# GPU for Deep Learning Algorithm

*CSC466 GPU class final project report*

## Introduction

There are many successful applications to take advantages of massive parallelization on GPU for deep learning algorithm. In this project, I implemented a basic deep learning algorithm, i.e. Autoencoder. Core parts of this project are based on CUBLAS and CUDA kernels.

I will first briefly introduce sparse autoencoder to make this report coherent, and to inspire the idea. Second, several technical highlights are summarized. In the end, experiment results will be presented.

## Sparse Autoencoder

A common machine learning pipeline is 1) representation learning, 2) feature extraction, and 3) classification. In this project, I am focusing on representation learning using sparse autoencoder.

Consider such an artificial neural network (ANN),

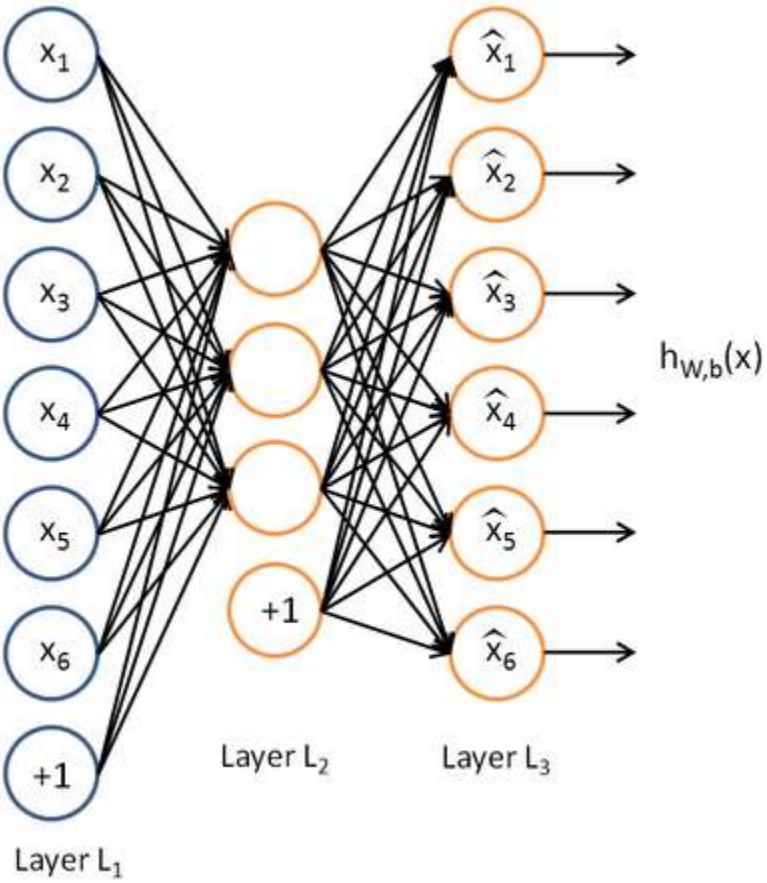


Figure 1 Autoencoder example

$\mathbf{X_1}$ to $\mathbf{X_6}$ in the Layer $\mathbf{L_1}$ are raw pixels. By feeding the raw pixels into neural network, we can get response in Layer $\mathbf{L_2}$ and $\mathbf{L_3}$. In the autoencoder framework, the $\mathbf{L_1}$ to $\mathbf{L_2}$ mapping is called *encoder* and $\mathbf{L_2}$ to $\mathbf{L_3}$ mapping is called *decoder*.

In information theory field, the goal of a compression encoder is to minimize information redundancy, which means using short bit length to keep as much as information as possible. So what does it mean by keep as much information as possible? One possibility is to ask to what extent the codes can reconstruct the original signal. Therefore, a decoder is attached next to the encoder in autoencoder to measure the quality of the encoder (**Figure 1**).

As illustrated in **Figure 1**, there are many parameters (the connection between nodes) to be optimized for both encoder and decoder. So what is the objective of this optimization procedure? In the end, we are looking for the best encoder, which means the best set of weights between $\mathbf{L_1}$ and $\mathbf{L_2}$, let's call it $\mathbf{W_1}$. Given one set of $\mathbf{W_1}$, we need to find the best decoder to reflect the information kept by the encoder, which means to optimize weights between $\mathbf{L_2}$ and $\mathbf{L_3}$, let's call it $\mathbf{W_2}$. This procedure is equivalent to jointly optimize $\mathbf{W_1}$ and $\mathbf{W_2}$ to minimize the following cost function, in which $\mathbf{y}$ is the original signal ($\mathbf{X_1}$ to $\mathbf{X_6}$), and $\mathbf{h_{W,b}(x)}$ is the output of the network,

