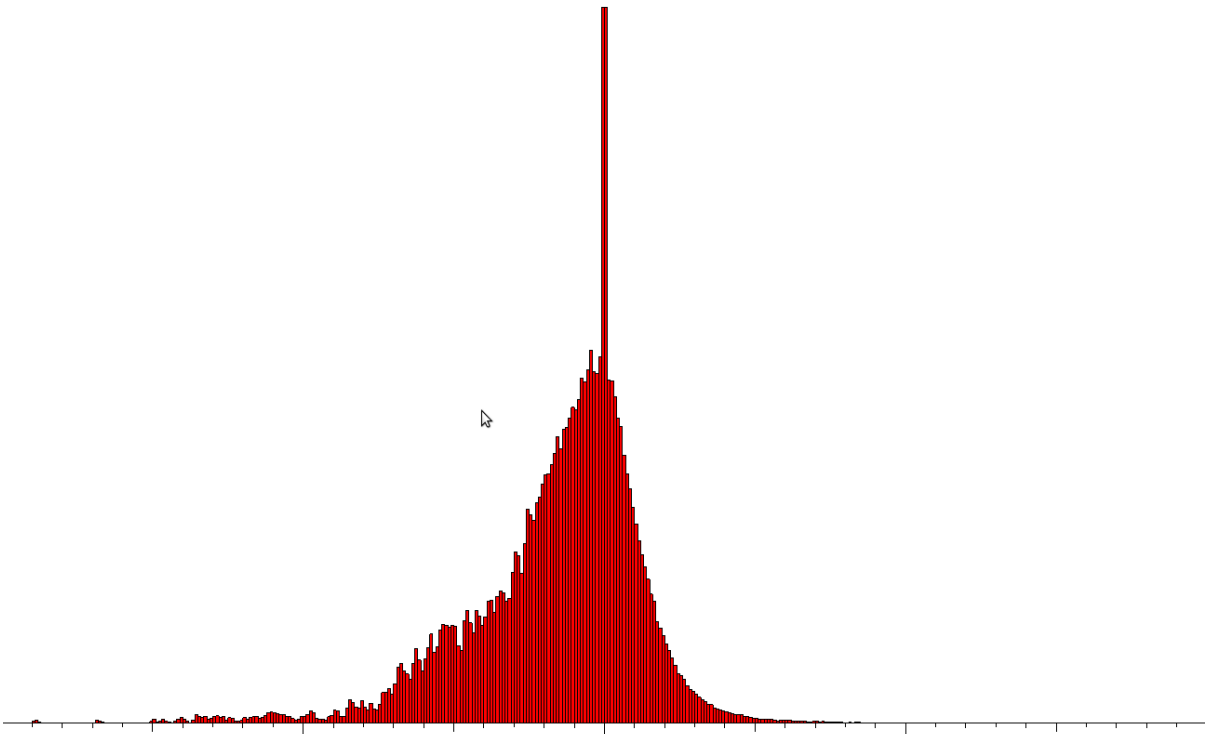

Figure A1: Red Histogram of Weight Values

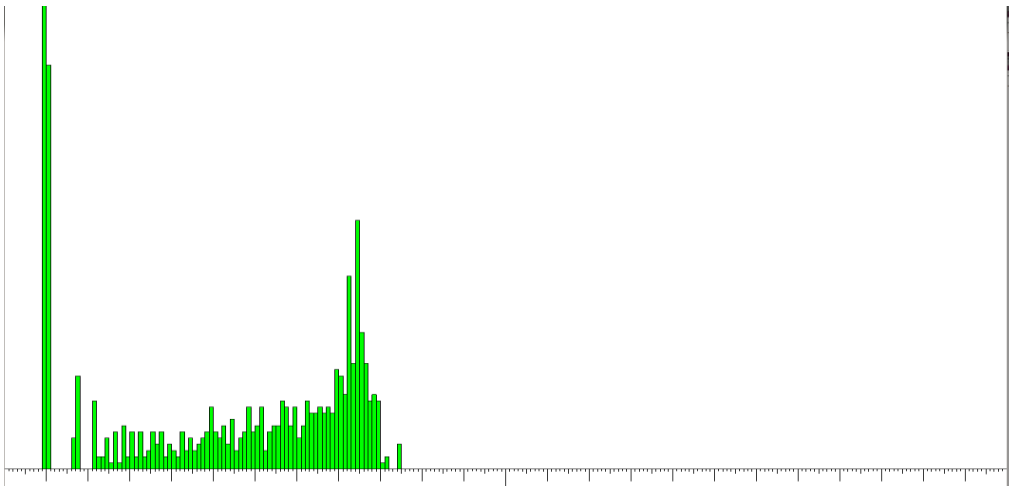

Figure A2: Green Histogram of Visual Bias

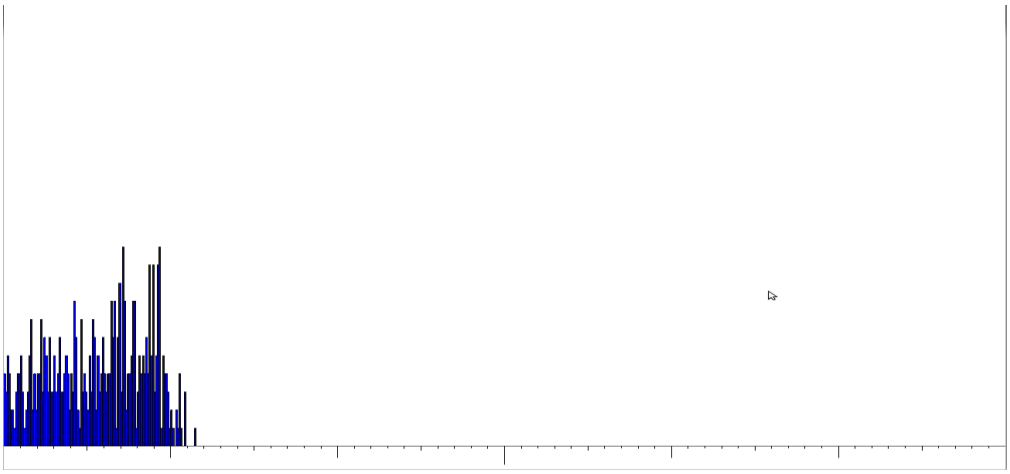

Figure A3: Blue Histogram of Hidden Bias

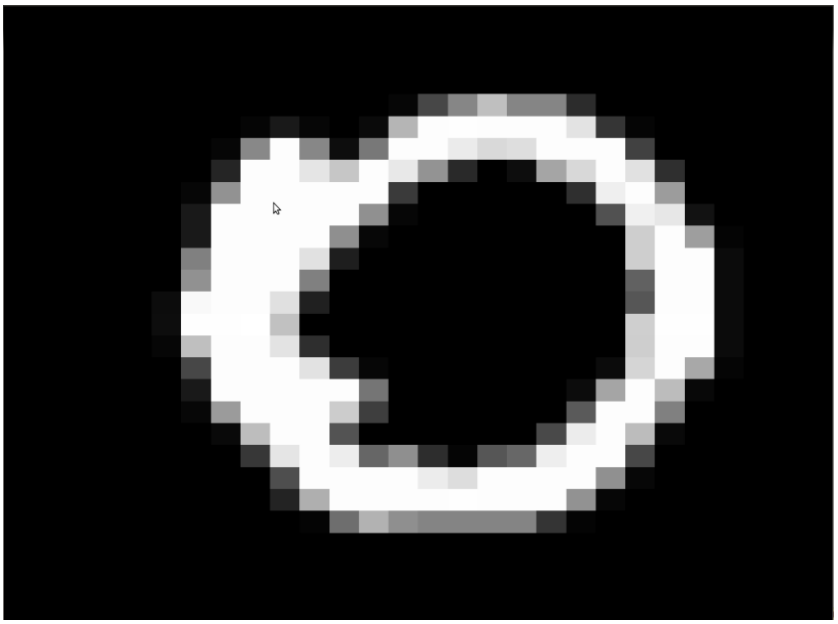

Figure A4: Training Data Example

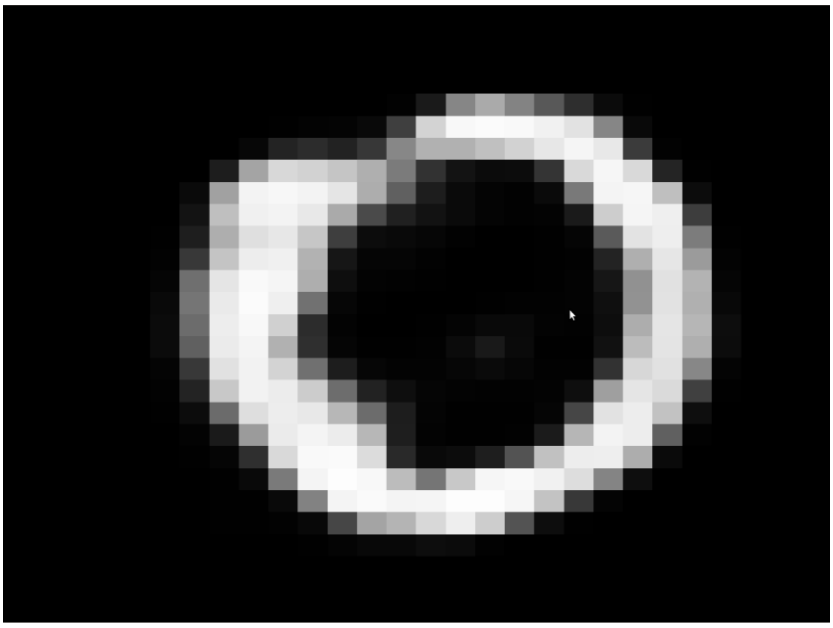

Figure A5: Reconstruction of Training Examples

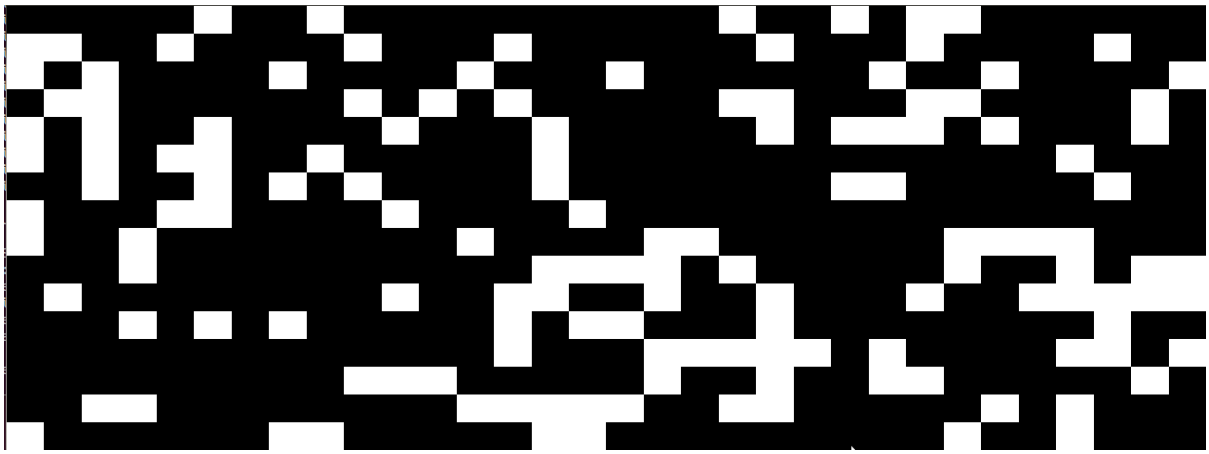

Figure A6: Visual Representation of Hidden Layer.

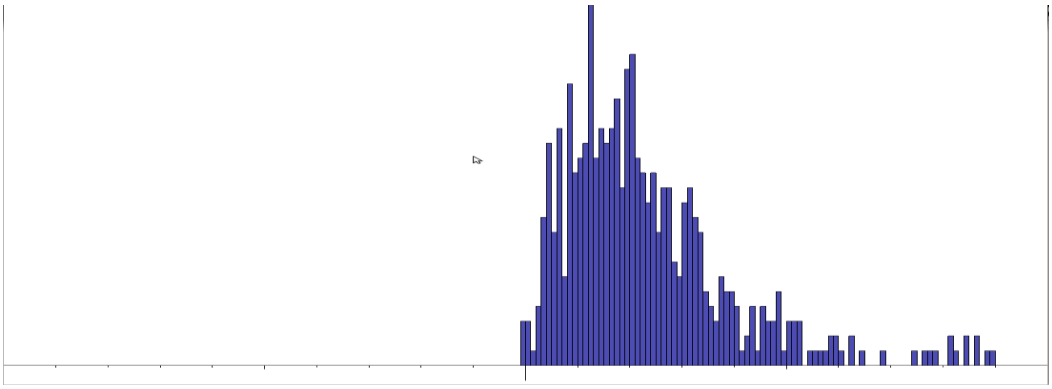

Figure A7: Histogram of Hidden Layer Probability Estimations

APPENDIX B

CODE

utilLearn.h

```
#ifndef UTIL_LEARN_H
#define UTIL_LEARN_H

#include <cuda.h>
#include <cuda_runtime.h>
#include "GL/gl.h"

namespace utilLearn{

   /* ----------------------------------------------------------------
-
      *          SWAP 4
      * Non-Intel -> Intel Byte formatting for 4 bytes
      * c | char* | pointer to 4 char array
      ----------------------------------------------------------------
----*/
   static void swap4(char* c)
   {
      char tmp[4];
      tmp[0] = c[3];
      tmp[1] = c[2];
      c[3] = c[0];
      c[2] = c[1];
      c[1] = tmp[1];
      c[0] = tmp[0];
      return;
   }

   /* ----------------------------------------------------------------
-
     *      DEV ALLOC
     * Helper function to allocate device memory. Helps keep code
clean
     * and keeps running count of allocated memory.
     *
     * d    | float**  | device location to allocate
     * size | int      | total bytes to allocate
     ----------------------------------------------------------------
--*/
   static void dev_alloc(float** d, int size)
   {
      cudaMalloc((void**)d,size);
      //_total_gpu_mem += size;
      printf("Allocating %f MBytes on GPU.\n",((float) size / 1024)
/ 1024);
```

```
      return;
   }
   static void dev_alloc(int** d, int size)
   {
      cudaMalloc((void**)d,size);
      //_total_gpu_mem += size;
      printf("Allocating %f MBytes on GPU.\n",((float) size / 1024)
/ 1024);
      return;
   }

   /* ------------------------------------------------------------
-
    *      SHOW
    * Draws the layer passed to it
    * lay  | float*| pointer to units to display
    * x    | int | width of layer
    * y    | int | height of layer
    ------------------------------------------------------------
--*/
   static void show(float* lay, int x, int y)
   {

         float px_size = 2.0/(float)x;
          for(int i=0;i<y;i++)
          {
               for(int j=0;j<x;j++)
               {
                    glColor3f(lay[i*x + j],lay[i*x + j],lay[i*x +
j]);
                    float v_off = 1.0-(float)(i+1)*px_size;
                    float h_off = -1.0+(float)j*px_size;
                    glBegin(GL_POLYGON);
                         glVertex2f(h_off, v_off);
                         glVertex2f(h_off, v_off + px_size);
                         glVertex2f(h_off + px_size, v_off +
px_size);
                         glVertex2f(h_off + px_size, v_off);
                    glEnd();
               }
          }
          return;
   }

   static void text(float* lay, int x, int y)
   {

          for(int i=0;i<y;i++)
          {
               for(int j=0;j<x;j++)
               {
                    printf("[%f]",lay[i*x + j]);
```

```
                }
                printf("\n");
            }
            return;
    }


}

#endif
```

connection.h

```
#ifndef CONNECTION_H
#define    CONNECTION_H

#include <fstream>
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <map>
#include <cuda.h>
#include <cuda_runtime.h>


#include "../../../0.Utils/utilLearn.h"

using namespace std;
using namespace utilLearn;

class Connection
{

public:
      Connection(int v_size, int h_size);
      ~Connection();

      //Initialization and saving
      void initParams();
      int save(ofstream *o_file, int loc);
      int load(ifstream *o_file, int loc);

      //Get
      int getVSize(){return _v_size;};
      float* getWTRow(int hidden_unit){return &_weight_t[hidden_unit
* _v_size];};

      //Set
      void setA(int index, float value){_a[index] = value;};
```

```
      void cpyA(){cudaMemcpy(d_a, _a, _v_size * sizeof(float),
cudaMemcpyHostToDevice);};
      void setB(int index, float value){_b[index] = value;};
      void cpyB(){cudaMemcpy(d_b, _b, _h_size * sizeof(float),
cudaMemcpyHostToDevice);};

      //Print
      void printW();
      void printWT();

      //Display (connection_disp.cpp)
      void histogramW();
      void histogramA();
      void histogramB();
      void histogramDw();

public:
      float*          d_a;  //bias to visible unit
      float*          d_b;  //bias to hidden unit
      float*          d_weight;
      float*          d_weight_t; //transposed weights
      float*          d_vel_weight; //velocity of weight updates
      float*          d_dw;

private:
      int   _v_size;
      int         _h_size;
      int         _w_size;

      float*          _a;
      float*          _b;
      float*          _weight;
      float*          _weight_t;
      float*          _vel_weight;//velocity of weights
      float*          _dw;
};

#endif
```

connection.cpp

```
#include "../inc/connection.h"

Connection::Connection(int v_size, int h_size)
{
      _v_size = v_size;
      _h_size = h_size;
      _w_size = _v_size * _h_size;

      //Host memory allocation
      _weight = (float*) malloc(_w_size * sizeof(float));
```

```
	_weight_t = (float*) malloc(_w_size * sizeof(float));
	_a = (float*) malloc(_v_size * sizeof(float));
	_b = (float*) malloc(_h_size * sizeof(float));
	_vel_weight = (float*)malloc(_w_size*sizeof(float));
	_dw = (float*)malloc(_w_size*sizeof(float));

	//Device memory allocation
	dev_alloc(&d_weight,_w_size *sizeof(float));
	dev_alloc(&d_weight_t,_w_size *sizeof(float));
	dev_alloc(&d_a, _v_size * sizeof(float));
	dev_alloc(&d_b, _h_size * sizeof(float));
	dev_alloc(&d_vel_weight,_w_size *sizeof(float));
	dev_alloc(&d_dw,_w_size *sizeof(float));

	return;
}

/* ------------------------------------------------------------------
 *         DESTRUCTOR
 ------------------------------------------------------------------
*/
Connection::~Connection()
{
	//Host memory
	free(_weight);
	free(_weight_t);
	free(_a);
	free(_b);
	free(_vel_weight);
	free(_dw);

	//Device memory
	cudaFree(d_weight);
	cudaFree(d_weight_t);
	cudaFree(d_a);
	cudaFree(d_b);
	cudaFree(d_dw);
	cudaFree(d_vel_weight);
}

void Connection::initParams()
{
	//Setup Weights | i->j | Visible 0 -> Hidden 1
		srand((unsigned)time(0));
		//Approximate Gaussian u=0 z=0.01
		for (int i=0;i<_w_size;i++)
		{
			float central = 0;
			for(int c=0;c<100;c++)
			{
				float u = (float)rand() / (float)RAND_MAX;
				central+= (2*u -1)*0.1;
```

```
                }
                central /= 100;
                _weight[i] = central;
          }

          cudaMemcpy(d_weight, _weight, _w_size *
sizeof(float),cudaMemcpyHostToDevice);
          //free(wij);

          //Hidden Bias = -2 to encourage sparsity
          for (int j=0;j<_h_size;j++)
          {
                _b[j] = -2.0;
          }

          //Visible Bias = set to 0
          for (int i=0;i<_v_size;i++)
          {
                _a[i] = 0;
                //printf("Pi=%f | ai[%d])=%f\n",Pi,i,log(Pi / (1-
Pi)));
          }

          //Places on device
          cudaMemcpy(d_a, _a,_v_size *
sizeof(float),cudaMemcpyHostToDevice);
          cudaMemcpy(d_b, _b,_h_size *
sizeof(float),cudaMemcpyHostToDevice);


          //Initialize velocities to 0
          memset(_vel_weight,0.0,_w_size*sizeof(float));
          cudaMemcpy(d_vel_weight, _vel_weight, _w_size *
sizeof(float), cudaMemcpyHostToDevice);


}

int Connection::save(ofstream *o_file, int loc)
{
     //place data on host
     cudaMemcpy(_weight, d_weight, _w_size *
sizeof(float),cudaMemcpyDeviceToHost);
     cudaMemcpy(_weight_t, d_weight_t, _w_size *
sizeof(float),cudaMemcpyDeviceToHost);
     cudaMemcpy(_a, d_a,_v_size *
sizeof(float),cudaMemcpyDeviceToHost);
     cudaMemcpy(_b, d_b,_h_size *
sizeof(float),cudaMemcpyDeviceToHost);
     cudaMemcpy(_vel_weight, d_vel_weight, _w_size * sizeof(float),
cudaMemcpyDeviceToHost);
```

```
	o_file->write((char*)_weight,_w_size * sizeof(float));
	loc += _w_size * sizeof(float);
	o_file->write((char*)_weight_t,_w_size * sizeof(float));
	loc += _w_size * sizeof(float);
	o_file->seekp(loc);
	o_file->write((char*)_a,_v_size * sizeof(float));
	loc += _v_size * sizeof(float);
	o_file->seekp(loc);
	o_file->write((char*)_b,_h_size * sizeof(float));
	loc += _h_size * sizeof(float);
	o_file->seekp(loc);
	o_file->write((char*)_vel_weight,_w_size * sizeof(float));
	loc += _w_size * sizeof(float);

	return loc;
}

int Connection::load(ifstream *i_file, int loc)
{
	//load weights
	for(int n=0;n<_w_size;n++)
	{
		i_file->seekg(loc);
		i_file->read((char*)&_weight[n],sizeof(float));
		loc += sizeof(float);
		//printf("w[%d]=%f ",n,w[n]);
	}
	//load transposed weights
	for(int n=0;n<_w_size;n++)
	{
		i_file->seekg(loc);
		i_file->read((char*)&_weight_t[n],sizeof(float));
		loc += sizeof(float);
		//printf("w[%d]=%f ",n,w[n]);
	}
	//load a
	for(int i=0;i<_v_size;i++)
	{
		i_file->seekg(loc);
		i_file->read((char*)&_a[i],sizeof(float));
		loc += sizeof(float);
		//printf("a[%d]=%f\t",i,a[i]);
	}
	printf("\n");
	//load b
	for(int j=0;j<_h_size;j++)
	{
		i_file->seekg(loc);
		i_file->read((char*)&_b[j],sizeof(float));
		loc += sizeof(float);
		//printf("b[%d]=%f\t",j,b[j]);
	}
```

```
	for(int v=0;v<_w_size;v++)
	{
		i_file->seekg(loc);
		i_file->read((char*)&_vel_weight[v],sizeof(float));
		loc += sizeof(float);
		//printf("vel[%d]=%f ",v,_vel_weight[v]);
	}
	cudaMemcpy(d_weight, _weight, _w_size *
sizeof(float),cudaMemcpyHostToDevice);
	cudaMemcpy(d_weight_t, _weight_t, _w_size *
sizeof(float),cudaMemcpyHostToDevice);
	cudaMemcpy(d_a, _a,_v_size *
sizeof(float),cudaMemcpyHostToDevice);
	cudaMemcpy(d_b, _b,_h_size *
sizeof(float),cudaMemcpyHostToDevice);
	cudaMemcpy(d_vel_weight, _vel_weight, _w_size *
sizeof(float),cudaMemcpyHostToDevice);

	return loc;
}

/* ------------------------------------------------------------------
 *         PRINT W
 * Prints the weights in matrix form
 *
 ------------------------------------------------------------------
*/
void Connection::printW()
{
	float* dubs = (float*)malloc(_w_size*sizeof(float));
	cudaMemcpy(dubs, d_weight,
_w_size*sizeof(float),cudaMemcpyDeviceToHost);

	printf("\n");
	//top left
	/*for (int i=0;i<10;i++)
	{
		for (int j=0;j<10;j++)
		{
			printf("[%.3f]",dubs[i*_h.size + j]*10);
		}
		printf("\n");
	}*/

	//bottom right
	for (int i=_v_size - 10;i<_v_size;i++)
	{
		for (int j=_h_size - 10;j<_h_size;j++)
		{
			printf("[%.3f]",dubs[i*_h_size + j]*10);
		}
		printf("\n");
```

```
	}

	printf("\n");
}

/* ------------------------------------------------------------------
 *         PRINT WT
 * Prints the weights transposed in matrix form
 *
 ------------------------------------------------------------------
*/
void Connection::printWT()
{
	float* dubsT = (float*)malloc(_w_size*sizeof(float));
	cudaMemcpy(dubsT, d_weight_t,
_w_size*sizeof(float),cudaMemcpyDeviceToHost);

	printf("\n");
	//top left
	/*for (int i=0;i<10;i++)
	{
		for (int j=0;j<10;j++)
		{
			printf("[%.3f]",dubsT[i*_v.size + j]*10);
		}
		printf("\n");
	}*/

	//bottom right
	for (int i=_h_size - 10;i<_h_size;i++)
	{
		for (int j=_v_size - 10;j<_v_size;j++)
		{
			printf("[%.3f]",dubsT[i*_v_size + j]*10);
		}
		printf("\n");
	}

	printf("\n");
}
```

connection_disp.cpp

```
#include "GL/freeglut.h"
#include "GL/gl.h"

#include <histo.h>

#include "../inc/connection.h"
```

```
void Connection::histogramW()
{
	Color tmp;
	tmp.r = 1.0;
	tmp.g = 0.0;
	tmp.b = 0.0;


	cudaMemcpy(_weight, d_weight, _w_size*sizeof(float),
cudaMemcpyDeviceToHost);
	Histo h(_weight,_w_size,2,tmp, 2.0);

	h.drawStatic();

}

void Connection::histogramDw()
{
	Color tmp;
	tmp.r = 0.7;
	tmp.g = 0.3;
	tmp.b = 0.3;


	cudaMemcpy(_dw, d_dw, _w_size*sizeof(float),
cudaMemcpyDeviceToHost);
	Histo h(_dw,_w_size,7,tmp,0.01);

	h.drawStatic();

}

void Connection::histogramA()
{
	Color tmp;
	tmp.r = 0.0;
	tmp.g = 1.0;
	tmp.b = 0.0;

	cudaMemcpy(_a, d_a, _v_size*sizeof(float),
cudaMemcpyDeviceToHost);
	Histo h(_a,_v_size,1,tmp,12.0);

	h.drawStatic();

}

void Connection::histogramB()
{
	Color tmp;
	tmp.r = 0.0;
	tmp.g = 0.0;
```

```
      tmp.b = 1.0;

      cudaMemcpy(_b, d_b, _h_size*sizeof(float),
cudaMemcpyDeviceToHost);
      Histo h(_b,_h_size,2,tmp,3.0);

      h.drawStatic();

}
```

layer.h

```
#ifndef LAYER_H
#define LAYER_H

#include <fstream>
#include <iostream>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <map>
#include <cuda.h>
#include <cuda_runtime.h>

#include "../../../0.Utils/utilLearn.h"

using namespace std;
using namespace utilLearn;

class Layer
{

public:
      Layer(int dim_x, int dim_y, int mini_batch_size, int
gibbs_samples, bool using_initial);
      ~Layer();
      void initParams();

      void randState(float prob);

      //Host Gets
      int getSize(){return _size;};

      //Device Gets
      int saveState(ofstream* out_file, int loc);
      int saveDim(ofstream* out_file, int loc);
      int saveQ(ofstream* out_file, int loc);
      int loadQ(ifstream* in_file, int loc);

      //Display Functions
```

```
	void showState(bool current, int b, int g);
	void printState(bool current, int b, int g);
	void projectWeights(float* weights);
	void histogramQ();

	void checkSparsity();

public:
	float* d_initial_state;
	float* d_state;
	float* d_rand; //random numbers used for monte carlo
	float* d_q; //sparsity estimation

	float* d_error; //error given input

protected:
	int _dim_x;
	int _dim_y;
	int _size;
	int _mini_batch_size;
	int _gibbs_samples;

	float*	_q; //probability estimate

	bool _using_initial; //set to true if the layers holds a state

	//display array
	float*	_disp;

};

#endif
```

layer.cpp

```
#include "../inc/layer.h"

Layer::Layer(int dim_x, int dim_y, int mini_batch_size, int
gibbs_samples, bool using_initial)
{
	_dim_x = dim_x;
	_dim_y = dim_y;
	_size = _dim_x * _dim_y;

	_mini_batch_size = mini_batch_size;
	_gibbs_samples = gibbs_samples;
	_using_initial = using_initial;

	//Host memory allocation
	_disp = (float*) malloc(_size * sizeof(float));
	_q = (float*)malloc(_size*sizeof(float));
```

```
	if(_using_initial)
		dev_alloc(&d_initial_state, _size * _mini_batch_size *
gibbs_samples * sizeof(float));

	dev_alloc(&d_state, _size * _mini_batch_size * gibbs_samples *
sizeof(float));
	dev_alloc(&d_error, _size * _mini_batch_size * gibbs_samples *
sizeof(float));
	dev_alloc(&d_rand, _size * mini_batch_size * gibbs_samples *
sizeof(float));
	dev_alloc(&d_q,_size *sizeof(float));

	return;
}

Layer::~Layer()
{
	//Host memory
	free(_disp);
	free(_q);

	//Device memory
	if(_using_initial)
		cudaFree(d_initial_state);

	cudaFree(d_state);
	cudaFree(d_rand);
	cudaFree(d_q);

	return;
}

void Layer::initParams()
{
	//Initialize activation probablity estimation to 0
	memset(_q,0.0,_size*sizeof(float));
	cudaMemcpy(d_q, _q, _size * sizeof(float),
cudaMemcpyHostToDevice);

	return;
}

int Layer::saveQ(ofstream* out_file, int loc)
{
	//place data on host
	cudaMemcpy(_q, d_q, _size * sizeof(float),
cudaMemcpyDeviceToHost);

	out_file->seekp(loc);
	out_file->write((char*)_q, _size * sizeof(float));
```

```
	loc+=_size * sizeof(float);
	return loc;
}

int Layer::loadQ(ifstream* in_file, int loc)
{
	for(int q=0;q<_size;q++)
	{
		in_file->seekg(loc);
		in_file->read((char*)&_q[q],sizeof(float));
		loc += sizeof(float);
		//printf("q[%d]=%f\t",q,_q_h[q]);
	}
	cudaMemcpy(d_q, _q,_size *
sizeof(float),cudaMemcpyHostToDevice);

	return loc;
}


/* ---------------------------------------------------------------
 *		RAND HIDDEN
 * Sets d_state(0,0) to random state w/ probability
 * prob	|	float |	probability of unit being on
 ---------------------------------------------------------------
*/
void Layer::randState(float prob)
{
	float tmp;
	for(int i=0;i<_size;i++)
	{
		tmp = rand() % 100;
		_disp[i] = (tmp < (prob*100)); //set target sparsity here
	}
	cudaMemcpy(d_state,_disp, _size * sizeof(float),
cudaMemcpyHostToDevice);
	return;
}

int Layer::saveState(ofstream* out_file, int loc)
{
	cudaMemcpy(_disp, d_state,_size *
sizeof(float),cudaMemcpyDeviceToHost);

	out_file->seekp(loc);
	out_file->write((char*)_disp,_size * sizeof(float));
	return (loc + (_size * sizeof(float)));
}

int Layer::saveDim(ofstream* out_file, int loc)
{
	//Save Hidden layer dimensions
```

```
	out_file->seekp(loc);
	out_file->write((char*)&_dim_x, sizeof(int));

	loc += sizeof(int);

	out_file->seekp(loc);
	out_file->write((char*)&_dim_y, sizeof(int));

	loc += sizeof(int);

	return loc;
}

void Layer::checkSparsity()
{
	cudaMemcpy(_q, d_q,
_size*sizeof(float),cudaMemcpyDeviceToHost);

	float sum=0;
	float max=0;
	float min=1;
	for(int j=0;j<_size;j++)
	{
		if(_q[j] > max)
			max = _q[j];
		if(_q[j] < min)
			min = _q[j];
		sum+= _q[j];
	}

	printf("Avg q_h:%f\t",sum/(float)_size);
	printf("Min q_h:%f\t",min);
	printf("Max q_h:%f\n",max);

}
```

layer_disp.cpp

```
#include "GL/freeglut.h"
#include "GL/gl.h"

#include <histo.h>

#include "../inc/layer.h"

/* ----------------------------------------------------------------
 *         SHOW HIDDENX
 * Draws the selected state of the layer
 *
 * final | bool | t=current state, f=initial state
 * b | int | layer batch
```

```
 * g | int | layer fantasy particle
 ---------------------------------------------------------------------
*/
void Layer::showState(bool current, int b, int g)
{
      if(b>=_mini_batch_size || b<0)
      {
            printf("Selected batch is out of range\n");
            return;
      }
      else
      {
            if(current)
            {
                  cudaMemcpy(_disp,&d_state[b*_gibbs_samples*_size +
g*_size],_size * sizeof(float), cudaMemcpyDeviceToHost);
                  return show( _disp,_dim_x,_dim_y);
            }
            else
            {
                  if(!_using_initial)
                  {
                        printf("Attempting to view unused initial
state\n");
                        return;
                  }
                  else
                  {

      cudaMemcpy(_disp,&d_initial_state[b*_gibbs_samples*_size +
g*_size],_size * sizeof(float), cudaMemcpyDeviceToHost);
                        return show( _disp,_dim_x,_dim_y);
                  }
            }

      }
}

void Layer::printState(bool current, int b, int g)
{
      if(b>=_mini_batch_size || b<0)
      {
            printf("Selected batch is out of range\n");
            return;
      }
      else
      {
            if(current)
            {
                  cudaMemcpy(_disp,&d_state[b*_gibbs_samples*_size +
g*_size],_size * sizeof(float), cudaMemcpyDeviceToHost);
                  return text( _disp,_dim_x,_dim_y);
```

```
            }
            else
            {
                  if(!_using_initial)
                  {
                        printf("Attempting to view unused initial
state\n");
                        return;
                  }
                  else
                  {

      cudaMemcpy(_disp,&d_initial_state[b*_gibbs_samples*_size +
g*_size],_size * sizeof(float), cudaMemcpyDeviceToHost);
                        return text( _disp,_dim_x,_dim_y);
                  }
            }

      }
}

void Layer::projectWeights(float* weights)
{
      memcpy(_disp,weights, _size * sizeof(float));
      for(int i=0;i< _size;i++)
            _disp[i] = _disp[i]/2.0 + 0.5;
      return show( _disp,_dim_x,_dim_y);
}


void Layer::histogramQ()
{
      Color tmp;
      tmp.r = 0.3;
      tmp.g = 0.3;
      tmp.b = 0.7;


      cudaMemcpy(_q, d_q, _size*sizeof(float),
cudaMemcpyDeviceToHost);
      Histo h(_q,_size,2,tmp, 1.0);

      h.drawStatic();

}
```

trainer.h

```
#ifndef TRAINER_H
#define TRAINER_H
```

```
#include <fstream>
#include <iostream>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include "../../../0.Utils/utilLearn.h"

using namespace std;
using namespace utilLearn;

class Trainer
{
public:
      Trainer(int mini_batch_size, int gibbs_samples, int
num_epochs, float learn_rate, int num_classes, float momentum, float
lr_beta);
      ~Trainer();

      //Training Status
      void incN();
      int getN(){return _n;};
      bool epochComplete();
    int getEpoch(){return _cur_epoch;};
    bool trainComplete(){return _cur_epoch >= _num_epochs;};
    float getMomentum(){return _momentum;};

    //Providing Training Data
      void randBatchV();
      int nextBatchTrain();
      int nextBatchValid();
      void setV(int n, int batch);
    int getTrainSize(){return _train_size;};
    int getValidSize(){return _valid_size;};
      float getLearnRate(){return _learn_rate;};
      int getNumFantasy(){return _mini_batch_size *
_gibbs_samples;};
      void showTraining(int n);
      void showCurrent(int b);
      float* getHostData(){return _train_data;};


      int answer(int n){return _train_label_vals[n];};
      int ansCurrent(int b){return _mini_batch_label_vals[b];};

      int batchClassification(float* d_top_prob, int batch_size);
      float batchError(float* d_top_prob, int batch_size);

      //Training Data Calculations
      float pixelProb(int index);

    //Loading
```

```
      int loadTrainingData(char* data_file);
      int loadTrainingLabels(char* label_file);
      int loadTrainingLabelsMAT(char* label_file);
      int loadConvertTrainingData(char* data_file);
      int loadTrainingDataMAT(char* data_file);

public:
    //GPU device memory
    float* d_mini_batch_data;
    float* d_mini_batch_labels;

private:
    //Set validation hold aside
    void update_valid(){_valid_size = floor((float)_train_size *
0.1);};


protected:
    //Training Status Variables
      int        _mini_batch_size;
      int        _gibbs_samples;
      int        _num_epochs;     //max epochs to train on
      int        _cur_epoch;      //current number of epochs
      int        _cur_batch; //current batch loaded (in order)
      int        _next_batch; //next batch to be loaded
      int        _n;        //training examples seen this epoch
      float _learn_rate;
      bool  _using_labels; //set when loaded
      float      _momentum;
      float      _lr_beta;

      //set by loadTrainingData
      int        _input_size;
      int        _input_dim_x;
      int        _input_dim_y;
      int        _train_size;
      int        _valid_size; //Number held aside for validation --
10% by default
      int        _num_classes;

      float*          _train_data; //all training data
      float*          _mini_batch_data;
      int*       _train_label_vals; //all training labels
      float*          _train_labels;
      float*          _mini_batch_labels;
      int*       _mini_batch_label_vals;
};

#endif
```

trainer.cpp

```
#include "../inc/trainer.h"

/* ----------------------------------------------------------------
 *         CONSTRUCTOR
 * mini_batch_size    | int | number of examples per batch
 * gibbs_samples| int | number of fantasy particles
 * num_epochs         | int | total epochs to train over
 * learn_rate         | float | the learning rate being used
 ----------------------------------------------------------------
*/
Trainer::Trainer(int mini_batch_size, int gibbs_samples, int
num_epochs, float learn_rate, int num_classes, float momentum, float
lr_beta)
{
       _mini_batch_size = mini_batch_size;
       _gibbs_samples = gibbs_samples;
       _num_epochs = num_epochs;
       _cur_epoch = 0;
       _cur_batch = 0;
       _next_batch = 0;
       _n = 0;
       _learn_rate = learn_rate;
       _num_classes = num_classes;
       _using_labels = false;
       _momentum = momentum;
       _lr_beta = lr_beta;

       _input_size = 1;
       _input_dim_x = 1;
       _input_dim_y = 1;
}

/* ----------------------------------------------------------------
 *         LOAD TRAINING DATA
 * Loads in the training examples from file
 *
 * data_file | char* | pointer to name of file
 ----------------------------------------------------------------
*/
int Trainer::loadTrainingData(char* data_file)
{
           ifstream i_img_train;
           i_img_train.open(data_file, ios::binary);

           if(i_img_train.is_open())
           {
                 printf("Loading Images...\n");
                 //MAGIC NUMBER
                 char c_magic_number[4];
                 int i_magic_number;
```

```
                i_img_train.read(c_magic_number,4);
                swap4(c_magic_number);
                memcpy(&i_magic_number,c_magic_number,4);
                printf("magic number: %d\n",i_magic_number);

                //# IMAGES
                char c_images[4];
                int i_images;
                i_img_train.seekg(4);
                i_img_train.read(c_images,4);
                swap4(c_images);
                memcpy(&i_images,c_images,4);
                printf("images: %d\n",i_images);
                _train_size = i_images; //Set trianing size
                update_valid();

                //ROWS
                char c_rows[4];
                int i_rows;
                i_img_train.seekg(8);
                i_img_train.read(c_rows,4);
                swap4(c_rows);
                memcpy(&i_rows,c_rows,4);
                printf("rows: %d\n",i_rows);
                _input_dim_x = i_rows;

                //COLUMNS
                char c_cols[4];
                int i_cols;
                i_img_train.seekg(12);
                i_img_train.read(c_cols,4);
                swap4(c_cols);
                memcpy(&i_cols,c_cols,4);
                printf("columns: %d\n",i_cols);
                _input_dim_y = i_cols;

                //SET INPUT SIZE
                _input_size = _input_dim_y * _input_dim_x;
                printf("Data items are size:%d\n",_input_size);

                //Allocate memory
                _mini_batch_data = (float*)malloc(_input_size *
_train_size * sizeof(float));
                dev_alloc(&d_mini_batch_data, _input_size *
_mini_batch_size * sizeof(float));
           }
           else
           {
                printf("Couldn't open file. Exiting...");
                return -1;
           }
           //Grab Images
```

```
            int input_ptr = 16;
            char* c_img;
            c_img = (char*)malloc(_input_size*sizeof(char));

            //Allocate Training Data Space
            _train_data = (float*)malloc(_train_size * _input_size *
sizeof(float));
            printf("Allocated host memory for training data\n");

            //Image #
            for(int n=0;n<_train_size;n++)
            {
                  i_img_train.seekg(input_ptr);
                  i_img_train.read(c_img,_input_size);
                  input_ptr+=_input_size;
                  //Down Column
                  //printf("read in");
                  for(int i=0;i<_input_size;i++)
                  {
                        //convert to 'binary'
                        /*(if (c_img[i] != 0x00)
                              c_img[i] = 0x01;*/
                        //_train_data[ n*_v_size + i] =
(float)c_img[i];
                        int tmp;
                        memcpy(&tmp,&c_img[i],1);

                        //memcpy(&_train_data[ n*_v_size + i*28 + j
],&c_img[i*28 + j],1);
                        //printf("[%d]",tmp);


                        /*//memcpy(&data[ n*V_SIZE + i*28 + j
],&c_img[i*28 + j],1);
                        if(tmp > 20)
                              _train_data[ n*_input_size + i] = 1.0f;
                        else
                              _train_data[ n*_input_size + i] =
0.0f;*/

                        _train_data[n*_input_size + i] =
(float)tmp/255;

                  }
                  //printf("copied");
                  //TEST PRINT
                  //print_img(&data[n*V_SIZE], 28, 28);
            }

            i_img_train.close();
```

```
            //free mem
            free(c_img);

            printf("Data Load Complete!\n");
            return 0;
}

int Trainer::loadTrainingDataMAT(char* data_file)
{

      ifstream i_img_train;
      i_img_train.open(data_file, ios::binary);

      if(i_img_train.is_open())
      {
            printf("Loading Images...\n");
            //MAGIC NUMBER
            char c_magic_number[4];
            int i_magic_number;
            i_img_train.read(c_magic_number,4);
            //swap4(c_magic_number);
            memcpy(&i_magic_number,c_magic_number,4);
            printf("magic number: %d\n",i_magic_number);

            //# NUMBER OF DIMENSIONS
            char c_ndim[4];
            int i_ndim;
            i_img_train.seekg(4);
            i_img_train.read(c_ndim,4);
            //swap4(c_ndim);
            memcpy(&i_ndim,c_ndim,4);
            printf("ndim: %d\n",i_ndim);

            //# IMAGES
            char c_images[4];
            int i_images;
            i_img_train.seekg(8);
            i_img_train.read(c_images,4);
            //swap4(c_images);
            memcpy(&i_images,c_images,4);
            printf("images: %d\n",i_images);
            _train_size = i_images; //Set trianing size
            update_valid();

            //IMAGES
            char c_number[4];
            int i_number;
            i_img_train.seekg(12);
            i_img_train.read(c_number,4);
            //swap4(c_dim);
            memcpy(&i_number,c_number,4);
            printf("views per example: %d\n",i_number);
```

```
            //ROWS
            char c_rows[4];
            int i_rows;
            i_img_train.seekg(16);
            i_img_train.read(c_rows,4);
            //swap4(c_rows);
            memcpy(&i_rows,c_rows,4);
            printf("rows: %d\n",i_rows);
            _input_dim_x = i_rows;

            //COLUMNS
            char c_cols[4];
            int i_cols;
            i_img_train.seekg(20);
            i_img_train.read(c_cols,4);
            //swap4(c_cols);
            memcpy(&i_cols,c_cols,4);
            printf("columns: %d\n",i_cols);
            _input_dim_y = i_cols;

            //SET INPUT SIZE
            _input_size = _input_dim_y * _input_dim_x;
            printf("Data items are size:%d\n",_input_size);

            //Allocate memory
            _mini_batch_data = (float*)malloc(_input_size *
_train_size * sizeof(float));
            dev_alloc(&d_mini_batch_data, _input_size *
_mini_batch_size * sizeof(float));
      }
      else
      {
            printf("Couldn't open file. Exiting...");
            return -1;
      }
      //Grab Images
      int input_ptr = 24;
      char* c_img;
      c_img = (char*)malloc(_input_size*sizeof(char));

      //Allocate Training Data Space
      _train_data = (float*)malloc(_train_size * _input_size *
sizeof(float));
      printf("Allocated host memory for training data\n");

      //Image #
      for(int n=0;n<_train_size;n++)
      {
            i_img_train.seekg(input_ptr);
            i_img_train.read(c_img,_input_size);
            input_ptr+=_input_size*2;
```

```
            //Down Column
            //printf("read in");
            for(int i=0;i<_input_size;i++)
            {
                  //convert to 'binary'
                  /*(if (c_img[i] != 0x00)
                        c_img[i] = 0x01;*/
                  //_train_data[ n*_v_size + i] = (float)c_img[i];
                  int tmp;
                  memcpy(&tmp,&c_img[i],1);

                  //memcpy(&_train_data[ n*_v_size + i*28 + j
],&c_img[i*28 + j],1);
                  //printf("[%d]",tmp);


                  /*//memcpy(&data[ n*V_SIZE + i*28 + j ],&c_img[i*28
+ j],1);
                  if(tmp > 20)
                        _train_data[ n*_input_size + i] = 1.0f;
                  else
                        _train_data[ n*_input_size + i] = 0.0f;*/

                  _train_data[n*_input_size + i] = (float)tmp/255;

            }
            //printf("copied");
            //TEST PRINT
            //print_img(&data[n*V_SIZE], 28, 28);
      }

      i_img_train.close();


      //free mem
      free(c_img);

      printf("Data Load Complete!\n");
      return 0;

}

int Trainer::loadTrainingLabels(char* label_file)
{
            ifstream i_lbl_train;
            i_lbl_train.open(label_file, ios::binary);

            if(i_lbl_train.is_open())
            {
                  printf("Loading Labels...\n");
                  //MAGIC NUMBER
                  char c_magic_number[4];
```

```
                int i_magic_number;
                i_lbl_train.read(c_magic_number,4);
                swap4(c_magic_number);
                memcpy(&i_magic_number,c_magic_number,4);
                printf("magic number: %d\n",i_magic_number);

                //# LABELS
                char c_labels[4];
                int i_labels;
                i_lbl_train.seekg(4);
                i_lbl_train.read(c_labels,4);
                swap4(c_labels);
                memcpy(&i_labels,c_labels,4);
                printf("labels: %d\n",i_labels);
                if(_train_size != i_labels)
                {
                      printf("Warning! Label size differs from Data
size\n");
                      return -1;
                }

                //Allocate memory
                _mini_batch_labels = (float*)malloc(_train_size *
_num_classes * sizeof(float));
                _mini_batch_label_vals =
(int*)malloc(_mini_batch_size * sizeof(int));
                dev_alloc(&d_mini_batch_labels, _mini_batch_size *
_num_classes * sizeof(float));

           }
           else
           {
                printf("Couldn't open file. Exiting...");
                return -1;
           }


           //Grab Images
           int input_ptr = 8;
           char* c_lbl;
           c_lbl = (char*)malloc(sizeof(char));

           //Allocate Training Data Space
           _train_label_vals = (int*)malloc(_train_size *
sizeof(int));
           _train_labels = (float*)malloc(_train_size * _num_classes
* sizeof(float));
           printf("Allocated host memory for training labels\n");

           //Image #
           for(int n=0;n<_train_size;n++)
           {
```

```
                i_lbl_train.seekg(input_ptr);
                i_lbl_train.read(c_lbl,1);
                input_ptr++;

                int tmp;
                memcpy(&tmp,c_lbl,1);

                _train_label_vals[n] = tmp;

                for(int c=0;c<_num_classes;c++)
                {
                      if (c == tmp)
                            _train_labels[n*_num_classes + c] = 1.0;
                      else
                            _train_labels[n*_num_classes + c] = 0.0;
                }

                //printf("Label[%d]=%d\n",n, _train_labels[n]);
          }

          i_lbl_train.close();


          //free mem
          free(c_lbl);

          printf("Label Load Complete!\n");
          _using_labels = true;
          return 0;
}


int Trainer::loadTrainingLabelsMAT(char* label_file)
{
          ifstream i_lbl_train;
          i_lbl_train.open(label_file, ios::binary);

          if(i_lbl_train.is_open())
          {
                printf("Loading Labels...\n");
                //MAGIC NUMBER
                char c_magic_number[4];
                int i_magic_number;
                i_lbl_train.read(c_magic_number,4);
                //swap4(c_magic_number);
                memcpy(&i_magic_number,c_magic_number,4);
                printf("magic number: %d\n",i_magic_number);

                //# NUMBER OF DIMENSIONS
                char c_ndim[4];
                int i_ndim;
                i_lbl_train.seekg(4);
```

```
                i_lbl_train.read(c_ndim,4);
                //swap4(c_ndim);
                memcpy(&i_ndim,c_ndim,4);
                printf("ndim: %d\n",i_ndim);

                //# LABELS
                char c_labels[4];
                int i_labels;
                i_lbl_train.seekg(8);
                i_lbl_train.read(c_labels,4);
                //swap4(c_labels);
                memcpy(&i_labels,c_labels,4);
                printf("labels: %d\n",i_labels);
                if(_train_size != i_labels)
                {
                     printf("Warning! Label size differs from Data
size\n");
                     return -1;
                }

                //ROWS
                char c_rows[4];
                int i_rows;
                i_lbl_train.seekg(12);
                i_lbl_train.read(c_rows,4);
                //swap4(c_rows);
                memcpy(&i_rows,c_rows,4);
                printf("ignore: %d\n",i_rows);

                //COLUMNS
                char c_cols[4];
                int i_cols;
                i_lbl_train.seekg(16);
                i_lbl_train.read(c_cols,4);
                //swap4(c_cols);
                memcpy(&i_cols,c_cols,4);
                printf("ignore: %d\n",i_cols);

                //Allocate memory
                _mini_batch_labels = (float*)malloc(_train_size *
_num_classes * sizeof(float));
                _mini_batch_label_vals =
(int*)malloc(_mini_batch_size * sizeof(int));
                dev_alloc(&d_mini_batch_labels, _mini_batch_size *
_num_classes * sizeof(float));

           }
           else
           {
                printf("Couldn't open file. Exiting...");
                return -1;
           }
```

```
            //Grab Images
            int input_ptr = 20;
            char* c_lbl;
            c_lbl = (char*)malloc(sizeof(char));

            //Allocate Training Data Space
            _train_label_vals = (int*)malloc(_train_size *
sizeof(int));
            _train_labels = (float*)malloc(_train_size * _num_classes
* sizeof(float));
            printf("Allocated host memory for training labels\n");

            //Image #
            for(int n=0;n<_train_size;n++)
            {
                  i_lbl_train.seekg(input_ptr);
                  i_lbl_train.read(c_lbl,4);
                  input_ptr+=4;

                  //cout << c_lbl[0] << c_lbl[1] << c_lbl[2] <<
c_lbl[3] << endl;

                  int tmp;
                  memcpy(&tmp,c_lbl,4);

                  _train_label_vals[n] = tmp;

                  for(int c=0;c<_num_classes;c++)
                  {
                        if (c == tmp)
                              _train_labels[n*_num_classes + c] = 1.0;
                        else
                              _train_labels[n*_num_classes + c] = 0.0;
                  }
                  //cout << "int:" << c_lbl << endl;
                  //printf("Label[%d]=%d\n",n, _train_label_vals[n]);


            }

            i_lbl_train.close();


            //free mem
            free(c_lbl);

            printf("Label Load Complete!\n");
            _using_labels = true;
            return 0;
}
```

```
/* ----------------------------------------------------------------
 *         LOAD CONVERT TRAINING DATA
 * Loads in the probability outputs, driven by the training data,
 * from previous layer from file
 *
 * data_file | char* | pointer to name of file
 ----------------------------------------------------------------
*/
int Trainer::loadConvertTrainingData(char* data_file)
{
          ifstream i_img_train;
          i_img_train.open(data_file, ios::binary);

          if(i_img_train.is_open())
          {
               printf("Loading Images...\n");

               //# IMAGES
               char c_images[4];
               int i_images;
               i_img_train.seekg(0);
               i_img_train.read(c_images,4);
               memcpy(&i_images,c_images,4);
               printf("images: %d\n",i_images);
               _train_size = i_images; //Set trianing size
               update_valid();

               //ROWS
               char c_rows[4];
               int i_rows;
               i_img_train.seekg(4);
               i_img_train.read(c_rows,4);
               memcpy(&i_rows,c_rows,4);
               printf("rows: %d\n",i_rows);
               _input_dim_x = i_rows;

               //COLUMNS
               char c_cols[4];
               int i_cols;
               i_img_train.seekg(8);
               i_img_train.read(c_cols,4);
               memcpy(&i_cols,c_cols,4);
               printf("columns: %d\n",i_cols);
               _input_dim_y = i_cols;

               //SET INPUT SIZE
               _input_size = _input_dim_y * _input_dim_x;
               printf("Data items are size:%d\n",_input_size);


               //Allocate memory
```

```
                _mini_batch_data = (float*)malloc(_input_size *
_train_size * sizeof(float));
                dev_alloc(&d_mini_batch_data, _input_size *
_mini_batch_size * sizeof(float));


                //Grab Images
                int input_ptr = 12;
                char* c_img;
                c_img = (char*)malloc(_input_size*sizeof(float));

                //Allocate Training Data Space
                _train_data = (float*)malloc(_train_size *
_input_size * sizeof(float));
                printf("Allocated host memory for training
data\n");

                //Image #
                for(int n=0;n<_train_size;n++)
                {
                     i_img_train.seekg(input_ptr);

     i_img_train.read(c_img,_input_size*sizeof(float));
                     input_ptr+=_input_size*sizeof(float);
                     memcpy(&_train_data[
n*_input_size],c_img,_input_size*sizeof(float));
                }

                i_img_train.close();

                //free mem
                free(c_img);

                printf("Data Load Complete!\n");
                return 0;
           }
           else
           {
                printf("Couldn't open file. Exiting...");
                return -1;
           }

}

/* ----------------------------------------------------------------
 *         SET V
 * Pushes a specific training example into a spot in the mini_batch
 * Copies mini_batch to device
 *
 * n | int | training example to use
 * batch | int | location in mini_batch
```

```
 ---------------------------------------------------------------
*/
void Trainer::setV(int n, int batch)
{
     memcpy(&_mini_batch_data[batch*_input_size],&_train_data[n*_in
put_size],_input_size*sizeof(float));
     cudaMemcpy(d_mini_batch_data,_mini_batch_data,_input_size*_min
i_batch_size*sizeof(float),cudaMemcpyHostToDevice);
     if(_using_labels)
     {
          memcpy(&_mini_batch_labels[n *
_num_classes],&_train_labels[n * _num_classes],_num_classes *
sizeof(float));
          _mini_batch_label_vals[n] = _train_label_vals[n];

     cudaMemcpy(d_mini_batch_labels,_mini_batch_labels,_mini_batch_
size*_num_classes*sizeof(float),cudaMemcpyHostToDevice);
     }
     return;

}

/* ---------------------------------------------------------------
 *         RAND BATCH V
 * Pushes a batch of training data from the host to V0 on device
 ---------------------------------------------------------------
*/
void Trainer::randBatchV()
{
     for(int b=0;b<_mini_batch_size;b++)
     {
          //setV(rand() % (_train_size - _valid_size),b);
          int n = rand() % (_train_size - _valid_size);

     memcpy(&_mini_batch_data[b*_input_size],&_train_data[n*_input_
size],_input_size*sizeof(float));
          if(_using_labels)
          {
               memcpy(&_mini_batch_labels[b *
_num_classes],&_train_labels[n * _num_classes],_num_classes *
sizeof(float));
               _mini_batch_label_vals[b] = _train_label_vals[n];
          }
     }
     cudaMemcpy(d_mini_batch_data,_mini_batch_data,_input_size*_min
i_batch_size*sizeof(float),cudaMemcpyHostToDevice);
     if(_using_labels)

     cudaMemcpy(d_mini_batch_labels,_mini_batch_labels,_mini_batch_
size*_num_classes*sizeof(float),cudaMemcpyHostToDevice);
     return;
}
```

```
//returns batch size
int Trainer::nextBatchTrain()
{
      _cur_batch = _next_batch;
      //check if we have enough examples left for full mini_batch
      int partial_batch_size = _mini_batch_size -((_train_size -
_valid_size)-_cur_batch);
      if(partial_batch_size>0)
      {

      memcpy(&_mini_batch_data[0],&_train_data[_cur_batch*_input_siz
e],partial_batch_size*_input_size*sizeof(float));

      cudaMemcpy(d_mini_batch_data,_mini_batch_data,_input_size*part
ial_batch_size*sizeof(float),cudaMemcpyHostToDevice);
            _next_batch = 0; //we've reached the end if we need to
load partial
            return partial_batch_size;
      }
      else
      {

      memcpy(&_mini_batch_data[0],&_train_data[_cur_batch*_input_siz
e],_mini_batch_size*_input_size*sizeof(float));

      cudaMemcpy(d_mini_batch_data,_mini_batch_data,_input_size*_min
i_batch_size*sizeof(float),cudaMemcpyHostToDevice);
            //update _cur_batch
            _next_batch+=_mini_batch_size;
            if (_next_batch>=(_train_size - _valid_size))
                  _next_batch=0;

            return _mini_batch_size;
      }
}

//returns batch size
int Trainer::nextBatchValid()
{
      if(_next_batch < (_train_size - _valid_size))
            _next_batch = (_train_size - _valid_size); //set to start
of validation data

      _cur_batch = _next_batch;

      //check if we have enough examples left for full mini_batch
      int partial_batch_size = _mini_batch_size -(_train_size-
_cur_batch);
      if(partial_batch_size>0)
      {
```

```
      memcpy(&_mini_batch_data[0],&_train_data[_cur_batch*_input_siz
e],partial_batch_size*_input_size*sizeof(float));

      cudaMemcpy(d_mini_batch_data,_mini_batch_data,_input_size*part
ial_batch_size*sizeof(float),cudaMemcpyHostToDevice);
            _next_batch = 0; //we've reached the end if we need to
load partial
            return partial_batch_size;
      }
      else
      {

      memcpy(&_mini_batch_data[0],&_train_data[_cur_batch*_input_siz
e],_mini_batch_size*_input_size*sizeof(float));

      cudaMemcpy(d_mini_batch_data,_mini_batch_data,_input_size*_min
i_batch_size*sizeof(float),cudaMemcpyHostToDevice);
            //update _cur_batch
            _next_batch+=_mini_batch_size;
            if (_next_batch >= _train_size )
                  _next_batch=0;

            return _mini_batch_size;
      }
}

int Trainer::batchClassification(float* d_top_prob, int batch_size)
{
      //int* answers = (int*)malloc(batch_size * sizeof(int));
      int incorrect = 0;
      float* probs = (float*)malloc(_num_classes * batch_size *
sizeof(float));
      cudaMemcpy(probs, d_top_prob,
batch_size*_num_classes*sizeof(float), cudaMemcpyDeviceToHost);

      float max;
      int max_loc;
      for(int n=0;n<batch_size;n++)
      {
            max = 0;
            max_loc = 0;
            for(int k=0;k<_num_classes;k++)
            {
                  if(probs[n*_num_classes + k] > max)
                  {
                        max = probs[n*_num_classes + k];
                        max_loc = k;
                  }
            }
            //answers[n] = max_loc;
```

```
            printf("Train[%d]: Guess[%d] ||
Answer[%d]\n",_cur_batch+n,max_loc,
_train_label_vals[_cur_batch+n]);
            if (max_loc != _train_label_vals[_cur_batch + n])
                  incorrect++;
      }

      free(probs);
      return incorrect;
      //return answers;
}

float Trainer::batchError(float* d_top_prob, int batch_size)
{
      //int* answers = (int*)malloc(batch_size * sizeof(int));
      int incorrect = 0;
      float* probs = (float*)malloc(_num_classes * batch_size *
sizeof(float));
      cudaMemcpy(probs, d_top_prob,
batch_size*_num_classes*sizeof(float), cudaMemcpyDeviceToHost);

      float sum=0;

      for(int n=0;n<batch_size;n++)
      {
            for(int k=0;k<_num_classes;k++)
            {
                  sum += (pow(probs[n*_num_classes + k] -
_train_labels[(_cur_batch*_num_classes) + (n*_num_classes) + k],2) /
2);
            }
      }

      free(probs);
      return sum;
      //return answers;
}

/* ------------------------------------------------------------
 *         SHOW TRAINING
 * Draws the selected training item
 * n | int | training data number to display
 ------------------------------------------------------------
*/
void Trainer::showTraining(int n)
{
      if(n>=_train_size || n<0)
      {
            printf("Selected training data does not exist\n");
            return;
      }
      else
```

```
      {
            return show(&_train_data[n*_input_size], _input_dim_x ,
_input_dim_y);
      }
}

void Trainer::showCurrent(int b)
{
      if(b > _mini_batch_size)
      {
            printf("Selected training batch item does not exist\n");
            return;
      }
      else
      {
            return show(&_mini_batch_data[b*_input_size],
_input_dim_x , _input_dim_y);
      }
}

/* ----------------------------------------------------------------
 *         INC N
 * Increment counter for the number of training examples seen
 * this epoch
 ----------------------------------------------------------------
*/
void Trainer::incN()
{
      _n += _mini_batch_size;
}

/* ----------------------------------------------------------------
 *         EPOCH COMPLETE
 * Check if all the training examples have been seen for this epoch
 *
 * return  true if all examples seen
 *         false otherwise
 ----------------------------------------------------------------
*/
bool Trainer::epochComplete()
{
      if( _n >= _train_size - _valid_size)
      {
            _n = 0;
            _cur_epoch++;
            _learn_rate = _learn_rate / (1 + (float)_cur_epoch *
_lr_beta);
            if(_momentum < 0.9)
            {
                  _momentum += 0.1;
                  printf("updating momentum to: %f\n", _momentum);
            }
```

```
            printf("Current learning rate: %f\n", _learn_rate);
            return true;
      }
      else
            return false;
}
```

trainer_calc.cpp

```
#include "../inc/trainer.h"

/* ------------------------------------------------------------------
 *         PIXEL PROB
 * Calculates p(v) over the training data
 *
 * index | int | pixel index to calculates
 ------------------------------------------------------------------
*/
float Trainer::pixelProb(int index)
{
      //Visible Bias = log[Pi/(1 - Pi)]
      // Pi calculated from training examples
      if(index<_input_size)
      {
            int sum = 0;
            //Calc Pi
            for (int n=0;n<_train_size;n++)
            {
                  sum += _train_data[n*_input_size + index];
            }
            //avoid log(0) = -inf
            if(sum == 0)
                  sum = 1;
            float Pi = (float)sum / _train_size;
            return log(Pi / (1-Pi));
            //printf("Pi=%f | ai[%d])=%f\n",Pi,i,log(Pi / (1-Pi)));
      }
      else
      {
            printf("Pixel Probability requested for index out of
bounds.\n");
            return 0;
      }
}
```

rbm.h

```
#ifndef RBM_H
#define RBM_H
```

```
#include <fstream>
#include <iostream>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <map>
#include <cuda.h>
#include <cuda_runtime.h>

#include "../../../0.Utils/utilLearn.h"

#include "GL/freeglut.h"
#include "GL/gl.h"

#include <histo.h>

#include "../../connection/inc/connection.h"
#include "../../layer/inc/layer.h"


using namespace std;
using namespace utilLearn;

class Rbm
{

public:
      //rbm.cpp
      Rbm(Layer *vis, Layer *hid, Connection *connection);
      ~Rbm();

      //Host Gets
      int getHSize(){return _hid->getSize();};
      int getVSize(){return _vis->getSize();};

      //Device Gets (Connections)
      float* getDevW(){return _connection->d_weight;};
      float* getDevWT(){return _connection->d_weight_t;};
      float* getDevA(){return _connection->d_a;};
      float* getDevB(){return _connection->d_b;};
      float* getDevDw(){return _connection->d_dw;};
      float* getDevVw(){return _connection->d_vel_weight;};

      //Device Gets (Layers)
      float* getVX(){return _vis->d_state;};
      float* getH0(){return _hid->d_initial_state;};
      float* getHX(){return _hid->d_state;};
      float* getHrand(){return _hid->d_rand;};
      float* getHQ(){return _hid->d_q;};

      //rbm_init.cpp
```

```
      void initParams();
      void save(char* out_file);
      int load(char* in_file);
      int saveHDim(ofstream* file, int loc){return _hid-
>saveDim(file, loc);};
      int saveH(ofstream* file, int loc){return _hid-
>saveState(file, loc);};

      void showHidden0(int b, int g){_hid->showState(false, b,g);};
      void showVisibleX(int b, int g){_vis->showState(true, b,g);};
      void showHiddenX(int b, int g){_hid->showState(true, b,g);};
      void histogramQ(){_hid->histogramQ();};
      void histogramW(){_connection->histogramW();};
      void histogramA(){_connection->histogramA();};
      void histogramB(){_connection->histogramB();};
      void histogramDw(){_connection->histogramDw();};

      //rbm_disp.cpp
      void projectH(int index);

      //rbm_calc.cpp
      //float calcFreeNRGTrain();
      //float calcFreeNRGValid();
      //void saveFreeNRG();
      void checkSparsityH(){_hid->checkSparsity();};


protected:
      //Objects
      Connection*      _connection;
      Layer*     _vis;
      Layer*     _hid;


      int        _total_gpu_mem;

};

#endif
```

rbm.cpp

```
#include "../inc/rbm.h"

/* ---------------------------------------------------------------
 *         CONSTRUCTOR
 * v_size_x           | int | visible layer X dimension (for
display)
 * v_size_y           | int | visible layer Y dimension (for
display)
 * h_size_x           | int | hidden layer X dimension (for display)
```

```
 * h_size_y           | int | hidden layer Y dimension (for display)
 * mini_batch_size    | int | number of examples per batch
 * gibbs_samples| int | number of fantasy particles
 ---------------------------------------------------------------------
*/
Rbm::Rbm(Layer *vis, Layer *hid, Connection *connection)
{
      _vis = vis;
      _hid = hid;
      _connection = connection;

      //Device memory allocation
      _total_gpu_mem = 0;

      return;
}

/* ---------------------------------------------------------------------
 *         DESTRUCTOR
 ---------------------------------------------------------------------
*/
Rbm::~Rbm()
{
      //Classes
      _connection->~Connection();
      _vis->~Layer();
      _hid->~Layer();

      return;
}
```

rbm_calc.cpp

```
#include "../inc/rbm.h"
/*
float Rbm::calcFreeNRGTrain()
{
      cudaMemcpy(_a, d_a,
_v.size*sizeof(float),cudaMemcpyDeviceToHost);
      cudaMemcpy(_b, d_b, _h.size*sizeof(float),
cudaMemcpyDeviceToHost);
      cudaMemcpy(_weight,d_weight,_w_size*sizeof(float),
cudaMemcpyDeviceToHost);

      float sum = 0;

      //iterate valid length up through training data
      for(int n=0;n<_valid_size;n++)
      {
```

```
            float left_sum = 0;
            for (int i=0;i<_v.size;i++)
            {
                  left_sum+= _a[i]*_train_data[n*_v.size + i];
            }

            float right_sum = 0;
            for(int j=0;j<_h.size;j++)
            {
                  float xj= _b[j];
                  for (int i=0;i<_v.size;i++)
                  {
                        xj+= _train_data[n*_v.size +
i]*_weight[(i*_h.size)+j];
                  }
                  right_sum+=log(1 + exp(xj));
            }

            sum += -left_sum - right_sum;
      }

      printf("FNRG (Training) = %f\n", sum/(float)_valid_size);
      return sum/(float)_valid_size;



}

float Rbm::calcFreeNRGValid()
{

      cudaMemcpy(_a, d_a,
_v.size*sizeof(float),cudaMemcpyDeviceToHost);
      cudaMemcpy(_b, d_b, _h.size*sizeof(float),
cudaMemcpyDeviceToHost);
      cudaMemcpy(_weight,d_weight,_w_size*sizeof(float),
cudaMemcpyDeviceToHost);

      float sum = 0;

      //iterate down from tail end (the validation data)
      for(int n=_train_size - 1;n>=_train_size - _valid_size;n--)
      {

            float left_sum = 0;
            for (int i=0;i<_v.size;i++)
            {
                  left_sum+= _a[i]*_train_data[n*_v.size + i];
            }

            float right_sum = 0;
            for(int j=0;j<_h.size;j++)
```

```
            {
                  float xj= _b[j];
                  for (int i=0;i<_v.size;i++)
                  {
                        xj+= _train_data[n*_v.size +
i]*_weight[(i*_h.size)+j];
                  }
                  right_sum+=log(1 + exp(xj));
            }

            sum += -left_sum - right_sum;
      }

      printf("FNRG (Validation) = %f\n", sum/(float)_valid_size);
      return sum/(float)_valid_size;

}

void Rbm::saveFreeNRG()
{
      printf("Calculating Free Energy...\n");
      ofstream o_file;
      o_file.open("experiments/last_run.fnrg", ios::app);
      o_file.seekp( ios::end );
      o_file << "Epoch: " << _cur_epoch << "\t";
      o_file << "FNRG(train): " << calcFreeNRGTrain() << "\t";
      o_file << "FNRG(valid): " << calcFreeNRGValid() << "\n";
      o_file.close();
}

*/
```

rbm_disp.cpp

```
#include "../inc/rbm.h"

void Rbm::projectH(int index)
{
      _vis->projectWeights( _connection->getWTRow(index));
}
```

rbm_init.cpp

```
#include "../inc/rbm.h"


/* -----------------------------------------------------------------
 *          INIT PARAMS
```

```
 * Initializes the paremeters for a new RBM.
 ---------------------------------------------------------------------
*/
void Rbm::initParams()
{

      _connection->initParams();
      _vis->initParams();
      _hid->initParams();

      //saveFreeNRG(); //Get initial FreeNRG

      return;
}

/* ---------------------------------------------------------------------
 *         SAVE
 * Saves the RBM Parameters to a file
 *
 * out_file | char* | pointer to name of file
 ---------------------------------------------------------------------
*/
void Rbm::save(char* out_file)
{
      printf("Saving RBM...");
      ofstream o_file;
      o_file.open(out_file, ios::binary);

      int loc = 0;
      if(o_file.is_open())
      {
            loc = _connection->save(&o_file, loc);

            loc = _hid->saveQ(&o_file, loc);

            o_file.close();
            printf("Completed\n");
      }
      else
            printf("Failed\n");
      return;
}

/* ---------------------------------------------------------------------
 *         LOAD
 * Loads the RBM Parameters from a file
 *
 * in_file | char* | pointer to name of file
 ---------------------------------------------------------------------
*/
int Rbm::load(char* in_file)
{
```

```
	printf("Loading RBM from %s...",in_file);
	ifstream i_file;
	i_file.open(in_file, ios::binary);
	if(i_file.is_open())
	{
		int loc = 0;

		loc = _connection->load(&i_file, loc);

		loc = _hid->loadQ(&i_file, loc);

		i_file.close();

		printf("Completed!\n");
		return 0;
	}
	else
	{
		printf("Failed to open file\n");
		return -1;
	}
}
```

dbn.h

```
#ifndef DBN_H
#define DBN_H

#include <cuda.h>
#include <cuda_runtime.h>



#include "../../connection/inc/connection.h"
#include "../../layer/inc/layer.h"


using namespace std;

class Dbn
{

public:
	Dbn(Layer *vis, Layer *h0, Layer *h1, Connection * v_h0,
Connection * h0_h1);
	~Dbn();

	//Host Gets
	int getH1Size(){return _h1->getSize();};
	int getH0Size(){return _h0->getSize();};
	int getVSize(){return _vis->getSize();};
```

```
      //Device Gets (Connections)
      float* getDevW_0(){return _v_h0->d_weight;};
      float* getDevWT_0(){return _v_h0->d_weight_t;};
      float* getDevVw_0(){return _v_h0->d_vel_weight;};
      float* getDevB_0(){return _v_h0->d_b;};
      float* getDevA_0(){return _v_h0->d_a;};
      float* getDevW_1(){return _h0_h1->d_weight;};
      float* getDevWT_1(){return _h0_h1->d_weight_t;};
      float* getDevVw_1(){return _h0_h1->d_vel_weight;};
      float* getDevB_1(){return _h0_h1->d_b;};
      float* getDevA_1(){return _h0_h1->d_a;};


      //Device Gets (Layers)
      float* getVX(){return _vis->d_state;};
      float* getVPred(){return _vis->d_q;};
      float* getH0(){return _h0->d_state;};
      float* getH0Rand(){return _h0->d_rand;};
      float* getH0In(){return _h0->d_initial_state;};
      float* getH0Pred(){return _h0->d_q;};
      float* getH1(){return _h1->d_state;};
      float* getH1Rand(){return _h1->d_rand;};
      float* getH1In(){return _h1->d_initial_state;};
      float* getH1Pred(){return _h0->d_q;};


      void showVX(int b){_vis->showState(true,b,0);};
      void showH0(int b){_h0->showState(true, b, 0);};
      void showH1(int b){_h1->showState(true, b, 0);};

      void saveLayers(char* out_file_lvl1, char* out_file_lvl2);
      int loadLayers(char* in_file_lvl1, char* in_file_lvl2);

protected:
      Layer*     _vis;
      Layer*     _h0;
      Layer*     _h1;

      Connection* _v_h0;
      Connection* _h0_h1;
};


#endif
```

dbn.cpp

```
#include "../inc/dbn.h"
```

```
Dbn::Dbn(Layer *vis, Layer *h0, Layer *h1, Connection * v_h0,
Connection * h0_h1)
{
      _vis = vis;
      _h0 = h0;
      _h1 = h1;

      _v_h0 = v_h0;
      _h0_h1 = h0_h1;
}

Dbn::~Dbn()
{

}

void Dbn::saveLayers(char* out_file_lvl1, char* out_file_lvl2)
{
      printf("Saving Layers...");
            ofstream o_file;
            o_file.open(out_file_lvl1, ios::binary);

            int loc = 0;
            if(o_file.is_open())
            {
                  loc = _v_h0->save(&o_file, 0);
                  o_file.close();
                  printf("Completed Layer 1\n");
            }
            else
                  printf("Failed Layer 1\n");

            o_file.open(out_file_lvl2, ios::binary);

            loc = 0;
            if(o_file.is_open())
            {
                  loc = _h0_h1->save(&o_file, loc);
                  o_file.close();
                  printf("Completed Layer 2\n");
            }
            else
                  printf("Failed Layer 2\n");



            return;
}

int Dbn::loadLayers(char* in_file_lvl1, char* in_file_lvl2)
{
      printf("Loading Layer from %s...",in_file_lvl1);
```

```
	ifstream i_file;
	i_file.open(in_file_lvl1, ios::binary);
	if(i_file.is_open())
	{
		int loc = 0;

		loc = _v_h0->load(&i_file, loc);

		i_file.close();

		printf("Completed!\n");
	}
	else
	{
		printf("Failed to open file\n");
		return -1;
	}

	printf("Loading Layer from %s...",in_file_lvl2);
	i_file.open(in_file_lvl2, ios::binary);
	if(i_file.is_open())
	{
		int loc = 0;

		loc = _h0_h1->load(&i_file, loc);

		i_file.close();

		printf("Completed!\n");
		return 0;
	}
	else
	{
		printf("Failed to open file\n");
		return -1;
	}
}
```

nn.h

```
#ifndef NN_H
#define NN_H

#include<list>


#include "../../connection/inc/connection.h"
#include "../../layer/inc/layer.h"


using namespace std;
```

```
class Nn
{

public:
      Nn(Layer *vis, Layer *h0, Layer *h1, Layer *top, Connection *
v_h0, Connection * h0_h1, Connection * h1_top);
      ~Nn();

      //Host Gets
      int getTopSize(){return _top->getSize();};
      int getH1Size(){return _h1->getSize();};
      int getH0Size(){return _h0->getSize();};
      int getVSize(){return _vis->getSize();};

      //Device Gets (Connections)
      float* getDevW_0(){return _v_h0->d_weight;};
      float* getDevVw_0(){return _v_h0->d_vel_weight;};
      float* getDevB_0(){return _v_h0->d_b;};
      float* getDevW_1(){return _h0_h1->d_weight;};
      float* getDevVw_1(){return _h0_h1->d_vel_weight;};
      float* getDevB_1(){return _h0_h1->d_b;};
      float* getDevW_2(){return _h1_top->d_weight;};
      float* getDevVw_2(){return _h1_top->d_vel_weight;};
      float* getDevB_2(){return _h1_top->d_b;};

      //Device Gets (Layers)
      float* getVX(){return _vis->d_state;};
      float* getH0(){return _h0->d_state;};
      float* getH0In(){return _h0->d_initial_state;};
      float* getH1(){return _h1->d_state;};
      float* getH1In(){return _h1->d_initial_state;};
      float* getTop(){return _top->d_state;};


      float* getTopError(){return _top->d_error;};
      float* getH1Error(){return _h1->d_error;};
      float* getH0Error(){return _h0->d_error;};

      int getAnswer();

      void showH0(int b){_h0->showState(true, b, 0);};
      void showH1(int b){_h1->showState(true, b, 0);};
      void showTop(int b){_top->showState(true, b, 0);};
      void printTop(int b){_top->printState(true,b, 0);};

      void setTopProb();

      void saveComplete(char* out_file);
      int loadComplete(char* in_file);
```

```
protected:
      //list<Connection*>   _connections;
      //list<Layer*>        _layers;

      Layer*     _vis;
      Layer*     _h0;
      Layer*     _h1;
      Layer*     _top;

      Connection* _v_h0;
      Connection* _h0_h1;
      Connection* _h1_top;
};

#endif
```

nn.cpp

```
#include "../inc/nn.h"

Nn::Nn(Layer *vis, Layer *h0, Layer *h1, Layer *top, Connection *
v_h0, Connection * h0_h1, Connection * h1_top)
{
      /*_layers.push_back(vis);
      _layers.push_back(h0);
      _layers.push_back(h1);
      _layers.push_back(top);

      _connections.push_back(v_h0);
      _connections.push_back(h0_h1);
      _connections.push_back(h1_top);*/

      //to do: add verification

      _vis = vis;
      _h0 = h0;
      _h1 = h1;
      _top = top;

      _v_h0 = v_h0;
      _h0_h1 = h0_h1;
      _h1_top = h1_top;
}

Nn::~Nn()
{

}

void Nn::setTopProb()
{
```

```
	float prob = 1 / (float)_top->getSize();
	//printf("prob=%f\n",prob);
	prob = log(prob / (1 - prob));
	//printf("log prob=%f\n",prob);
	for(int i=0;i<_top->getSize();i++)
	{
		_h1_top->setB(i,prob);
	}
	_h1_top->cpyB();
}

void Nn::saveComplete(char* out_file)
{
	printf("Saving Net...");
		ofstream o_file;
		o_file.open(out_file, ios::binary);

		int loc = 0;
		if(o_file.is_open())
		{
			loc = _v_h0->save(&o_file, loc);
			loc = _h0_h1->save(&o_file, loc);
			loc = _h1_top->save(&o_file, loc);

			o_file.close();
			printf("Completed\n");
		}
		else
			printf("Failed\n");
		return;
}

int Nn::loadComplete(char* in_file)
{
	printf("Loading Net from %s...",in_file);
	ifstream i_file;
	i_file.open(in_file, ios::binary);
	if(i_file.is_open())
	{
		int loc = 0;

		loc = _v_h0->load(&i_file, loc);
		loc = _h0_h1->load(&i_file, loc);
		loc = _h1_top->load(&i_file, loc);

		i_file.close();

		printf("Completed!\n");
		return 0;
	}
	else
	{
```

```
            printf("Failed to open file\n");
            return -1;
      }
}
```

rbm.cu

```
#include "connection/inc/connection.h"
#include "rbm/inc/rbm.h"
#include "trainer/inc/trainer.h"
#include <curand.h>

using namespace utilLearn;

//RBM LEARNING PARAMETERS
#define BATCH              100
#define SAMPLES            1
#define STEPS              1
#define EPOCH              200

//RBM SIZE PARAMETERS
//#define  VSIZE_X              96
//#define VSIZE_Y               96
//#define VSIZE             9216
#define    VSIZE_X              64
#define VSIZE_Y            64
#define VSIZE              4096

#define    HSIZE_X              64
#define HSIZE_Y            64
#define HSIZE              4096

//RBM MODE
#define MODE               0          //0=learn 1=think 2=project
3=convert
#define VISUAL             false
#define PERSISTENT         1
#define BOTTOM             false
#define LOAD               true
#define TRAIN_EXAMPLE 4444

char * const param_file = "params/norb-persistent-lvl2.rbm";
char * const converted_data_file = "data/norb-images-lvl2.floats";
//char * const data_file = "data/train-images.idx3-ubyte";
char * const data_file = "data/smallnorb-5x46789x9x18x6x2x96x96-
training-dat.mat";

//For Transpose
#define TILE_DIM     16
#define BLOCK_ROWS         16
```

```
#define    BLOCKS_LAYER    16

//Display Functions
void train();
void train_novis();
void project();
void think();

//Helper Functions
void init_fantasy();
void train_mini();
void train_mini_persistent();
void update_params();
void convert(char* out_file);
void saveFreeNRG();
float calcFreeNRGTrain();
float calcFreeNRGValid();

//CUDA Functions
__global__ void transpose(float *w, float *wt);
__global__ void upPassInitProb(float* v0, float* h0, float* b,
float* w);
__global__ void upPassInit(float* v0, float* h0, float* b, float* w,
float* rnd);
__global__ void upPass(float* vX, float* hX, float* b, float* w,
float* rnd);
__global__ void upPassProb(float* vX, float* hX, float* b, float*
w);
__global__ void downPass(float* vX, float* hX, float* a, float* wt);
__global__ void updateW(float* v0, float* vX, float* h0, float* hX,
float* w, float* vel_w, float momentum, float* q, float* dw, float
l_rate);
__global__ void updateA(float* v0, float* vX, float* a, float
l_rate);
__global__ void updateB(float* h0, float* hX, float* b, float* q,
float l_rate);
__global__ void calcLeftSum(float* sum, float* v0, float* a);
__global__ void calcRightSum(float* sum, float* v0, float* b, float*
w);

//Globals
Rbm *my_rbm;
Trainer *my_trainer;
curandGenerator_t d_rand;
int visual_iterator;
int projection_iterator;

//float total_time;

/* ----------------------------------------------------------------
 *         MAIN
 * Sets up RBM for a selected task and initializes a loop
```

```
 *
 * argc | int    | number of arguments passed
 * argv | char**| arguments passed in
 ----------------------------------------------------------------
*/
int main(int argc, char** argv)
{
      visual_iterator = 1;
      projection_iterator = 0;
      //total_time = 0;
      //Set GPU 1 (currently not used for display)
      cudaSetDevice(1);

      //Set up basic units
      Connection* my_connection = new Connection(VSIZE, HSIZE);
      Layer* my_visible = new Layer(VSIZE_X, VSIZE_Y, BATCH,
SAMPLES, false);
      Layer* my_hidden = new Layer(HSIZE_X, HSIZE_Y, BATCH, SAMPLES,
true);

      my_rbm = new Rbm(my_visible, my_hidden, my_connection);
      my_trainer = new Trainer(BATCH, SAMPLES, EPOCH, 0.001, 10,
0.5, 0.0001);

      //Load Data Set
      if(BOTTOM)
      {
            if(my_trainer->loadTrainingDataMAT(data_file) < 0)
            {
                  printf("An error occurred loading the training
data. Exiting...\n");
                  return -1;
            }
      }
      else
      {
            if(my_trainer-
>loadConvertTrainingData(converted_data_file) < 0)
            {
                  printf("An error occurred loading the converted
training data. Exiting...\n");
                  return -1;
            }
      }

      //Set up RBM Parameters
      if(LOAD)
      {
            if(my_rbm->load(param_file) < 0)
            {
                  printf("An error occurred loading the parameters.
Exiting...\n");
```

```
                return -1;
            }
        }
        else
        {
            printf("Initializing Paramters\n");
            my_rbm->initParams();
            dim3 grid(my_rbm->getHSize()/TILE_DIM, my_rbm-
>getVSize()/TILE_DIM), threads(TILE_DIM,BLOCK_ROWS);
            transpose<<<grid,threads>>>(my_rbm->getDevW(), my_rbm-
>getDevWT());
            printf("Setting visual bias to data probability\n");
            //Set visual bias to training data probability
            for(int i=0;i<my_connection->getVSize();i++)
            {
                //printf("visual bias[%d]=%f\n",i, my_trainer-
>pixelProb(i));
                my_connection->setA(i,my_trainer->pixelProb(i));
            }
            my_connection->cpyA(); //Place on device
            //saveFreeNRG();
            //saveFreeNRG(); //Check Free Energy of New System
        }

        //Set up Random Initializer
        curandCreateGenerator(&d_rand, CURAND_RNG_PSEUDO_MTGP32);
        srand((unsigned)time(0));
        int seed = (rand() % 1000);
        curandSetPseudoRandomGeneratorSeed(d_rand, seed);

        if(!VISUAL)
        {
            if(MODE==0)
            {
                if(PERSISTENT)
                    init_fantasy();
                train_novis();
            }
            else if(MODE==3)
            {
                convert(converted_data_file);
                return 0;
            }
            else
                printf("Incorrect Mode Selected\n");

            return 0;
        }
        else
        {
            //Set up GLUT Parameters
            glutInit(&argc, argv);
```

```
            glutInitDisplayMode(GLUT_SINGLE);
            glutInitWindowSize(500,500);
            glutInitWindowPosition(250,250);
            glutCreateWindow("Restricted Boltzmann Machine");

            if(MODE==0)
            {
                  if(PERSISTENT)
                        init_fantasy();
                  glutDisplayFunc(train);
            }
            else if(MODE==1)
            {
                  glutDisplayFunc(think);
                  my_hidden->randState(0.1);
            }
            else if(MODE==2)
                  glutDisplayFunc(project);
            else if(MODE==3)
            {
                  convert(converted_data_file);
                  return 0;
            }

            glutMainLoop();

            return 0;
      }
}

/*=====================================================================
=
 *              DISPLAY FUNCTIONS

=====================================================================*
/

void train_novis()
{

      while(!my_trainer->trainComplete())
      {
            //cudaEvent_t start, stop;
            //float time;
            //cudaEventCreate(&start);
            //cudaEventCreate(&stop);

            //cudaEventRecord(start, 0);
            if(PERSISTENT)
                  train_mini_persistent();
            else
                  train_mini();
```

```
            //cudaEventRecord(stop, 0);
            //cudaEventSynchronize(stop);
            //cudaEventElapsedTime(&time, start, stop);

            //printf("Time for mini_batch (100): %f ms\n", time);
            //total_time+=time;

            dim3 blockDim(BLOCKS_LAYER,SAMPLES,BATCH);
            dim3 threadDimUp(my_rbm->getHSize()/BLOCKS_LAYER);
            dim3 threadDimDown(my_rbm->getVSize()/BLOCKS_LAYER);

            //visual_iterator++;
            if(visual_iterator>=7)
                  visual_iterator = 0;

            //Update training status
            my_trainer->incN();
            if(my_trainer->epochComplete())
            {
                  printf("Epoch %d Complete!\n", my_trainer-
>getEpoch());
                  my_rbm->save(param_file);
                  //saveFreeNRG();
                  my_rbm->checkSparsityH();
            }

      }

      printf("Training run complete!\n");
      //printf("Average Time for mini_batch(100): %f ms\n",
total_time / ( (float)(my_trainer->getTrainSize() - my_trainer-
>getValidSize()) / (float)BATCH) );

}

//Trains the RBM with a selected visual feeback
void train()
{
      glClearColor(1.0,1.0,1.0,1.0);
      glClear(GL_COLOR_BUFFER_BIT);
      glOrtho(-1.0,1.0,-1.0,1.0,-1.0,1.0);


      //cudaEvent_t start, stop;
      //float time;
      //cudaEventCreate(&start);
      //cudaEventCreate(&stop);

      //cudaEventRecord(start, 0);
      if(PERSISTENT)
            train_mini_persistent();
      else
```

```
            train_mini();
      //cudaEventRecord(stop, 0);
      //cudaEventSynchronize(stop);
      //cudaEventElapsedTime(&time, start, stop);

      //printf("Time for mini_batch (100): %f ms\n", time);
      //total_time+=time;

      dim3 blockDim(BLOCKS_LAYER,SAMPLES,BATCH);
      dim3 threadDimUp(my_rbm->getHSize()/BLOCKS_LAYER);
      dim3 threadDimDown(my_rbm->getVSize()/BLOCKS_LAYER);

      //Visual Display
      switch (visual_iterator)
      {
      case 0:
            my_trainer->showTraining(TRAIN_EXAMPLE);
            break;
      case 1:
            my_trainer->setV(TRAIN_EXAMPLE,0);
            curandGenerateUniform(d_rand, (float *) my_rbm-
>getHrand(), my_rbm->getHSize()*my_trainer->getNumFantasy());
            upPassInit<<<blockDim,threadDimUp>>>(my_trainer-
>d_mini_batch_data,my_rbm->getH0(),my_rbm->getDevB(),my_rbm-
>getDevW(), my_rbm->getHrand());
            downPass<<<blockDim,threadDimDown>>>(my_rbm-
>getVX(),my_rbm->getH0(),my_rbm->getDevA(),my_rbm->getDevWT());
            my_rbm->showVisibleX(0,0);
            break;
      case 2:
            my_trainer->setV(TRAIN_EXAMPLE,0);
            curandGenerateUniform(d_rand, (float *) my_rbm-
>getHrand(), my_rbm->getHSize()*my_trainer->getNumFantasy());
            upPassInit<<<blockDim,threadDimUp>>>(my_trainer-
>d_mini_batch_data,my_rbm->getH0(),my_rbm->getDevB(),my_rbm-
>getDevW(), my_rbm->getHrand());
            my_rbm->showHidden0(0,0);
            break;
      case 3:
            my_rbm->histogramW();
            break;
      case 4:
            my_rbm->histogramA();
            break;
      case 5:
            my_rbm->histogramB();
            break;
      case 6:
            my_rbm->histogramQ();
            break;
      default:
            printf("Display request for invalid argument\n");
```

```
            break;
      }
      glFlush();

      //visual_iterator++;
      if(visual_iterator>=7)
            visual_iterator = 0;

      //Update training status
      my_trainer->incN();
      if(my_trainer->epochComplete())
      {
            printf("Epoch %d Complete!\n", my_trainer->getEpoch());
            my_rbm->save(param_file);
            //saveFreeNRG();
            my_rbm->checkSparsityH();
      }

      if(!my_trainer->trainComplete())
            glutPostRedisplay();
      else
      {
            printf("Training run complete!\n");
            //printf("Average Time for mini_batch(100): %f ms\n",
total_time / ( (float)(my_trainer->getTrainSize() - my_trainer-
>getValidSize()) / (float)BATCH) );
      }

}

void project()
{
      glClearColor(1.0,1.0,1.0,1.0);
      glClear(GL_COLOR_BUFFER_BIT);
      glOrtho(-1.0,1.0,-1.0,1.0,-1.0,1.0);

      my_rbm->projectH(projection_iterator);
      projection_iterator++;

      glFlush();
      sleep(2);
      glutPostRedisplay();


}

//Iterates over unlimited gibbs steps and displays the visible units
after each pass
void think()
{
      glClearColor(1.0,1.0,1.0,1.0);
      glClear(GL_COLOR_BUFFER_BIT);
```

```
	glOrtho(-1.0,1.0,-1.0,1.0,-1.0,1.0);

	dim3 blockDim(BLOCKS_LAYER,1,1);
	dim3 threadDimUp(my_rbm->getHSize()/BLOCKS_LAYER);
	dim3 threadDimDown(my_rbm->getVSize()/BLOCKS_LAYER);

	curandGenerateUniform(d_rand, (float *) my_rbm->getHrand(),
my_rbm->getHSize()*my_trainer->getNumFantasy());
	downPass<<<blockDim,threadDimDown>>>(my_rbm->getVX(),my_rbm-
>getHX(),my_rbm->getDevA(),my_rbm->getDevWT());
	upPass<<<blockDim,threadDimUp>>>(my_rbm->getVX(),my_rbm-
>getHX(),my_rbm->getDevB(),my_rbm->getDevW(), my_rbm->getHrand());

	my_rbm->showVisibleX(0,0);
	//my_rbm->showHiddenX(0,0);

	glFlush();
	glutPostRedisplay();

}

/*=================================================================
=
 *               HELPER FUNCTIONS

==================================================================*
/

void init_fantasy()
{
	my_trainer->randBatchV();
	curandGenerateUniform(d_rand, (float *) my_rbm->getHrand(),
my_rbm->getHSize()*my_trainer->getNumFantasy());

	//Calculate HX
	dim3 blockDim(BLOCKS_LAYER,SAMPLES,BATCH);
	dim3 threadDimUp(my_rbm->getHSize()/BLOCKS_LAYER);
	upPassInit<<<blockDim,threadDimUp>>>(my_trainer-
>d_mini_batch_data,my_rbm->getHX(),my_rbm->getDevB(),my_rbm-
>getDevW(), my_rbm->getHrand());
}

//Trains a single mini-batch
void train_mini()
{
	//Select Batch Samples V0
	my_trainer->randBatchV();
	curandGenerateUniform(d_rand, (float *) my_rbm->getHrand(),
my_rbm->getHSize()*my_trainer->getNumFantasy());

	//Calculate H0
	dim3 blockDim(BLOCKS_LAYER,SAMPLES,BATCH);
```

```
	dim3 threadDimUp(my_rbm->getHSize()/BLOCKS_LAYER);
	dim3 threadDimDown(my_rbm->getVSize()/BLOCKS_LAYER);
	upPassInit<<<blockDim,threadDimUp>>>(my_trainer-
>d_mini_batch_data,my_rbm->getH0(),my_rbm->getDevB(),my_rbm-
>getDevW(), my_rbm->getHrand());

	//Calculate V1
	downPass<<<blockDim,threadDimDown>>>(my_rbm->getVX(),my_rbm-
>getH0(),my_rbm->getDevA(),my_rbm->getDevWT());

	//Iterate over gibbs steps HX and VX
	for (int g=1;g<STEPS;g++)
	{
		curandGenerateUniform(d_rand, (float *) my_rbm-
>getHrand(),my_rbm->getHSize()*my_trainer->getNumFantasy());
		upPass<<<blockDim,threadDimUp>>>(my_rbm->getVX(),my_rbm-
>getHX(),my_rbm->getDevB(),my_rbm->getDevW(), my_rbm->getHrand());
		downPass<<<blockDim,threadDimDown>>>(my_rbm-
>getVX(),my_rbm->getHX(),my_rbm->getDevA(),my_rbm->getDevWT());
	}

	//Calculate HX (probabilities)
	upPassProb<<<blockDim,threadDimUp>>>(my_rbm->getVX(),my_rbm-
>getHX(),my_rbm->getDevB(),my_rbm->getDevW());

	update_params();

	return;
}

void train_mini_persistent()
{
	//Select Batch Samples V0
	my_trainer->randBatchV();
	curandGenerateUniform(d_rand, (float *) my_rbm->getHrand(),
my_rbm->getHSize()*my_trainer->getNumFantasy());

	//Calculate H0
	dim3 blockDim(BLOCKS_LAYER,SAMPLES,BATCH);
	dim3 threadDimUp(my_rbm->getHSize()/BLOCKS_LAYER);
	dim3 threadDimDown(my_rbm->getVSize()/BLOCKS_LAYER);
	upPassInit<<<blockDim,threadDimUp>>>(my_trainer-
>d_mini_batch_data,my_rbm->getH0(),my_rbm->getDevB(),my_rbm-
>getDevW(), my_rbm->getHrand());

	//Calculate V1
	downPass<<<blockDim,threadDimDown>>>(my_rbm->getVX(),my_rbm-
>getHX(),my_rbm->getDevA(),my_rbm->getDevWT());

	//Iterate over gibbs steps HX and VX
	for (int g=1;g<STEPS;g++)
	{
```

```
            curandGenerateUniform(d_rand, (float *) my_rbm-
>getHrand(),my_rbm->getHSize()*my_trainer->getNumFantasy());
            upPass<<<blockDim,threadDimUp>>>(my_rbm->getVX(),my_rbm-
>getHX(),my_rbm->getDevB(),my_rbm->getDevW(), my_rbm->getHrand());
            downPass<<<blockDim,threadDimDown>>>(my_rbm-
>getVX(),my_rbm->getHX(),my_rbm->getDevA(),my_rbm->getDevWT());
      }

      //Calculate HX (probabilities for update)
      upPassProb<<<blockDim,threadDimUp>>>(my_rbm->getVX(),my_rbm-
>getHX(),my_rbm->getDevB(),my_rbm->getDevW());

      update_params();

      upPass<<<blockDim,threadDimUp>>>(my_rbm->getVX(),my_rbm-
>getHX(),my_rbm->getDevB(),my_rbm->getDevW(), my_rbm->getHrand());


      return;
}

void update_params()
{
      //Update Parameters
      dim3 threadDimUp(my_rbm->getHSize()/BLOCKS_LAYER);
      dim3 threadDimDown(my_rbm->getVSize()/BLOCKS_LAYER);
      dim3 updateBlockDim(BLOCKS_LAYER,my_rbm->getVSize());
      updateW<<<updateBlockDim,threadDimUp>>>(my_trainer-
>d_mini_batch_data,my_rbm->getVX(),my_rbm->getH0(),my_rbm-
>getHX(),my_rbm->getDevW()
                  ,my_rbm->getDevVw(), my_trainer->getMomentum(),
my_rbm->getHQ(), my_rbm->getDevDw(), my_trainer->getLearnRate());

      dim3 grid(my_rbm->getHSize()/TILE_DIM, my_rbm-
>getVSize()/TILE_DIM), threads(TILE_DIM,BLOCK_ROWS);
      transpose<<<grid,threads>>>(my_rbm->getDevW(), my_rbm-
>getDevWT());

      updateA<<<BLOCKS_LAYER,threadDimDown>>>(my_trainer-
>d_mini_batch_data,my_rbm->getVX(),my_rbm->getDevA(),my_trainer-
>getLearnRate());
      updateB<<<BLOCKS_LAYER,threadDimUp>>>(my_rbm->getH0(),my_rbm-
>getHX(),my_rbm->getDevB(), my_rbm->getHQ(), my_trainer-
>getLearnRate());

}

void convert(char* out_file)
{

      printf("Converting Data...\n");
      ofstream o_file;
```

```
	o_file.open(out_file, ios::binary);

	int loc = 0;
	if(o_file.is_open())
	{
		//Save number of training images
		int num = my_trainer->getTrainSize();
		o_file.seekp(loc);
		o_file.write((char*)&num, sizeof(int));

		loc += sizeof(int);

		loc = my_rbm->saveHDim(&o_file, loc);

		dim3 blockDim(BLOCKS_LAYER,1,1);
		dim3 threadDimUp(my_rbm->getHSize()/BLOCKS_LAYER);

		for(int i=0;i<num;i++)
		{
			//printf("Converting Image: %d\n",i);
			//Select Batch Samples V0
			my_trainer->setV(i,0);

			//Calculate H0


	upPassInitProb<<<blockDim,threadDimUp>>>(my_trainer-
>d_mini_batch_data,my_rbm->getHX(),my_rbm->getDevB(),my_rbm-
>getDevW());
			loc = my_rbm->saveH(&o_file,loc);


		}
		o_file.close();
		printf("Completed\n");
	}
	else
		printf("Failed\n");
	return;

}
/*
void saveFreeNRG()
{
	printf("Calculating Free Energy...\n");
	ofstream o_file;
	o_file.open("experiments/last_run.fnrg", ios::app);
	o_file.seekp( ios::end );
	o_file << "Epoch: " << my_trainer->getEpoch() << "\t";
	o_file << "FNRG(train): " << calcFreeNRGTrain() << "\t";
	o_file << "FNRG(valid): " << calcFreeNRGValid() << "\n";
	o_file.close();
```

```
}

float calcFreeNRGTrain()
{
      float sum=0;
      int num_seen = 0;

      float* left_sum_array, *right_sum_array;
      left_sum_array=(float*)malloc(BATCH*VSIZE*sizeof(float));
      right_sum_array=(float*)malloc(BATCH*HSIZE*sizeof(float));

      float* d_left_sum_array,*d_right_sum_array;
      dev_alloc(&d_left_sum_array, BATCH*VSIZE*sizeof(float));
      dev_alloc(&d_right_sum_array, BATCH*HSIZE*sizeof(float));

      while(num_seen<my_trainer->getValidSize())
      {
            //Grab sample and update counter
            int batch_size = my_trainer->nextBatchTrain();
            num_seen+= batch_size;

            dim3 blockDim(BLOCKS_LAYER,1,batch_size);
            dim3 threadDimDown(my_rbm->getVSize()/BLOCKS_LAYER);
            dim3 threadDimUp(my_rbm->getHSize()/BLOCKS_LAYER);

            calcLeftSum<<<blockDim,threadDimDown>>>(d_left_sum_array,
my_trainer->d_mini_batch_data, my_rbm->getDevA());
            cudaMemcpy(left_sum_array, d_left_sum_array,
VSIZE*sizeof(float), cudaMemcpyDeviceToHost);

            calcRightSum<<<blockDim,
threadDimUp>>>(d_right_sum_array, my_trainer->d_mini_batch_data,
my_rbm->getDevB(), my_rbm->getDevW());
            cudaMemcpy(right_sum_array, d_right_sum_array,
HSIZE*sizeof(float), cudaMemcpyDeviceToHost);


            for(int n=0;n<batch_size;n++)
            {
                  float left_sum=0;
                  //Add left sum
                  for (int i=0;i<my_rbm->getVSize();i++)
                  {

                        left_sum+= left_sum_array[i + n*my_rbm-
>getVSize()];
                  }

                  float right_sum = 0;
                  for(int j=0;j<my_rbm->getHSize();j++)
                  {
```

```
                        right_sum += right_sum_array[j + n*my_rbm-
>getHSize()];
                }
                sum += -left_sum - right_sum;
            }
        }

        printf("FNRG (Training) = %f\n", sum/(float)my_trainer-
>getValidSize());

        free(left_sum_array);
        free(right_sum_array);
        cudaFree(d_left_sum_array);
        cudaFree(d_right_sum_array);


        return sum/(float)my_trainer->getValidSize();
}

float calcFreeNRGValid()
{
        float sum=0;
        int num_seen = 0;
        float* left_sum_array, *right_sum_array;
        left_sum_array=(float*)malloc(BATCH*VSIZE*sizeof(float));
        right_sum_array=(float*)malloc(BATCH*HSIZE*sizeof(float));

        float* d_left_sum_array,*d_right_sum_array;
        dev_alloc(&d_left_sum_array, BATCH*VSIZE*sizeof(float));
        dev_alloc(&d_right_sum_array, BATCH*HSIZE*sizeof(float));

        while(num_seen<my_trainer->getValidSize())
        {
            //Grab sample and update counter
            int batch_size = my_trainer->nextBatchValid();
            num_seen+= batch_size;

            dim3 blockDim(BLOCKS_LAYER,1,batch_size);
            dim3 threadDimDown(my_rbm->getVSize()/BLOCKS_LAYER);
            dim3 threadDimUp(my_rbm->getHSize()/BLOCKS_LAYER);

            calcLeftSum<<<blockDim,threadDimDown>>>(d_left_sum_array,
my_trainer->d_mini_batch_data, my_rbm->getDevA());
            cudaMemcpy(left_sum_array, d_left_sum_array,
VSIZE*sizeof(float), cudaMemcpyDeviceToHost);

            calcRightSum<<<blockDim,
threadDimUp>>>(d_right_sum_array, my_trainer->d_mini_batch_data,
my_rbm->getDevB(), my_rbm->getDevW());
            cudaMemcpy(right_sum_array, d_right_sum_array,
HSIZE*sizeof(float), cudaMemcpyDeviceToHost);
```

```
            for(int n=0;n<batch_size;n++)
            {
                  float left_sum=0;
                  //Add left sum
                  for (int i=0;i<my_rbm->getVSize();i++)
                  {

                        left_sum+= left_sum_array[i + n*my_rbm-
>getVSize()];
                  }

                  float right_sum = 0;
                  for(int j=0;j<my_rbm->getHSize();j++)
                  {
                        right_sum += right_sum_array[j + n*my_rbm-
>getHSize()];
                  }
                  sum += -left_sum - right_sum;
            }
      }

      printf("FNRG (Validation) = %f\n", sum/(float)my_trainer-
>getValidSize());

      free(left_sum_array);
      free(right_sum_array);
      cudaFree(d_left_sum_array);
      cudaFree(d_right_sum_array);


      return sum/(float)my_trainer->getValidSize();
}*/

float calcFreeNRGTrain()
{
      float* a = (float*)malloc(my_rbm->getVSize() * sizeof(float));
      float* b = (float*)malloc(my_rbm->getHSize() * sizeof(float));
      float* w = (float*)malloc(my_rbm->getHSize() * my_rbm-
>getVSize() * sizeof(float));
      cudaMemcpy(a,my_rbm->getDevA(), my_rbm-
>getVSize()*sizeof(float),cudaMemcpyDeviceToHost);
      cudaMemcpy(b, my_rbm->getDevB(), my_rbm-
>getHSize()*sizeof(float), cudaMemcpyDeviceToHost);
      cudaMemcpy(w,my_rbm->getDevW(),my_rbm->getHSize() * my_rbm-
>getVSize() * sizeof(float), cudaMemcpyDeviceToHost);

      float* train_data = my_trainer->getHostData();

      float sum = 0;

      //iterate valid length up through training data
```

```
	for(int n=0;n<my_trainer->getValidSize();n++)
	{

		float left_sum = 0;
		for (int i=0;i<my_rbm->getVSize();i++)
		{
			left_sum+= a[i]*train_data[n*my_rbm->getVSize() +
i];
		}

		float right_sum = 0;
		for(int j=0;j<my_rbm->getHSize();j++)
		{
			float xj= b[j];
			for (int i=0;i<my_rbm->getVSize();i++)
			{
				xj+= train_data[n*my_rbm->getVSize() +
i]*w[(i*my_rbm->getHSize())+j];
			}
			right_sum+=log(1 + exp(xj));
		}

		sum += -left_sum - right_sum;
	}

	printf("FNRG (Training) = %f\n", sum/(float)my_trainer-
>getValidSize());
	return sum/(float)my_trainer->getValidSize();



}

float calcFreeNRGValid()
{
	float* a = (float*)malloc(my_rbm->getVSize() * sizeof(float));
	float* b = (float*)malloc(my_rbm->getHSize() * sizeof(float));
	float* w = (float*)malloc(my_rbm->getHSize() * my_rbm-
>getVSize() * sizeof(float));
	cudaMemcpy(a,my_rbm->getDevA(), my_rbm-
>getVSize()*sizeof(float),cudaMemcpyDeviceToHost);
	cudaMemcpy(b, my_rbm->getDevB(), my_rbm-
>getHSize()*sizeof(float), cudaMemcpyDeviceToHost);
	cudaMemcpy(w,my_rbm->getDevW(),my_rbm->getHSize() * my_rbm-
>getVSize() * sizeof(float), cudaMemcpyDeviceToHost);

	float* train_data = my_trainer->getHostData();

	float sum = 0;

	//iterate valid length up through training data
```

```
      for(int n=my_trainer->getTrainSize() - 1;n>=my_trainer-
>getTrainSize() - my_trainer->getValidSize();n--)
      {

            float left_sum = 0;
            for (int i=0;i<my_rbm->getVSize();i++)
            {
                  left_sum+= a[i]*train_data[n*my_rbm->getVSize() +
i];
            }

            float right_sum = 0;
            for(int j=0;j<my_rbm->getHSize();j++)
            {
                  float xj= b[j];
                  for (int i=0;i<my_rbm->getVSize();i++)
                  {
                        xj+= train_data[n*my_rbm->getVSize() +
i]*w[(i*my_rbm->getHSize())+j];
                  }
                  right_sum+=log(1 + exp(xj));
            }

            sum += -left_sum - right_sum;
      }

      printf("FNRG (Validation) = %f\n", sum/(float)my_trainer-
>getValidSize());
      return sum/(float)my_trainer->getValidSize();

}

void saveFreeNRG()
{
      printf("Calculating Free Energy...\n");
      ofstream o_file;
      o_file.open("experiments/last_run_train.fnrg", ios::app);
      o_file.seekp( ios::end );
      //o_file << "Epoch: " << my_trainer->getEpoch() << "\t";
      o_file << calcFreeNRGTrain() << ",";
      o_file.close();

      o_file.open("experiments/last_run_valid.fnrg", ios::app);
      o_file << calcFreeNRGValid() << ",";
      o_file.close();
}

/*=================================================================
=
 *               CUDA FUNCTIONS
```

```
================================================================*
/

/* ----------------------------------------------------------------
 *         UP PASS INIT
 * Initial V0->H0 pass. This is necessarily different because
 * all fantasy particles use the same initial V0.
 *
 * v0 | float* | Training examples
 * h0 | float* | Hidden Layers to calculate
 * b  | float* | Bias to hidden units
 * w  | float* | Weights
 * rnd| float* | Random vectors to compete H prob to
 ----------------------------------------------------------------
*/
__global__ void upPassInit(float* v0, float* h0, float* b, float* w,
float* rnd)
{
      int h_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
      int g_off = blockIdx.y;
      int b_off = blockIdx.z;
      int t_off = ( (b_off * gridDim.y + g_off) * HSIZE ) + h_idx;

      float sum = b[h_idx];
      //printf("sum = %f \n",b[h_idx]);
      for(int i=0;i<VSIZE;i++)
      {
            sum += v0[b_off*VSIZE + i] * w[ i*HSIZE + h_idx];
      }
      //printf("sum = %f \n",b[h_idx]);
      float prob = 1 / (1 + __expf(-1 * sum));

      //printf("p(H[%d]=1|v) = %f > %f\n",h_idx, prob, rnd[h_idx +
b_offset]);
      h0[t_off] = (prob > rnd[t_off]);
}

__global__ void upPassInitProb(float* v0, float* h0, float* b,
float* w)
{
      int h_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
      int g_off = blockIdx.y;
      int b_off = blockIdx.z;
      int t_off = ( (b_off * gridDim.y + g_off) * HSIZE ) + h_idx;

      float sum = b[h_idx];
      //printf("sum = %f \n",b[h_idx]);
      for(int i=0;i<VSIZE;i++)
      {
            sum += v0[b_off*VSIZE + i] * w[ i*HSIZE + h_idx];
      }
```

```
      //printf("sum = %f \n",b[h_idx]);
      h0[t_off] = 1 / (1 + __expf(-1 * sum));
}

/* ---------------------------------------------------------------
 *         UP PASS
 * Any VX->HX pass. Output is Binary.
 *
 * vX | float* | Visible Layers to use
 * hX | float* | Hidden Layers to calculate
 * b  | float* | Bias to hidden units
 * w  | float* | Weights
 * rnd| float* | Random vectors to compete H prob to
 ---------------------------------------------------------------
*/
__global__ void upPass(float* vX, float* hX, float* b, float* w,
float* rnd)
{
      int h_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
      int g_off = blockIdx.y;
      int b_off = blockIdx.z * gridDim.y;
      int t_off = ( (b_off + g_off) * HSIZE ) + h_idx;

      float sum = b[h_idx];
      //printf("sum = %f \n",b[h_idx]);
      for(int i=0;i<VSIZE;i++)
      {
            sum += vX[(b_off + g_off)*VSIZE + i] * w[ i*HSIZE +
h_idx];
      }
      //printf("sum = %f \n",b[h_idx]);
      float prob = 1 / (1 + __expf(-1 * sum));

      //printf("p(H[%d]=1|v) = %f > %f\n",h_idx, prob, rnd[h_idx +
b_offset]);
      hX[t_off] = (prob > rnd[t_off]);
}

/* ---------------------------------------------------------------
 *         UP PASS PROB
 * Final VX->HX pass. Output is probability.
 *
 * vX | float* | Visible Layers to use
 * hX | float* | Hidden Layers to calculate
 * b  | float* | Bias to hidden units
 * w  | float* | Weights
 ---------------------------------------------------------------
*/
__global__ void upPassProb(float* vX, float* hX, float* b, float* w)
{
      int h_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
      int g_off = blockIdx.y;
```

```
      int b_off = blockIdx.z * gridDim.y;
      int t_off = ( (b_off + g_off) * HSIZE ) + h_idx;

      float sum = b[h_idx];
      //printf("sum = %f \n",b[h_idx]);
      for(int i=0;i<VSIZE;i++)
      {
            sum += vX[(b_off + g_off)*VSIZE + i] * w[ i*HSIZE +
h_idx];
      }
      //printf("sum = %f \n",b[h_idx]);
      hX[t_off] = 1 / (1 + __expf(-1 * sum));
}

/* ---------------------------------------------------------------
 *         DOWN PASS
 * Any HX->VX pass. Output is probability.
 *
 * vX | float* | Visible Layers to calculate
 * hX | float* | Hidden Layers to use
 * a  | float* | Bias to visible units
 * wt | float* | Weights Transposed
 ---------------------------------------------------------------
*/
__global__ void downPass(float* vX, float* hX, float* a, float* wt)
{
      int v_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
      int g_off = blockIdx.y;
      int b_off = blockIdx.z * gridDim.y;
      int t_off = ( (b_off + g_off) * VSIZE ) + v_idx;

      float sum = a[v_idx];
      //printf("sum = %f \n",b[h_idx]);
      for(int i=0;i<HSIZE;i++)
      {
            //sum += hX[b_off + g_off + i] * w[ i*512 + v_idx];
            sum += hX[(b_off + g_off)*HSIZE + i] * wt[ i*VSIZE +
v_idx];
      }
      //printf("sum = %f \n",b[h_idx]);
      vX[t_off] = 1 / (1 + __expf(-1 * sum));

}

/* ---------------------------------------------------------------
 *         UPDATE W
 * Calculates the change to the weights
 *
 * v0            | float* | Visible layer from data
 * vX            | float* | Final Visible layer from model
 * h0            | float* | Hidden layer one pass from data
 * hX            | float* | Hidden layer from model
```

```
 * w             | float* | Weights
 * l_rate  | float  | learning rate
 ----------------------------------------------------------------------
*/
__global__ void updateW(float* v0, float* vX, float* h0, float* hX,
float* w, float* vel_w, float momentum, float* q, float* dw, float
l_rate)
{
      int h_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
      int v_idx = (blockIdx.y);
      int v_offset = v_idx * blockDim.x * gridDim.x;

      float delta = 0.0;
      float sum_h = 0.0;

      for(int batch=0;batch<BATCH;batch++)
      {
            for(int gibbs=0;gibbs<SAMPLES;gibbs++)
            {
                  int h_off = h_idx + batch*SAMPLES*HSIZE +
gibbs*HSIZE;
                  int v_off = v_idx + batch*SAMPLES*VSIZE +
gibbs*VSIZE;

                  delta += (v0[v_idx + batch*VSIZE] * h0[h_off]) -
(vX[v_off] * hX[h_off]);
                  sum_h += (hX[h_off]);
            }
      }

      //Calculate probability estimate
      q[h_idx] = ((.95)*q[h_idx]) + (1-.95)*(sum_h / (BATCH *
SAMPLES));



      //if(v_idx == 200 && h_idx < 5)
            //printf("Q = %f\n", q[h_idx]);
      //if(h_idx + v_offset == 555)
            //printf("w[%d]=%f += %f\n", h_idx + v_offset, w[h_idx +
v_offset], delta);

      //VELOCITY
      vel_w[h_idx + v_offset] = momentum * vel_w[h_idx + v_offset] +
( (delta * l_rate) / (SAMPLES * BATCH) );

      //DECAY
      float decay = (0.0005 * w[h_idx + v_offset] ) * l_rate;


      //SPARSITY
      // = penalty * ( probability estimation - probability target)
```

```
	float sparsity = 0.0001 * (q[h_idx]-0.1);

	dw[h_idx + v_offset] = (vel_w[h_idx + v_offset] - decay -
sparsity);
	w[h_idx + v_offset] += (vel_w[h_idx + v_offset] - decay -
sparsity);

	//VELOCITY AND SPARSITY ONLY
	//dw[h_idx + v_offset] = (vel_w[h_idx + v_offset]  -
sparsity);
	//w[h_idx + v_offset] += (vel_w[h_idx + v_offset]  -
sparsity);

	//w[h_idx + v_offset] += (delta * l_rate) / (SAMPLES * BATCH);
	//w[h_idx + v_offset] = delta;

	//dw[h_idx + v_offset] = ((delta * l_rate) / (SAMPLES * BATCH)
) - (decay * l_rate);
	//w[h_idx + v_offset] += ((delta * l_rate) / (SAMPLES * BATCH)
) - (decay * l_rate);
}

/* ------------------------------------------------------------
 *         UPDATE A
 * Calculates the change to the visible bias
 *
 * v0           | float* | Visible layer from data
 * vX           | float* | Final Visible layer from model
 * a            | float* | Visible bias
 * l_rate  | float  | learning rate
 ------------------------------------------------------------
*/
__global__ void updateA(float* v0, float* vX, float* a, float
l_rate)
{
	int v_idx = (blockIdx.x * blockDim.x) + threadIdx.x;

	float delta = 0.0;

	for(int batch=0;batch<BATCH;batch++)
	{
		for(int gibbs=0;gibbs<SAMPLES;gibbs++)
		{
			int v_off = v_idx + batch*SAMPLES*VSIZE +
gibbs*VSIZE;
			delta += (v0[v_idx + batch*VSIZE]) - (vX[v_off]);
		}
	}

	a[v_idx] += ( (delta * l_rate) / (SAMPLES * BATCH) );
}
```

```
/* ---------------------------------------------------------------
 *         UPDATE B
 * Calculates the change to the hidden bias
 *
 * h0            | float* | Hidden layer one pass from data
 * hX            | float* | Hidden layer from model
 * b             | float* | Hidden bias
 * l_rate  | float  | learning rate
 ---------------------------------------------------------------
*/
__global__ void updateB(float* h0, float* hX, float* b, float* q,
float l_rate)
{
      int h_idx = (blockIdx.x * blockDim.x) + threadIdx.x;

      float delta = 0.0;

      for(int batch=0;batch<BATCH;batch++)
      {
            for(int gibbs=0;gibbs<SAMPLES;gibbs++)
            {
                  int h_off = h_idx + batch*SAMPLES*HSIZE +
gibbs*HSIZE;
                  delta += (h0[h_off]) - (hX[h_off]);
            }
      }

      //float sparsity = (0.0001 * ((q[h_idx]-0.04)*(q[h_idx]-0.04))
);
      // = penalty * ( probability estimation - probability target)
      float sparsity = 0.0001 * (q[h_idx]-0.1);

      //if(h_idx < 5)
      //printf("sparsity penalty = %f\n",sparsity);

      b[h_idx] += ( (delta * l_rate) / (SAMPLES * BATCH) )  -
sparsity;
      //b[h_idx] += ( (delta * l_rate) / (SAMPLES * BATCH) );
}

/* ---------------------------------------------------------------
 *         TRANSPOSE
 * Coalesced transpose with no bank conflicts.
 *
 * w  | float* | Weights
 * wt | float* | Weights Transposed
 ---------------------------------------------------------------
*/
__global__ void transpose(float *w, float *wt)
{
    __shared__ float tile[TILE_DIM][TILE_DIM+1];
```

```
    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
    int index_in = xIndex + (yIndex)*HSIZE;

    xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
    yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
    int index_out = xIndex + (yIndex)*VSIZE;

      for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS)
      {
            tile[threadIdx.y+i][threadIdx.x] = w[index_in+i*HSIZE];
      }

      __syncthreads();

      for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS)
      {
            wt[index_out+i*VSIZE] = tile[threadIdx.x][threadIdx.y+i];
      }
}

__global__ void calcLeftSum(float* sum, float* v0, float* a)
{
      int v_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
      int g_off = 0;
      int b_off = blockIdx.z;
      int t_off = ( (b_off + g_off) * VSIZE ) + v_idx;

      sum[t_off] = a[v_idx] * v0[t_off];
}

__global__ void calcRightSum(float* sum, float* v0, float* b, float*
w)
{
      int h_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
      int g_off = blockIdx.y;
      int b_off = blockIdx.z;
      int t_off = ( (b_off + g_off) * HSIZE ) + h_idx;

      float xj = b[h_idx];
      for(int i=0;i<VSIZE;i++)
      {
            xj += v0[(b_off + g_off)*VSIZE + i] * w[ i*HSIZE +
h_idx];
      }
      sum[t_off] = __logf(1 + __expf(xj));
}
```

dbn.cu

```
#include "connection/inc/connection.h"
```

```
#include "dbn/inc/dbn.h"
#include "trainer/inc/trainer.h"
#include <curand.h>

#include "GL/freeglut.h"
#include "GL/gl.h"

using namespace utilLearn;


//NN LEARNING PARAMETERS
#define BATCH              100
#define SAMPLES            1
#define STEPS              1
#define EPOCH              400

//NN SIZE PARAMETERS
//#define VSIZE_X               28
//#define  VSIZE_Y              28
//#define VSIZE            784
#define VSIZE_X            96
#define    VSIZE_Y              96
#define VSIZE              9216

//#define  H0SIZE_X        32
//#define H0SIZE_Y         16
//#define H0SIZE           512
#define    H0SIZE_X        64
#define H0SIZE_Y      64
#define H0SIZE             4096

//#define  H1SIZE_X        32
//#define H1SIZE_Y         16
//#define H1SIZE           512
#define    H1SIZE_X        64
#define H1SIZE_Y      64
#define H1SIZE             4096

//NN MODE
#define MODE               0 //0=train 1=classification
#define TRAIN_EXAMPLE 1232

//char * const param_file_1 = "params/persistent-lvl1.rbm";
//char * const param_file_2 = "params/persistent-lvl2.rbm";
//char * const data_file = "data/train-images.idx3-ubyte";

char * const param_file_1 = "params/norb-persistent-lvl1.rbm";
char * const param_file_2 = "params/norb-persistent-lvl2.rbm";
char * const data_file = "data/smallnorb-5x46789x9x18x6x2x96x96-
training-dat.mat";

//For Transpose
```

```
#define TILE_DIM      16
#define BLOCK_ROWS          16

#define    BLOCKS_LAYER    16

//Display Functions
void train();

//Helper Functions
void train_mini();
void update_params();


//CUDA Functions
__global__ void transpose(float *w, float *wt);

__global__ void upPassInit(float* v0, float* h0, float* b, float* w,
float* rnd);
__global__ void upPass(float* vX, float* hX, float* b, float* w,
float* rnd);
__global__ void upPassProb(float* vX, float* hX, float* b, float*
w);
__global__ void downPass(float* vX, float* hX, float* a, float* wt,
float* rnd);
__global__ void downPassProb(float* vX, float* hX, float* a, float*
wt);

__global__ void updateW0T(float* v0, float* v0Pred, float* h0,
float* wt, float l_rate);
__global__ void updateW0(float* v0X, float* h0X, float* h0Pred,
float* w, float l_rate);
__global__ void updateW1(float* h0, float* h0X, float* h1, float*
h1X, float* w, float l_rate);

__global__ void updateA0(float* v0, float* v0Pred, float* a, float
l_rate);
__global__ void updateB0(float* h0X, float* h0XPred, float* b, float
l_rate);
__global__ void updateA1(float* h0, float* h0X, float* a, float
l_rate);
__global__ void updateB1(float* h1, float* h1X, float* b, float
l_rate);

//Globals
Dbn *my_dbn;
Trainer *my_trainer;
curandGenerator_t d_rand;

//float total_time;

int main(int argc, char** argv)
{
```

```
	//total_time = 0;
	//Set GPU 1 (currently not used for display)
	cudaSetDevice(1);

	//Set up basic units
	Layer* visible = new Layer(VSIZE_X, VSIZE_Y, BATCH, SAMPLES,
false);
	Connection* v_to_h0 = new Connection(VSIZE, H0SIZE);
	Layer* h0 = new Layer(H0SIZE_X, H0SIZE_Y, BATCH, SAMPLES,
true);
	Connection* h0_to_h1 = new Connection(H0SIZE, H1SIZE);
	Layer* h1 = new Layer(H1SIZE_X, H1SIZE_Y, BATCH, SAMPLES,
true);

	my_dbn = new Dbn(visible, h0, h1, v_to_h0, h0_to_h1);
	my_trainer = new Trainer(BATCH, SAMPLES, EPOCH, 0.0005, 5,
0.5, 0.0000);

	//if(my_trainer->loadTrainingData(data_file) < 0)
	if(my_trainer->loadTrainingDataMAT(data_file) < 0)
	{
		printf("An error occurred loading the training data.
Exiting...\n");
		return -1;
	}


	if(my_dbn->loadLayers(param_file_1, param_file_2) < 0)
	{
		printf("An error occurred loading the parameters.
Exiting...\n");
		return -1;
	}


	//Set up Random Initializer
	curandCreateGenerator(&d_rand, CURAND_RNG_PSEUDO_MTGP32);
	srand((unsigned)time(0));
	int seed = (rand() % 1000);
	curandSetPseudoRandomGeneratorSeed(d_rand, seed);

	//Set up GLUT Parameters
	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_SINGLE);
	glutInitWindowSize(500,500);
	glutInitWindowPosition(250,250);
	glutCreateWindow("Deep Belief Network");

	glutDisplayFunc(train);

	glutMainLoop();
```

```
      return 0;

}

/*================================================================
=
 *               DISPLAY FUNCTIONS

================================================================*
/

//Trains the RBM with a selected visual feeback
void train()
{
      glClearColor(1.0,1.0,1.0,1.0);
      glClear(GL_COLOR_BUFFER_BIT);
      glOrtho(-1.0,1.0,-1.0,1.0,-1.0,1.0);


      //cudaEvent_t start, stop;
      //float time;
      //cudaEventCreate(&start);
      //cudaEventCreate(&stop);

      //cudaEventRecord(start, 0);
      train_mini();
      //cudaEventRecord(stop, 0);
      //cudaEventSynchronize(stop);
      //cudaEventElapsedTime(&time, start, stop);

      //printf("Time for mini_batch (100): %f ms\n", time);
      //total_time+=time;


      my_trainer->setV(TRAIN_EXAMPLE,0);
      dim3 blockDim(BLOCKS_LAYER,SAMPLES,BATCH);

      //V->H0
      dim3 threadDimH0(my_dbn->getH0Size()/BLOCKS_LAYER);
      curandGenerateUniform(d_rand, (float *) my_dbn->getH0Rand(),
my_dbn->getH0Size()*my_trainer->getNumFantasy());
      upPassInit<<<blockDim,threadDimH0>>>(my_trainer-
>d_mini_batch_data,my_dbn->getH0In(),my_dbn->getDevB_0(),my_dbn-
>getDevW_0(), my_dbn->getH0Rand());
      //H0->H1
      dim3 threadDimH1(my_dbn->getH1Size()/BLOCKS_LAYER);
      curandGenerateUniform(d_rand, (float *) my_dbn->getH1Rand(),
my_dbn->getH1Size()*my_trainer->getNumFantasy());
      upPass<<<blockDim,threadDimH1>>>(my_dbn->getH0In(),my_dbn-
>getH1In(),my_dbn->getDevB_1(),my_dbn->getDevW_1(), my_dbn-
>getH1Rand());
      //H0<-H1
```

```
	curandGenerateUniform(d_rand, (float *) my_dbn->getH0Rand(),
my_dbn->getH0Size()*my_trainer->getNumFantasy());
	downPass<<<blockDim,threadDimH0>>>(my_dbn->getH0(),my_dbn-
>getH1In(),my_dbn->getDevA_1(),my_dbn->getDevWT_1(), my_dbn-
>getH0Rand());
	//H0f->H1f
	curandGenerateUniform(d_rand, (float *) my_dbn-
>getH1Rand(),my_dbn->getH1Size()*my_trainer->getNumFantasy());
	upPass<<<blockDim,threadDimH1>>>(my_dbn->getH0(),my_dbn-
>getH1(),my_dbn->getDevB_1(),my_dbn->getDevW_1(), my_dbn-
>getH1Rand());
	//H0f<-H1f
	curandGenerateUniform(d_rand, (float *) my_dbn->getH0Rand(),
my_dbn->getH0Size()*my_trainer->getNumFantasy());
	downPass<<<blockDim,threadDimH0>>>(my_dbn->getH0(),my_dbn-
>getH1(),my_dbn->getDevA_1(),my_dbn->getDevWT_1(), my_dbn-
>getH0Rand());
	//Vf<-H0f
	dim3 threadDimV(my_dbn->getVSize()/BLOCKS_LAYER);
	downPassProb<<<blockDim,threadDimV>>>(my_dbn->getVX(),my_dbn-
>getH0(),my_dbn->getDevA_0(),my_dbn->getDevWT_0());

	my_dbn->showVX(0);


	glFlush();


	//Update training status
	my_trainer->incN();
	if(my_trainer->epochComplete())
	{
		printf("Epoch %d Complete!\n", my_trainer->getEpoch());
		my_dbn->saveLayers(param_file_1, param_file_2);
	}

	if(!my_trainer->trainComplete())
		glutPostRedisplay();
	else
	{
		printf("Training run complete!\n");
		//printf("Average Time for mini_batch(100): %f ms\n",
total_time / ( (float)(my_trainer->getTrainSize() - my_trainer-
>getValidSize()) / (float)BATCH) );
	}

}

//Trains a single mini-batch
void train_mini()
{
	//Select Batch Samples V0
```

```
      my_trainer->randBatchV();


      //WAKE PHASE
      dim3 blockDim(BLOCKS_LAYER,SAMPLES,BATCH);

      //V->H0
      dim3 threadDimH0(my_dbn->getH0Size()/BLOCKS_LAYER);
      curandGenerateUniform(d_rand, (float *) my_dbn->getH0Rand(),
my_dbn->getH0Size()*my_trainer->getNumFantasy());
      upPassInit<<<blockDim,threadDimH0>>>(my_trainer-
>d_mini_batch_data,my_dbn->getH0In(),my_dbn->getDevB_0(),my_dbn-
>getDevW_0(), my_dbn->getH0Rand());

      //H0->H1
      dim3 threadDimH1(my_dbn->getH1Size()/BLOCKS_LAYER);
      curandGenerateUniform(d_rand, (float *) my_dbn->getH1Rand(),
my_dbn->getH1Size()*my_trainer->getNumFantasy());
      upPass<<<blockDim,threadDimH1>>>(my_dbn->getH0In(),my_dbn-
>getH1In(),my_dbn->getDevB_1(),my_dbn->getDevW_1(), my_dbn-
>getH1Rand());

      //H0<-H1
      curandGenerateUniform(d_rand, (float *) my_dbn->getH0Rand(),
my_dbn->getH0Size()*my_trainer->getNumFantasy());
      downPass<<<blockDim,threadDimH0>>>(my_dbn->getH0(),my_dbn-
>getH1In(),my_dbn->getDevA_1(),my_dbn->getDevWT_1(), my_dbn-
>getH0Rand());

      //Additional Gibbs steps
      for (int g=1;g<STEPS;g++)
      {
            curandGenerateUniform(d_rand, (float *) my_dbn-
>getH1Rand(),my_dbn->getH1Size()*my_trainer->getNumFantasy());
            upPass<<<blockDim,threadDimH1>>>(my_dbn->getH0(),my_dbn-
>getH1(),my_dbn->getDevB_1(),my_dbn->getDevW_1(), my_dbn-
>getH1Rand());
            curandGenerateUniform(d_rand, (float *) my_dbn-
>getH0Rand(),my_dbn->getH0Size()*my_trainer->getNumFantasy());
            downPass<<<blockDim,threadDimH0>>>(my_dbn-
>getH0(),my_dbn->getH1(),my_dbn->getDevA_1(),my_dbn->getDevWT_1(),
my_dbn->getH0Rand());
      }
      //SLEEP PHASE

      //H0f->H1f
      curandGenerateUniform(d_rand, (float *) my_dbn-
>getH1Rand(),my_dbn->getH1Size()*my_trainer->getNumFantasy());
      upPass<<<blockDim,threadDimH1>>>(my_dbn->getH0(),my_dbn-
>getH1(),my_dbn->getDevB_1(),my_dbn->getDevW_1(), my_dbn-
>getH1Rand());
```

```
	//H0f<-H1f
	curandGenerateUniform(d_rand, (float *) my_dbn->getH0Rand(),
my_dbn->getH0Size()*my_trainer->getNumFantasy());
	downPass<<<blockDim,threadDimH0>>>(my_dbn->getH0(),my_dbn-
>getH1(),my_dbn->getDevA_1(),my_dbn->getDevWT_1(), my_dbn-
>getH0Rand());

	//Vf<-H0f
	dim3 threadDimV(my_dbn->getVSize()/BLOCKS_LAYER);
	downPassProb<<<blockDim,threadDimV>>>(my_dbn->getVX(),my_dbn-
>getH0(),my_dbn->getDevA_0(),my_dbn->getDevWT_0());

	//Predictions
	downPassProb<<<blockDim,threadDimV>>>(my_dbn-
>getVPred(),my_dbn->getH0In(),my_dbn->getDevA_0(),my_dbn-
>getDevWT_0());

	upPassProb<<<blockDim, threadDimH0>>>(my_dbn->getVX(), my_dbn-
>getH0Pred(), my_dbn->getDevB_0(), my_dbn->getDevW_0());

	update_params();

	return;
}

void update_params()
{
	//Update Parameters
	dim3 threadDimV(my_dbn->getVSize()/BLOCKS_LAYER);
	dim3 threadDimH0(my_dbn->getH0Size()/BLOCKS_LAYER);
	dim3 threadDimH1(my_dbn->getH1Size()/BLOCKS_LAYER);

	dim3 updateBlockW0(BLOCKS_LAYER,my_dbn->getVSize());
	dim3 updateBlockW0T(BLOCKS_LAYER,my_dbn->getH0Size());
	dim3 updateBlockW1(BLOCKS_LAYER,my_dbn->getH0Size());

	updateW0<<<updateBlockW0,threadDimH0>>>(my_dbn-
>getVX(),my_dbn->getH0(),my_dbn->getH0Pred(),my_dbn->getDevW_0()
		,my_trainer->getLearnRate());

	updateW0T<<<updateBlockW0T,threadDimV>>>(my_trainer-
>d_mini_batch_data,my_dbn->getVX(),my_dbn->getH0In(),my_dbn-
>getDevWT_0()
		,my_trainer->getLearnRate());

	updateW1<<<updateBlockW1,threadDimH1>>>(my_dbn-
>getH0In(),my_dbn->getH0(),my_dbn->getH1In(),my_dbn->getH1(),my_dbn-
>getDevW_1()
		,my_trainer->getLearnRate());
	dim3 grid(my_dbn->getH1Size()/TILE_DIM, my_dbn-
>getH0Size()/TILE_DIM), threads(TILE_DIM,BLOCK_ROWS);
```

```
	transpose<<<grid,threads>>>(my_dbn->getDevW_1(), my_dbn-
>getDevWT_1());

	updateA0<<<BLOCKS_LAYER,threadDimV>>>(my_trainer-
>d_mini_batch_data, my_dbn->getVX(), my_dbn->getDevA_0(),
my_trainer->getLearnRate());
	updateB0<<<BLOCKS_LAYER,threadDimH0>>>(my_dbn->getH0In(),
my_dbn->getH0(), my_dbn->getDevB_0(), my_trainer->getLearnRate());
	updateA1<<<BLOCKS_LAYER,threadDimH0>>>(my_dbn->getH0In(),
my_dbn->getH0(), my_dbn->getDevA_1(), my_trainer->getLearnRate());
	updateB1<<<BLOCKS_LAYER,threadDimH1>>>(my_dbn->getH1In(),
my_dbn->getH1(), my_dbn->getDevB_1(), my_trainer->getLearnRate());

}

/*=================================================================
=
 *               CUDA FUNCTIONS

================================================================*
/

/* ----------------------------------------------------------------
 *         UP PASS INIT
 * Initial V0->H0 pass. This is necessarily different because
 * all fantasy particles use the same initial V0.
 *
 * v0 | float* | Training examples
 * h0 | float* | Hidden Layers to calculate
 * b  | float* | Bias to hidden units
 * w  | float* | Weights
 * rnd| float* | Random vectors to compete H prob to
 ----------------------------------------------------------------
*/
__global__ void upPassInit(float* v0, float* h0, float* b, float* w,
float* rnd)
{
	int h_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
	int g_off = blockIdx.y;
	int b_off = blockIdx.z;
	int t_off = ( (b_off * gridDim.y + g_off) * H0SIZE ) + h_idx;

	float sum = b[h_idx];
	//printf("sum = %f \n",b[h_idx]);
	for(int i=0;i<VSIZE;i++)
	{
		sum += v0[b_off*VSIZE + i] * w[ i*H0SIZE + h_idx];
	}
	//printf("sum = %f \n",b[h_idx]);
	float prob = 1 / (1 + __expf(-1 * sum));
```

```
      //printf("p(H[%d]=1|v) = %f > %f\n",h_idx, prob, rnd[h_idx +
b_offset]);
      h0[t_off] = (prob > rnd[t_off]);
}

/* ---------------------------------------------------------------
 *         UP PASS
 * Any VX->HX pass. Output is Binary.
 *
 * vX | float* | Visible Layers to use
 * hX | float* | Hidden Layers to calculate
 * b  | float* | Bias to hidden units
 * w  | float* | Weights
 * rnd| float* | Random vectors to compete H prob to
 ---------------------------------------------------------------
*/
__global__ void upPass(float* vX, float* hX, float* b, float* w,
float* rnd)
{
      int h_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
      int g_off = blockIdx.y;
      int b_off = blockIdx.z * gridDim.y;
      int t_off = ( (b_off + g_off) * H1SIZE ) + h_idx;

      float sum = b[h_idx];
      //printf("sum = %f \n",b[h_idx]);
      for(int i=0;i<H0SIZE;i++)
      {
            sum += vX[(b_off + g_off)*H0SIZE + i] * w[ i*H1SIZE +
h_idx];
      }
      //printf("sum = %f \n",b[h_idx]);
      float prob = 1 / (1 + __expf(-1 * sum));

      //printf("p(H[%d]=1|v) = %f > %f\n",h_idx, prob, rnd[h_idx +
b_offset]);
      hX[t_off] = (prob > rnd[t_off]);
}

/* ---------------------------------------------------------------
 *         UP PASS PROB
 * Final VX->HX pass. Output is probability.
 *
 * vX | float* | Visible Layers to use
 * hX | float* | Hidden Layers to calculate
 * b  | float* | Bias to hidden units
 * w  | float* | Weights
 ---------------------------------------------------------------
*/
__global__ void upPassProb(float* vX, float* hX, float* b, float* w)
{
      int h_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
```

```
	int g_off = blockIdx.y;
	int b_off = blockIdx.z * gridDim.y;
	int t_off = ( (b_off + g_off) * H0SIZE ) + h_idx;

	float sum = b[h_idx];
	//printf("sum = %f \n",b[h_idx]);
	for(int i=0;i<VSIZE;i++)
	{
		sum += vX[(b_off + g_off)*VSIZE + i] * w[ i*H0SIZE +
h_idx];
	}
	//printf("sum = %f \n",b[h_idx]);
	hX[t_off] = 1 / (1 + __expf(-1 * sum));
}

/* ------------------------------------------------------------------
 *		DOWN PASS
 * Any HX->VX pass. Output is probability.
 *
 * vX | float* | Visible Layers to calculate
 * hX | float* | Hidden Layers to use
 * a  | float* | Bias to visible units
 * wt | float* | Weights Transposed
 ------------------------------------------------------------------
*/
__global__ void downPass(float* vX, float* hX, float* a, float* wt,
float* rnd)
{
	int v_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
	int g_off = blockIdx.y;
	int b_off = blockIdx.z * gridDim.y;
	int t_off = ( (b_off + g_off) * H0SIZE ) + v_idx;

	float sum = a[v_idx];
	//printf("sum = %f \n",b[h_idx]);
	for(int i=0;i<H1SIZE;i++)
	{
		//sum += hX[b_off + g_off + i] * w[ i*512 + v_idx];
		sum += hX[(b_off + g_off)*H1SIZE + i] * wt[ i*H0SIZE +
v_idx];
	}
	//printf("sum = %f \n",b[h_idx]);
	float prob = 1 / (1 + __expf(-1 * sum));

	vX[t_off] = (prob > rnd[t_off]);
}

__global__ void downPassProb(float* vX, float* hX, float* a, float*
wt)
{
	int v_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
	int g_off = blockIdx.y;
```

```
	int b_off = blockIdx.z * gridDim.y;
	int t_off = ( (b_off + g_off) * VSIZE ) + v_idx;

	float sum = a[v_idx];
	//printf("sum = %f \n",b[h_idx]);
	for(int i=0;i<H0SIZE;i++)
	{
		//sum += hX[b_off + g_off + i] * w[ i*512 + v_idx];
		sum += hX[(b_off + g_off)*H0SIZE + i] * wt[ i*VSIZE +
v_idx];
	}
	//printf("sum = %f \n",b[h_idx]);
	vX[t_off] = 1 / (1 + __expf(-1 * sum));
}

__global__ void updateW0T(float* v0, float* v0Pred, float* h0,
float* wt, float l_rate)
{
	int v_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
	int h_idx = (blockIdx.y);
	int h_offset = h_idx * blockDim.x * gridDim.x;

	float delta = 0.0;

	for(int batch=0;batch<BATCH;batch++)
	{
		for(int gibbs=0;gibbs<SAMPLES;gibbs++)
		{
			int h_off = h_idx + batch*SAMPLES*H0SIZE +
gibbs*H0SIZE;
			int v_off = v_idx + batch*SAMPLES*VSIZE +
gibbs*VSIZE;

			delta += h0[h_off] * (v0[v_idx + batch*VSIZE]-
v0Pred[v_off]);
		}
	}
	//DECAY
	float decay = (0.0005 * wt[v_idx + h_offset] ) * l_rate;

	wt[v_idx + h_offset] += ((delta * l_rate) / (SAMPLES * BATCH))
- decay;
}

__global__ void updateW0(float* v0X, float* h0X, float* h0Pred,
float* w, float l_rate)
{
	int h_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
	int v_idx = (blockIdx.y);
	int v_offset = v_idx * blockDim.x * gridDim.x;

	float delta = 0.0;
```

```
      for(int batch=0;batch<BATCH;batch++)
      {
            for(int gibbs=0;gibbs<SAMPLES;gibbs++)
            {
                  int h_off = h_idx + batch*SAMPLES*H0SIZE +
gibbs*H0SIZE;
                  int v_off = v_idx + batch*SAMPLES*VSIZE +
gibbs*VSIZE;

                  delta += v0X[v_off] * (h0X[h_off]- h0Pred[h_off]);
            }
      }
      //DECAY
      float decay = (0.0005 * w[h_idx + v_offset] ) * l_rate;

      w[h_idx + v_offset] += ((delta * l_rate) / (SAMPLES * BATCH))
- decay;
}


__global__ void updateW1(float* h0, float* h0X, float* h1, float*
h1X, float* w, float l_rate)
{
      {
            int h_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
            int v_idx = (blockIdx.y);
            int v_offset = v_idx * blockDim.x * gridDim.x;

            float delta = 0.0;

            for(int batch=0;batch<BATCH;batch++)
            {
                  for(int gibbs=0;gibbs<SAMPLES;gibbs++)
                  {
                        int h_off = h_idx + batch*SAMPLES*H1SIZE +
gibbs*H1SIZE;
                        int v_off = v_idx + batch*SAMPLES*H0SIZE +
gibbs*H0SIZE;

                        delta += (h0[v_off] * h1[h_off]) - (h0X[v_off]
* h1X[h_off]);
                  }
            }
            //DECAY
            float decay = (0.0005 * w[h_idx + v_offset] ) * l_rate;

            w[h_idx + v_offset] += ((delta * l_rate) / (SAMPLES *
BATCH)) - decay;
      }
}
```

```
__global__ void updateA0(float* v0, float* v0Pred, float* a, float
l_rate)
{
      int v_idx = (blockIdx.x * blockDim.x) + threadIdx.x;

      float delta = 0.0;

      for(int batch=0;batch<BATCH;batch++)
      {
            for(int gibbs=0;gibbs<SAMPLES;gibbs++)
            {
                  int v_off = v_idx + batch*SAMPLES*VSIZE +
gibbs*VSIZE;
                  delta += (v0[v_idx + batch*VSIZE]) -
(v0Pred[v_off]);
            }
      }

      a[v_idx] += ( (delta * l_rate) / (SAMPLES * BATCH) );
}

__global__ void updateB0(float* h0X, float* h0XPred, float* b, float
l_rate)
{
      int h_idx = (blockIdx.x * blockDim.x) + threadIdx.x;

      float delta = 0.0;

      for(int batch=0;batch<BATCH;batch++)
      {
            for(int gibbs=0;gibbs<SAMPLES;gibbs++)
            {
                  int h_off = h_idx + batch*SAMPLES*H0SIZE +
gibbs*H0SIZE;
                  delta += (h0X[h_off]) - (h0XPred[h_off]);
            }
      }

      b[h_idx] += ( (delta * l_rate) / (SAMPLES * BATCH) );
}
__global__ void updateA1(float* h0, float* h0X, float* a, float
l_rate)
{
      int v_idx = (blockIdx.x * blockDim.x) + threadIdx.x;

      float delta = 0.0;

      for(int batch=0;batch<BATCH;batch++)
      {
            for(int gibbs=0;gibbs<SAMPLES;gibbs++)
            {
```

```
                int v_off = v_idx + batch*SAMPLES*H0SIZE +
gibbs*H0SIZE;
                delta += (h0[v_off]) - (h0X[v_off]);
           }
     }

     a[v_idx] += ( (delta * l_rate) / (SAMPLES * BATCH) );
}

__global__ void updateB1(float* h1, float* h1X, float* b, float
l_rate)
{
     int h_idx = (blockIdx.x * blockDim.x) + threadIdx.x;

     float delta = 0.0;

     for(int batch=0;batch<BATCH;batch++)
     {
          for(int gibbs=0;gibbs<SAMPLES;gibbs++)
          {
                int h_off = h_idx + batch*SAMPLES*H1SIZE +
gibbs*H1SIZE;
                delta += (h1[h_off]) - (h1X[h_off]);
           }
     }

     b[h_idx] += ( (delta * l_rate) / (SAMPLES * BATCH) );
}

/* ------------------------------------------------------------------
 *         TRANSPOSE
 * Coalesced transpose with no bank conflicts.
 *
 * w  | float* | Weights
 * wt | float* | Weights Transposed
 ------------------------------------------------------------------
*/
__global__ void transpose(float *w, float *wt)
{
    __shared__ float tile[TILE_DIM][TILE_DIM+1];

    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
    int index_in = xIndex + (yIndex)*H1SIZE;

    xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
    yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
    int index_out = xIndex + (yIndex)*H0SIZE;

     for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS)
     {
          tile[threadIdx.y+i][threadIdx.x] = w[index_in+i*H1SIZE];
```

```
      }

      __syncthreads();

      for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS)
      {
            wt[index_out+i*H0SIZE] =
tile[threadIdx.x][threadIdx.y+i];
      }
}
```

nn.cu

```
#include "connection/inc/connection.h"
#include "nn/inc/nn.h"
#include "trainer/inc/trainer.h"

#include "GL/freeglut.h"
#include "GL/gl.h"

//NN LEARNING PARAMETERS
#define BATCH              20
#define SAMPLES            1
#define EPOCH              200

//NN SIZE PARAMETERS
//#define VSIZE_X               28
//#define  VSIZE_Y              28
//#define VSIZE              784
#define VSIZE_X            96
#define    VSIZE_Y               96
#define VSIZE              9216

//#define  H0SIZE_X          32
//#define H0SIZE_Y          16
//#define H0SIZE            512
#define    H0SIZE_X          64
#define H0SIZE_Y      64
#define H0SIZE               4096

//#define  H1SIZE_X          32
//#define H1SIZE_Y          16
//#define H1SIZE            512
#define    H1SIZE_X          64
#define H1SIZE_Y      64
#define H1SIZE               4096

#define CLASS              5

//NN MODE
#define MODE               0 //0=train 1=classification
```

```
#define LOAD               1 //0=fresh 1=lower_params 2=complete
#define TRAIN_EXAMPLE 2

#define    BLOCKS_LAYER    16

//Display Functions
void train();
void test();


void classifyTrain();
void classifyValid();
void validationError();

//Helper Functions
int  load_from_rbm(char* rbm_file, Connection* con);
void train_mini();

//File list
//char * const param_file_v_h0 = "params/persistent-lvl1.rbm";
//char * const param_file_h0_h1 = "params/persistent-lvl2.rbm";
//char * const param_file_complete = "params/persistent-full.nn";

char * const param_file_v_h0 = "params/norb-persistent-lvl1.rbm";
char * const param_file_h0_h1 = "params/norb-persistent-lvl2.rbm";
char * const param_file_complete = "params/norb-full.nn";


//char * const param_file_v_h0 = "params/persistent-lvl1.rbm";
//char * const param_file_h0_h1 = "params/persistent-lvl2.rbm";

//char * const param_file_complete = "params/dbn-full.nn";

//char * const label_file = "data/train-labels.idx1-ubyte";
//char * const data_file = "data/train-images.idx3-ubyte";

char * const label_file = "data/smallnorb-5x46789x9x18x6x2x96x96-
training-cat.mat";
char * const data_file = "data/smallnorb-5x46789x9x18x6x2x96x96-
training-dat.mat";

char * const test_label_file = "";
char * const test_data_file = "";

//CUDA Functions
__global__ void vToH0(float* v0, float* h0, float* aj, float* b,
float* w);
__global__ void H0ToH1(float* h0, float* h1, float* aj, float* b,
float* w);
__global__ void H1ToTop(float* h1, float* top, float* b, float* w);
__global__ void calcErrorTop(float* top, float* label, float*
error);
```

```
__global__ void calcErrorH1(float* top_error, float* aj, float* w,
float* error);
__global__ void calcErrorH0(float* h1_error, float* aj, float* w,
float* error);
__global__ void updateW0(float* v0, float* error, float* w, float*
vel_w, float momentum, float l_rate);
__global__ void updateW1(float* h0, float* error, float* w, float*
vel_w, float momentum, float l_rate);
__global__ void updateW2(float* h1, float* error, float* w, float*
vel_w, float momentum, float l_rate);
__global__ void updateBiasH0(float* error, float* bias, float
l_rate);
__global__ void updateBiasH1(float* error, float* bias, float
l_rate);
__global__ void updateBiasTop(float* error, float* bias, float
l_rate);

//Globals
Nn *my_nn;
Trainer *my_trainer;

int test_iter;
int example;

//float total_time;

int main(int argc, char** argv)
{
      cudaSetDevice(1);

      example = 0;
      test_iter = 0;
      //total_time= 0;

      Layer* visible = new Layer(VSIZE_X, VSIZE_Y, BATCH, SAMPLES,
false);
      Connection* v_to_h0 = new Connection(VSIZE, H0SIZE);
      Layer* h0 = new Layer(H0SIZE_X, H0SIZE_Y, BATCH, SAMPLES,
true);
      Connection* h0_to_h1 = new Connection(H0SIZE, H1SIZE);
      Layer* h1 = new Layer(H1SIZE_X, H1SIZE_Y, BATCH, SAMPLES,
true);
      Connection* h1_to_top = new Connection(H1SIZE, CLASS);
      Layer* top = new Layer(CLASS, 1, BATCH, SAMPLES, false);

      my_nn = new Nn(visible, h0, h1, top, v_to_h0, h0_to_h1,
h1_to_top);
      my_trainer = new Trainer(BATCH, SAMPLES, EPOCH, 0.0001, CLASS,
0.5, 0.00005);

      if(LOAD==0)
      {
```

```
            //Init top connection
            v_to_h0->initParams();
            h0_to_h1->initParams();
            h1_to_top->initParams();
            //Set bias towards top to
            my_nn->setTopProb();
      }
      else if(LOAD==1)
      {
            //Load connection information
            if(   load_from_rbm(param_file_h0_h1, h0_to_h1) < 0)
            {
                  printf("Failed to load params from
%s\n",param_file_h0_h1);
                  return -1;
            }
            if(   load_from_rbm(param_file_v_h0, v_to_h0) < 0)
            {
                  printf("Failed to load params from
%s\n",param_file_v_h0);
                  return -1;
            }
            //Init top connection
            h1_to_top->initParams();
            //Set bias towards top to
            my_nn->setTopProb();
      }
      else if(LOAD==2)
      {
            my_nn->loadComplete(param_file_complete);
      }



      //if(my_trainer->loadTrainingData(data_file) < 0)
      if(my_trainer->loadTrainingDataMAT(data_file) < 0)
      {
            printf("An error occurred loading the training data.
Exiting...\n");
            return -1;
      }
      //if(my_trainer->loadTrainingLabels(label_file) < 0)
      if(my_trainer->loadTrainingLabelsMAT(label_file) < 0)
      {
            printf("An error occurred loading the training labels.
Exiting...\n");
            return -1;
      }

      //Set up GLUT Parameters
      glutInit(&argc, argv);
      glutInitDisplayMode(GLUT_SINGLE);
```

```
	glutInitWindowSize(500,500);
	glutInitWindowPosition(250,250);
	glutCreateWindow("Neural Network");

	if(MODE==0)
	{
		printf("Correct Label for Item %d: %d\n", TRAIN_EXAMPLE,
my_trainer->answer(TRAIN_EXAMPLE));
		glutDisplayFunc(train);
		//glutDisplayFunc(test);
		glutMainLoop();
	}
	else if(MODE==1)
		classifyValid();
		//classifyTrain();



	return 0;


}

/*=================================================================
=
 *               DISPLAY FUNCTIONS

==================================================================*
/
void test()
{
	glClearColor(1.0,1.0,1.0,1.0);
	glClear(GL_COLOR_BUFFER_BIT);
	glOrtho(-1.0,1.0,-1.0,1.0,-1.0,1.0);

	//my_trainer->setV(example,0);


	if(test_iter ==0)
	{
		my_trainer->randBatchV();
		dim3 blockDim(BLOCKS_LAYER,SAMPLES,BATCH);
		dim3 threadDim0(my_nn->getH0Size()/BLOCKS_LAYER);
		vToH0<<<blockDim,threadDim0>>>(my_trainer-
>d_mini_batch_data, my_nn->getH0(), my_nn->getH0In(), my_nn-
>getDevB_0(), my_nn->getDevW_0());
		dim3 threadDim1(my_nn->getH1Size()/BLOCKS_LAYER);
		H0ToH1<<<blockDim,threadDim1>>>(my_nn->getH0(), my_nn-
>getH1(), my_nn->getH1In(), my_nn->getDevB_1(), my_nn->getDevW_1());
		H1ToTop<<<BATCH,my_nn->getTopSize()>>>(my_nn->getH1(),
my_nn->getTop(), my_nn->getDevB_2(), my_nn->getDevW_2());
		my_trainer->showCurrent(0);
```

```
	}
	else if(test_iter ==1)
	{
		printf("Image Category: %d\n",my_trainer->ansCurrent(0));
		float tmp[5];
		cudaMemcpy(tmp,my_trainer-
>d_mini_batch_labels,5*sizeof(float),cudaMemcpyDeviceToHost);
		printf("[%f][%f][%f][%f][%f]\n", tmp[0], tmp[1], tmp[2],
tmp[3], tmp[4]);
		my_nn->showTop(0);
	}

	test_iter++;
	if(test_iter>=2)
	{
		test_iter=0;
		example++;
	}



	glFlush();
	sleep(2);
	glutPostRedisplay();
}

void train()
{
	glClearColor(1.0,1.0,1.0,1.0);
	glClear(GL_COLOR_BUFFER_BIT);
	glOrtho(-1.0,1.0,-1.0,1.0,-1.0,1.0);

	//cudaEvent_t start, stop;
	//float time;
	//cudaEventCreate(&start);
	//cudaEventCreate(&stop);

	//cudaEventRecord(start, 0);
	train_mini();
	//cudaEventRecord(stop, 0);
	//cudaEventSynchronize(stop);
	//cudaEventElapsedTime(&time, start, stop);

	//printf("Time for mini_batch (20): %f ms\n", time);
	//total_time+=time;

	//Show training examples
	my_trainer->setV(TRAIN_EXAMPLE,0);

	dim3 blockDim(BLOCKS_LAYER,SAMPLES,BATCH);
	dim3 threadDim0(my_nn->getH0Size()/BLOCKS_LAYER);
```

```
	vToH0<<<blockDim,threadDim0>>>(my_trainer->d_mini_batch_data,
my_nn->getH0(), my_nn->getH0In(), my_nn->getDevB_0(), my_nn-
>getDevW_0());


	dim3 threadDim1(my_nn->getH1Size()/BLOCKS_LAYER);
	H0ToH1<<<blockDim,threadDim1>>>(my_nn->getH0(), my_nn-
>getH1(), my_nn->getH1In(), my_nn->getDevB_1(), my_nn->getDevW_1());

	//my_nn->showH1(0);
	//printf("Top Size = %d", my_nn->getTopSize());
	H1ToTop<<<BATCH,my_nn->getTopSize()>>>(my_nn->getH1(), my_nn-
>getTop(), my_nn->getDevB_2(), my_nn->getDevW_2());
	my_nn->showTop(0);

	glFlush();

	//Update training status
	my_trainer->incN();
	if(my_trainer->epochComplete())
	{
		printf("Epoch %d Complete!\n", my_trainer->getEpoch());
		validationError();
		my_nn->saveComplete(param_file_complete);
	}

	if(!my_trainer->trainComplete())
		glutPostRedisplay();
	else
	{
		printf("Training run complete!\n");
		//printf("Average Time for mini_batch(20): %f ms\n",
total_time / ( (float)(my_trainer->getTrainSize() - my_trainer-
>getValidSize()) / (float)BATCH) );
	}

}

void classifyTrain()
{
	int batch_size=0;
	int num_seen = 0;
	int num_wrong =0;

	while(num_seen < my_trainer->getTrainSize()-my_trainer-
>getValidSize())
	//while(num_seen < my_trainer->getValidSize())
	{
		batch_size = my_trainer->nextBatchTrain();
		//batch_size = my_trainer->nextBatchValid();
		num_seen += batch_size;
		dim3 blockDim(BLOCKS_LAYER,SAMPLES,batch_size);
```

```
            dim3 threadDim0(my_nn->getH0Size()/BLOCKS_LAYER);

            vToH0<<<blockDim,threadDim0>>>(my_trainer-
>d_mini_batch_data, my_nn->getH0(), my_nn->getH0In(), my_nn-
>getDevB_0(), my_nn->getDevW_0());
            dim3 threadDim1(my_nn->getH1Size()/BLOCKS_LAYER);

            H0ToH1<<<blockDim,threadDim1>>>(my_nn->getH0(), my_nn-
>getH1(), my_nn->getH1In(), my_nn->getDevB_1(), my_nn->getDevW_1());

            H1ToTop<<<batch_size,my_nn->getTopSize()>>>(my_nn-
>getH1(), my_nn->getTop(), my_nn->getDevB_2(), my_nn->getDevW_2());

            num_wrong += my_trainer->batchClassification(my_nn-
>getTop(),batch_size);
      }
      printf("Missed %d out of %d\n",num_wrong, num_seen);
      printf("Misclassification rate: %f", (float)num_wrong /
(float)num_seen);
}

void classifyValid()
{
      int batch_size=0;
      int num_seen = 0;
      int num_wrong =0;

      while(num_seen < my_trainer->getValidSize())
      {
            batch_size = my_trainer->nextBatchValid();
            num_seen += batch_size;
            dim3 blockDim(BLOCKS_LAYER,SAMPLES,batch_size);
            dim3 threadDim0(my_nn->getH0Size()/BLOCKS_LAYER);

            vToH0<<<blockDim,threadDim0>>>(my_trainer-
>d_mini_batch_data, my_nn->getH0(), my_nn->getH0In(), my_nn-
>getDevB_0(), my_nn->getDevW_0());
            dim3 threadDim1(my_nn->getH1Size()/BLOCKS_LAYER);

            H0ToH1<<<blockDim,threadDim1>>>(my_nn->getH0(), my_nn-
>getH1(), my_nn->getH1In(), my_nn->getDevB_1(), my_nn->getDevW_1());

            H1ToTop<<<batch_size,my_nn->getTopSize()>>>(my_nn-
>getH1(), my_nn->getTop(), my_nn->getDevB_2(), my_nn->getDevW_2());

            num_wrong += my_trainer->batchClassification(my_nn-
>getTop(),batch_size);
      }
      printf("Missed %d out of %d\n",num_wrong, num_seen);
      printf("Misclassification rate: %f", (float)num_wrong /
(float)num_seen);
}
```

```
void validationError()
{

      printf("Calculating Avg Error on Validation Set...\n");

      ofstream o_file;
      o_file.open("experiments/last_run.err", ios::app);
      o_file.seekp( ios::end );
      //o_file << "Epoch: " << my_trainer->getEpoch() << "\t";


      int batch_size=0;
      float num_seen = 0;
      float total_error =0;

      while(num_seen < my_trainer->getValidSize())
      {
            batch_size = my_trainer->nextBatchValid();
            num_seen += batch_size;
            dim3 blockDim(BLOCKS_LAYER,SAMPLES,batch_size);
            dim3 threadDim0(my_nn->getH0Size()/BLOCKS_LAYER);

            vToH0<<<blockDim,threadDim0>>>(my_trainer-
>d_mini_batch_data, my_nn->getH0(), my_nn->getH0In(), my_nn-
>getDevB_0(), my_nn->getDevW_0());
            dim3 threadDim1(my_nn->getH1Size()/BLOCKS_LAYER);

            H0ToH1<<<blockDim,threadDim1>>>(my_nn->getH0(), my_nn-
>getH1(), my_nn->getH1In(), my_nn->getDevB_1(), my_nn->getDevW_1());

            H1ToTop<<<BATCH,my_nn->getTopSize()>>>(my_nn->getH1(),
my_nn->getTop(), my_nn->getDevB_2(), my_nn->getDevW_2());

            total_error += my_trainer->batchError(my_nn-
>getTop(),batch_size);
      }

      //o_file << "Avg Error(valid): " << total_error / num_seen <<
"\n";

      o_file << total_error / num_seen << ",";

      o_file.close();


      printf("Average Error on Validation Set: %f\n",total_error /
num_seen);
}

/*================================================================
=
```

```
 *               HELPER FUNCTIONS

================================================================*
/

void train_mini()
{
      my_trainer->randBatchV();
      //my_trainer->showCurrent(0);
      //printf("\nLabel: %d\t",my_trainer->ansCurrent(0));

      dim3 blockDim(BLOCKS_LAYER,SAMPLES,BATCH);

      dim3 threadDim0(my_nn->getH0Size()/BLOCKS_LAYER);
      vToH0<<<blockDim,threadDim0>>>(my_trainer->d_mini_batch_data,
my_nn->getH0(), my_nn->getH0In(), my_nn->getDevB_0(), my_nn-
>getDevW_0());

      dim3 threadDim1(my_nn->getH1Size()/BLOCKS_LAYER);
      H0ToH1<<<blockDim,threadDim1>>>(my_nn->getH0(), my_nn-
>getH1(), my_nn->getH1In(), my_nn->getDevB_1(), my_nn->getDevW_1());

      H1ToTop<<<BATCH,my_nn->getTopSize()>>>(my_nn->getH1(), my_nn-
>getTop(), my_nn->getDevB_2(), my_nn->getDevW_2());
      //my_nn->printTop(0);


      calcErrorTop<<<BATCH,my_nn->getTopSize()>>>(my_nn->getTop(),
my_trainer->d_mini_batch_labels, my_nn->getTopError());
      calcErrorH1<<<blockDim,threadDim1>>>(my_nn->getTopError(),
my_nn->getH1In(), my_nn->getDevW_2(), my_nn->getH1Error());
      calcErrorH0<<<blockDim,threadDim0>>>(my_nn->getH1Error(),
my_nn->getH0In(), my_nn->getDevW_1(), my_nn->getH0Error());

      updateW0<<<BLOCKS_LAYER,threadDim0>>>(my_trainer-
>d_mini_batch_data, my_nn->getH0Error(), my_nn->getDevW_0(), my_nn-
>getDevVw_0(), my_trainer->getMomentum(), my_trainer-
>getLearnRate());
      updateW1<<<BLOCKS_LAYER,threadDim1>>>(my_nn->getH0(), my_nn-
>getH1Error(), my_nn->getDevW_1(), my_nn->getDevVw_1(), my_trainer-
>getMomentum(), my_trainer->getLearnRate());
      updateW2<<<1,my_nn->getTopSize()>>>(my_nn->getH1(), my_nn-
>getTopError(), my_nn->getDevW_2(), my_nn->getDevVw_2(), my_trainer-
>getMomentum(), my_trainer->getLearnRate());

      updateBiasH0<<<BLOCKS_LAYER,threadDim0>>>(my_nn->getH0Error(),
my_nn->getDevB_0(), my_trainer->getLearnRate());
      updateBiasH1<<<BLOCKS_LAYER,threadDim1>>>(my_nn->getH1Error(),
my_nn->getDevB_1(), my_trainer->getLearnRate());
      updateBiasTop<<<1,my_nn->getTopSize()>>>(my_nn->getTopError(),
my_nn->getDevB_2(), my_trainer->getLearnRate());
```

```
}

int   load_from_rbm(char* rbm_file, Connection* con)
{
      printf("Loading RBM from %s...",rbm_file);
      ifstream i_file;
      i_file.open(rbm_file, ios::binary);
      if(i_file.is_open())
      {
            int loc = 0;

            loc = con->load(&i_file, loc);

            i_file.close();

            printf("Completed!\n");
            return 0;
      }
      else
      {
            printf("Failed to open file\n");
            return -1;
      }
}

/*================================================================
=
 *               CUDA FUNCTIONS

================================================================*
/

__global__ void vToH0(float* v0, float* h0, float* aj, float* b,
float* w)
{
      int h_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
      int b_off = blockIdx.z;
      int t_off = ( b_off * H0SIZE ) + h_idx;

      aj[t_off] = b[h_idx];
      //printf("sum = %f \n",b[h_idx]);
      for(int i=0;i<VSIZE;i++)
      {
            aj[t_off] += v0[b_off*VSIZE + i] * w[ i*H0SIZE + h_idx];
      }
      //printf("sum = %f \n",b[h_idx]);
      h0[t_off] = 1 / (1 + __expf(-1 * aj[t_off]));
}

__global__ void H0ToH1(float* h0, float* h1, float* aj, float* b,
float* w)
{
```

```
	int h_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
	int b_off = blockIdx.z;
	int t_off = (b_off * H1SIZE ) + h_idx;

	aj[t_off] = b[h_idx];
	//printf("sum = %f \n",b[h_idx]);
	for(int i=0;i<H0SIZE;i++)
	{
		aj[t_off] += h0[b_off*H0SIZE + i] * w[ i*H1SIZE + h_idx];
	}
	//printf("sum = %f \n",b[h_idx]);
	h1[t_off] = 1 / (1 + __expf(-1 * aj[t_off]));
}

__global__ void H1ToTop(float* h1, float* top, float* b, float* w)
{
	int h_idx = threadIdx.x;
	int b_off = blockIdx.x;
	int t_off = b_off*CLASS + h_idx;

	float sum = b[h_idx];
	//printf("sum = %f \n",b[h_idx]);
	for(int i=0;i<H1SIZE;i++)
	{
		sum += h1[b_off*H1SIZE + i] * w[ i*CLASS + h_idx];
	}
	//printf("sum = %f \n",b[h_idx]);
	//top[t_off] = 1 / (1 + __expf(-1 * sum));
	top[t_off] = __expf(sum);

	__syncthreads();

	sum = 0;
	for(int k=0;k<CLASS;k++)
	{
		sum+=top[b_off*CLASS+k];
	}

	__syncthreads();

	top[t_off] = top[t_off] / sum;
}

__global__ void calcErrorTop(float* top, float* label, float* error)
{
	int h_idx = threadIdx.x;
	int b_off = blockIdx.x;
	int t_off = b_off*CLASS + h_idx;

	error[t_off] =  top[t_off] - label[t_off];
	//printf("dk[%d]=%f\t",t_off,error[t_off]);
}
```

```
__global__ void calcErrorH1(float* top_error, float* aj, float* w,
float* error)
{
      int h_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
      int g_off = 0;
      int b_off = blockIdx.z;
      int t_off = ( (b_off * gridDim.y + g_off) * H1SIZE ) + h_idx;

      float sum=0;

      for(int i=0;i<CLASS;i++)
      {
            sum += top_error[b_off*CLASS + i] * w[ i*CLASS + h_idx];
      }

      error[t_off] = (  __expf(aj[t_off]) / pow((1 +
__expf(aj[t_off])),2)  ) * sum;
      //if(t_off < 50)
            //printf("dj[%d]=%f\n",t_off,error[t_off]);
}

__global__ void calcErrorH0(float* h1_error, float* aj, float* w,
float* error)
{
      int h_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
      int g_off = 0;
      int b_off = blockIdx.z;
      int t_off = ( (b_off * gridDim.y + g_off) * H0SIZE ) + h_idx;

      float sum=0;

      for(int i=0;i<H1SIZE;i++)
      {
            sum += ((h1_error[b_off*H1SIZE + i] * w[ i*H1SIZE +
h_idx]));
      }

      //error[t_off] = (  __expf(aj[t_off]/2) / pow((1 +
__expf(aj[t_off]/2)),2)  );
      error[t_off] = (  __expf(aj[t_off]/2) / pow((1 +
__expf(aj[t_off]/2)),2)  ) * sum;
      //error[t_off] *= 100;
      //if(t_off < 50)

      //printf("aj=%f\tdi[%d]=%f\n",aj[t_off]/2,t_off,error[t_off]);
}

__global__ void updateW0(float* v0, float* error, float* w, float*
vel_w, float momentum, float l_rate)
{
      int h_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
```

```
	float deltaW = 0;
	//printf("sum = %f \n",b[h_idx]);

	for(int i=0;i<VSIZE;i++)
	{
		deltaW = 0;
		for(int b=0;b<BATCH;b++)
		{
			deltaW += error[h_idx + b*H0SIZE]*v0[i + VSIZE*b];
		}
		//printf("w[%d]=%f + %f\n",i*H0SIZE + h_idx, w[i*H0SIZE +
h_idx], l_rate*deltaW);

		//VELOCITY
		vel_w[i*H0SIZE + h_idx] = momentum * vel_w[i*H0SIZE +
h_idx] + l_rate*(deltaW/BATCH);
		w[i*H0SIZE + h_idx] -= vel_w[i*H0SIZE + h_idx];

		//w[i*H0SIZE + h_idx] -= l_rate*(deltaW/BATCH);
	}
}

__global__ void updateW1(float* h0, float* error, float* w, float*
vel_w, float momentum, float l_rate)
{
	int h_idx = (blockIdx.x * blockDim.x) + threadIdx.x;

	float deltaW = 0;
	//printf("sum = %f \n",b[h_idx]);

	for(int i=0;i<H0SIZE;i++)
	{
		deltaW = 0;
		for(int b=0;b<BATCH;b++)
		{
			deltaW += error[h_idx + b*H1SIZE]*h0[i + H0SIZE*b];
		}
		//printf("w[%d]=%f + %f\n",i*H1SIZE + h_idx, w[i*H1SIZE +
h_idx], l_rate*deltaW);
		vel_w[i*H1SIZE + h_idx] = momentum * vel_w[i*H1SIZE +
h_idx] + l_rate*(deltaW/BATCH);
		w[i*H1SIZE + h_idx] -= vel_w[i*H1SIZE + h_idx];

		//w[i*H1SIZE + h_idx] -=  l_rate*(deltaW/BATCH);
	}
}

__global__ void updateW2(float* h1, float* error, float* w, float*
vel_w, float momentum, float l_rate)
{
	int h_idx = threadIdx.x;
```

```
	float deltaW = 0;
	//printf("sum = %f \n",b[h_idx]);

	for(int i=0;i<H1SIZE;i++)
	{
		deltaW = 0;
		for(int b=0;b<BATCH;b++)
		{
			deltaW += error[h_idx + b*CLASS]*h1[i + H1SIZE*b];
		}
		//printf("w[%d->%d]=%f - %f\n",i, h_idx, w[i*CLASS +
h_idx], l_rate*deltaW);
		vel_w[i*CLASS + h_idx] = momentum * vel_w[i*CLASS +
h_idx] + l_rate*(deltaW/BATCH);
		w[i*CLASS + h_idx] -= vel_w[i*CLASS + h_idx];

		//w[i*CLASS + h_idx] -= l_rate*(deltaW/BATCH);
	}
}

__global__ void updateBiasH0(float* error, float* bias, float
l_rate)
{
	int h_idx = (blockIdx.x * blockDim.x) + threadIdx.x;

	float deltaB = 0;
	//printf("sum = %f \n",b[h_idx]);

	for(int b=0;b<BATCH;b++)
	{
		deltaB += error[h_idx + b*H0SIZE];
	}
	//printf("b[%d]=%f + %f\n",h_idx, bias[h_idx], l_rate*deltaB);
	bias[h_idx] -= l_rate*(deltaB/BATCH);
}

__global__ void updateBiasH1(float* error, float* bias, float
l_rate)
{
	int h_idx = (blockIdx.x * blockDim.x) + threadIdx.x;

	float deltaB = 0;
	//printf("sum = %f \n",b[h_idx]);

	for(int b=0;b<BATCH;b++)
	{
		deltaB += error[h_idx + b*H1SIZE];
	}
	//printf("w[%d]=%f + %f\n",i*H0SIZE + h_idx, w[i*H0SIZE +
h_idx], l_rate*deltaW);
	bias[h_idx] -= l_rate*(deltaB/BATCH);
```

```
}

__global__ void updateBiasTop(float* error, float* bias, float
l_rate)
{
      int h_idx = (blockIdx.x * blockDim.x) + threadIdx.x;

      float deltaB = 0;
      //printf("sum = %f \n",b[h_idx]);

      for(int b=0;b<BATCH;b++)
      {
            deltaB += error[h_idx + b*CLASS];
      }
      //printf("b[->%d]=%f - %f\n", h_idx, bias[h_idx],
l_rate*deltaB);
      bias[h_idx] -= l_rate*(deltaB/BATCH);
}
```