$$J(W,b;x,y) = \frac{1}{2}\left\|h_{W,b}(x) - y\right\|^2.$$

**Figure 2 Autoencoder Objective**

In machine learning field, the representation learning has some similarities with 'optimal encoder' in information theory field. However, there are many fundamental differences, which make the 'sparse' stand out. In representation learning, we are not very interested in compressing signal, so the number of nodes in $\mathbf{L_2}$ can be much larger in $\mathbf{L_1}$, in the experiments, the number of nodes in $\mathbf{L_2}$ is 400 and in $\mathbf{L_1}$, it is 192. It means we are using longer bit length to represent original signal, so trivial solution can be easily found for objective in **Figure 2**, which are not useful. Therefore, an additional constraint, called sparsity, is enforced in **Figure 2**, so that at least useful information can be found in the optimization. Specifically, we prefer the encoder that outputs as many zeros in $\mathbf{L_2}$ as possible ('many zeros' is another word for sparse). The objective is changed into the following form,

$$J_{\text{sparse}}(W,b) = J(W,b) + \beta \sum_{j=1}^{s_2} \text{KL}(\rho||\hat{\rho}_j),$$

**Figure 3 Autoencoder objective with sparsity**

, in which, $\text{KL}(\rho||\hat{\rho}_j) = \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho)\log\frac{1-\rho}{1-\hat{\rho}_j}$

, in which $\hat{\rho}_j = \frac{1}{m}\sum_{i=1}^{m}\left[a_j^{(2)}(x^{(i)})\right]$, called average activation for node **j**, and $\rho = 0.05$, called target activation.

In theory, there are many other key benefits to enforce this kind of sparsity, (other than output non-trivial solution). Briefly speaking, the sparsity enforced optimization is useful to find quantities that are invariant to irrelevant changes. For example, in face recognition, we are looking for invariants that are robust to changes of pose, lighting condition, and occlusion.

In practice, gradients of the objective in **Figure 3** can be calculated by error back propagation. Along than gradients calculation, an optimization procedure such as L-BFGS can be used to minimize the objective.

## Implementation

The ultimate goal of this project is to minimize the objective in Figure 3, which is composed of following components,

1. Feed forward, i.e. calculate the output of the network given a set of $\mathbf{W_1}$ and $\mathbf{W_2}$.
2. Cost calculation, i.e. calculate the objective in **Figure 3**.
3. Back propagation, i.e. calculate the gradients of the objective in **Figure 3**.
4. Optimize $\mathbf{W_1}$ and $\mathbf{W_2}$ given cost and gradients using L-BFGS.
5. Go back to step 1 until convergence.

In this project, we are using standard L-BFGS routines for step 4, so only step 1 to step 3 is implemented on GPU. Following are the code references,

- Feed forward in *twoLayerFF.cu*,
- Cost calculation in *twoLayerCost.cu,*
- Back propagation in *twoLayerBP.cu*.

These are core parts of this project. Design strategy and key peripheral helper functions are discussed as follows.

### *Testing*

It becomes *de facto standard* to compare the results of GPU against CPU to 1) confirm that the GPU implementation is valid; 2) the benefit of GPU is significant. Because it's so important for any GPU based algorithm, I discuss my testing strategy before anything else. The algorithm was firstly implemented in MATLAB, so the testing is straightforward. For each component, compare output of GPU and MATLAB, and they should close to each other. Following is the procedure to test a given component,

1. Generate testing input data use MATLAB and write to hard disk.
2. Read the testing data from hard disk.

3. Perform the calculation using GPU.
4. Write the output into hard disk.
5. Compare output from GPU with that of MATLAB.

The MATLAB implementation of the algorithm can be found under matlab/*.The code related to testing is listed as follows,

| Components | Feed forward | Cost Calculation | Back Propagation |
|---|---|---|---|
| **CUDA Implementation (cuda)** | *twoLayerFF.cu* | *twoLayerCost.cu* | *twoLayerBP.cu* |
| **MATLAB Implementation (matlab)** | *twoLayerFF.m* | *twoLayerCost.m* | *twoLayerBP.m* |
| **CUDA testing code (cudaUnits)** | *testTwoLayerFF.cu* | *testTwoLayerCost.cu* | *testTwoLayerBP.cu* |
| **Generate testing data (cudaUnits)** | *gen_2ff_test.m* | *gen_2cost_test.m* | *gen_2bp_test.m* |
| **Confirm results (cudaUnits)** | *confirm_2ff_test.m* | *confirm_2cost _test.m* | *confirm_2bp _test.m* |

### *Matrix IO*

As indicated in the testing section, I need to transfer data back and forth from MATLAB using hard disk, so matrix IO is another big headache in the project. I used an open source package libmatio [1] to deal with the MATLAB file format. The code related to Matrix IO is listed as follows,

- matrix.cpp, matrix.hpp, wrapper for libmatio.
- common.cuh/ IO_MATRIX_WRAPPER to make read and write matrix super easy.

### *CUDA API wrapper*

The CUDA API is notoriously ugly to use. To make it the code look clean, a set of macro is implemented. (all in common.cuh)

- CUDA_*: GPU memory allocation and transfer data between CPU and GPU.
- RUN_KERNEL_*: run kernel with different number of parameters.
- gpu_blas_*: CUBLAS wrapper, including dot production, 2-norm, scaling, copy, multiplication

### *Optimization wrapper*

I used the solver from [2] for optimization, and what I need to implement is gradients and cost function. The code related to optimization is listed as follows,

- *OWLQN.cpp, OWLQN.h, TerminationCriterion.cpp* and *TerminationCriterion.h*, provided by [2] for optimization;

- *sparseAutoencoderLinearCost.cpp* and *sparseAutoencoderLinearCost.h*, wrapper around all CUDA operations and adapting it to OWLQN solver. The CUDA wrapper is responsible for allocating device memory and fire up device.
- *twoLayerOpt.cpp* , generate executable to find optimal $W_1$ and $W_2$.

### *Documentation for future dev*

This project can be a good starting point for porting MATLAB code, so I wrote detailed documentation about how to setup building and testing, which are included in README and comments among the code.

## Experiment Results

Testing Platform:
CPU related: MATLAB 2012b, Intel i5-3210M, 2C/4T 2.5GHz. (Notice that MATLAB will use 2 cores for computation)
GPU related: Device: Nvidia TESLA C2075, Host: Intel i7 920 4C/8T.

The runtime comparison is based on optimize the objective to a certain convergent criterion. Since different solvers are used with MATLAB and CUDA, the numbers of iterations are different. Note that the 'Total Acceleration' includes the gain from the more efficient cpp solver.

| | Number of iterations | Total Runtime(s) | Runtime per Iteration (s) | Total acceleration | Acceleration by GPU |
|---|---|---|---|---|---|
| **MATLAB** | 538 | 2288.18 | 4.25 | N/A | N/A |
| **CUDA with double precision** | 240 | 111.96 | 0.47 | 21x | 10x |
| **CUDA with float precision** | 241 | 54.66 | 0.23 | 43x | 20x |
| **CUDA with float precision and fast math** | 229 | 51.59 | 0.23 | 45x | 20x |

Using the trained network to extract features then train soft-max classifiers [4], the held-out testing accuracy is 80% ± 0.1% for both MATLAB and CUDA results, which means the autoencoder training is successfully. Note that the feature extraction and classifier training is not implemented on CUDA, but on MATLAB.

In addition to the runtime comparison for the whole algorithm, comparisons are also made for independent components. To compare independent components, the functions are called repeatedly for a large number of times. To illustrate the overhead of data transfer, the runtime of function call with fetching data and without fetching data are also compared. Note that in the optimization iterations, only results of Back propagation components need to be fetched from GPU to CPU. The results are listed as follows, (the compile configuration on device is single precision and fast-math):

| | MATLAB (s) | Without fetching results (s) | With fetching results (s) | Number of iterations |
|---|---|---|---|---|
| Feed Forward | 936.59 | 30.89 (30.32x) | 52.79 (17.74x) | 500 |
| Cost Calculation | 99.13 | 4.66 (21.27x) | 12.15 (8.15x) | 500 |
| Back Propagation | 975.57 | 65.33 (14.93x) | 65.57 (14.88x) | 500 |

## Conclusion

The experiment result shows that the GPU implementation accelerates the autoencoder algorithm by around **20x** compare to a 2-core implementation. Further development will focus more on profiling to optimize low efficient code. Also, as indicated from the results, the data fetching is very expensive, so a native GPU optimization solver will help a lot, so that no data fetching will be needed during the iterations.

## Reference

[1] Libmatio, http://sourceforge.net/projects/matio/
[2] OWLQN: http://research.microsoft.com/en-us/downloads/b1eb1016-1738-4bd5-83a9-370c9d498a03/
[3] L-BFGS: http://en.wikipedia.org/wiki/Broyden%E2%80%93Fletcher%E2%80%93Goldfarb%E2%80%93Shanno_algorithm
[4] UFLDL: http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial