# Deep Learning Benchmarks

Mumtaz Vauhkonen, Quaizar Vohra, Saurabh Madaan
in collaboration with Adam Coates

**Abstract:** *This project aims at creating a benchmark for Deep Learning (DL) algorithms by identifying a set of basic operations which together account for most of the CPU usage in these algorithms. These operations would then be standardized into an API, similar to BLAS or LINPACK, decoupling their implementation from the DL algorithms themselves. The goal here is to facilitate optimization of DL algorithms for supercomputing and HPC experts by boiling them down to these simple operations. To this end, we have implemented 3 algorithms covering different aspects of DL: 1) Sparse Auto-encoder (AE) for mapping input to useful features 2) Convolutional Neural Nets (CNN) for supervised classification, and 3) Fast Iterative Shrinkage Threshold Algorithm (FISTA), an optimization algorithm for Sparse Coding. These algorithms were implemented and tested in Python and Matlab. Reference implementation of the API was created using Numpy and Theano (with GPU support). Our results show that API implementation using Theano GPU performs 3 to 15 times faster (depending on the algorithm) than the one based on Numpy. This validates the usefulness of our API with respect to optimization.*

## Introduction

There have been many efforts in trying to improve the efficiency of deep learning algorithms. For instance, Coates et. al. have shown that deep learning implementations based on high-performance computing (HPC) can train large networks using fewer machines [1]. However, performance benchmarks for deep learning algorithms using HPC are lacking.

Our project aims to develop benchmarks for deep learning algorithms in the context of high-performance computing (HPC). At the start, this benchmark will cover 3 algorithms from different areas of DL. These algorithms (CNN, AE and FISTA) were selected by our collaborator and are introduced in corresponding sections below. To simplify the adoption of these benchmarks in the HPC/Supercomputing world, we have identified a small set of optimizable operations by profiling our code in Matlab and Python and abstracted them into an API. This decouples their implementation and fine-tuning from the implementation of the DL algorithms themselves. The operations are simple matrix manipulations and are easy to understand as opposed to the DL algorithms themselves. The supercomputing/HPC experts can focus on optimizing this API as opposed to expending their energy on understanding the Deep-learning algorithms. This is depicted in figure 1. Subsequently, our project collaborator, Adam Coates, will use this implementation to create benchmarks on a supercomputer. Readers who are familiar with these algorithms may skip over to the Section II.

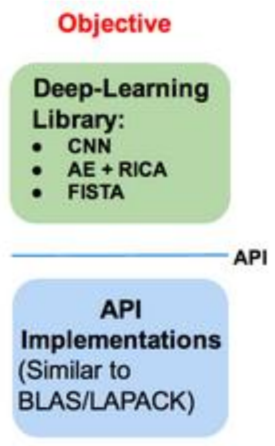


Figure 1: Project Objective

## Section I: Algorithms Implemented

**Sparse Auto-encoders (AE):** Sparse auto-encoders are used in mapping input data to a set of useful features, e.g. edges at different orientations in an image which are useful in image recognition tasks. The set of useful features could be very large, but only a small subset of these features are typically present in a real life data sample, e.g. an image patch. Hence a sparse combination of these features is required to represent the input data. To capture this sparsity property, AEs are modeled as 3 layer neural networks with the middle layer generating the useful features. To model sparsity, the optimization problem consists of a *sparsity penalty.* For more details, please refer to [2]. The input of the middle layer can be fed into CNN to achieve greater training and run time efficiency as input data has already been mapped to features that are useful to the task of classification performed by CNN. Some of the most expensive operations for auto-encoders involve matrix multiplications, transpose and computing the sigmoid function on large matrices. This covered in section TBD in more detail.

**Convolutional Neural Neural Networks (CNN)**: CNNs are inspired by visual cortex. Each Neuron in the cortex layer is activated by only a small subregion of the input image. These neurons are tiled together in a such a way as to capture the entire image. Activations from these are further pooled and fed to higher level neurons. Several activation and pooling layers could be stacked to form a hierarchy. At each level, the property of only connecting to spatially local neurons at the level below is preserved. Layers closer to and including the visual cortex recognize very basic and locally correlated features like edges while layers at higher level have a more global view of the input and recognize complex, non-linear features. The last layer is a classification layer like softmax which is fully connected to all the neurons in the previous layer. For a detailed explanation of CNN please refer to [2]

CNN consists of 2 basic operations: convolve and pool (and upsample during backpropagation). These operations are the most expensive in terms of CPU usage and form the most important part of our API

**Fast Iterative and Shrinkage Algorithm (FISTA):** FISTA is an iterative algorithm, similar to gradient descent, for optimizing Sparse Coding. Sparse coding, much like AE maps input to a set of hidden features. The difference lies in it applying the feature matrix W in the reverse order compared with AE. Here instead of multiplying W to input x, we have to solve the system of equations given by Ws = x, where s is the sparse code vector. During the training process for finding W from input x, estimation on W and s are iteratively optimized in 2 separate but dependent steps. Once W is trained, production data is still mapped to s using an iterative algorithm like FISTA. FISTA optimizes on 2 components. Just as in ISTA, it performs a gradient Descent on reconstruction penalty ||Ws -x||22, and applies a shrinkage function on s, to minimize the sparsity penalty |s|1. FISTA uniquely adds a momentum component based on the previous 2 value of the sparse codes. For more details, refer to [3].

## Section II: Implementation Details

Our methodology initially involved getting familiar with the UFLDL tutorial in MATLAB by implementing the required parts of the CNN, AE and Sparse Coding. The following steps detail the methodology and Figure 2 represents our workflow.

- Prototype and test the UFLDL exercise implementation of the algorithm in matlab
- Convert these implementations into python using Numpy
- Profile the python implementation to identify a set of optimizable operations
- Define an API consisting of optimizable operations
- Create reference implementation of the API using Numpy and Theano
- Test the implementations with and without GPU
- Analyze and compare performance of implementations

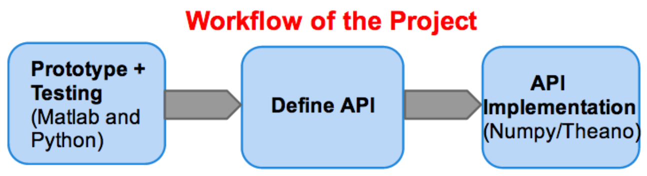

Workflow of the Project
Prototype + Testing (Matlab and Python)
Define API
API Implementation (Numpy/Theano)

Figure 2: Project Workflow

The code can be found in a github repositary at [4].

**Implementation Issues:** The following list while not complete gives a flavor of some of the implementation issues we faced:

**Choosing dimensions in CNN**: We found the CNN operations involving convolve and up sample can be exposed in the API as either 2-D or 4-D operations. Even though we felt that 2-D operations are easier to understand in the context of an API, we chose to expose them as 4-D operations as that allows better control over optimizing these algorithms. We implemented the same operations in 3 different ways:

1. As 2-D by looping over image and filter dimension
2. As 2-D by flattening from 4-D to 2-D
3. As 4-D

Option 3 gave us the greatest performance. The second option was better than the first but it was worse than the last as it required reshaping operations.

**Incompatible 4D operations**: Some of the operations e.g. kron() had 4-D versions which were not compatible with CNN's requirement of upsampling. For those we had to be creative by flattening to 2-D and doing a bunch of shape manipulations and that gave us an order of magnitude improvement in performance.

**Memory allocation**: To ensure that intermediate results across multiple operations are not shunted between GPU and host we wanted to provide greater control over memory allocation, placement and disposal. We haven't fully explored this issue. Based on our early understanding, we have provided operations like zero() and rand_initialize() which can be used to both initialize and control how memory is allocated and placed.

**Array ordering and indices**: We realized that the ordering of array dimensions and indices in Matlab was reverse to that of Numpy. This naturally brought up the question of which order to follow in our API. We have used the Matlab or Fortran order as suggested by our collaborator. To the end we had to right a bunch of wrappers to map from one order to the other.

1. Matlab follows Fortran indexing (leftmost index closely packed)
2. Numpy follows C indexing (rightmost index closely packed)

**Theano Constraints**: Not all operations were adopted for GPU usage (e.g. pool and rotate operations) because they were not directly accessible in Theano. Theano likes us to construct algorithms like CNN as one large symbolic expression which it then converts to a graph and optimizes internally. This approach is not directly amenable to our API which likes to break a large algorithm into small state change operations. For such operations we intend to write C-CUDA or PY-CUDA implementations in coming weeks.

## Section III: Results

### CNN

The figure and the pie-charts below identify the most expensive operations in our API for CNN. Figure 3 highlights the most important operations in the CNN API and their location in the CNN pipeline. The pie-chart in figure 4a,4b and 5 shows the 4 most expensive operations for CNN in Numpy and Theano (with GPU) respectively. When we started with Numpy, CNN would take 45 minutes to train over the MNIST dataset of 60K images over 3 epochs (using mini-batching with 256 images images in each batch and going through all the images in an epoch). Convolve used in filter_convolve and grad_convolve was the most expensive operation taking almost 65% of the total time. Moving to Theano and using 4-D version of convolve in theano, we get a speedup of 4.5 times with GPU (and 3 times without GPU). The time taken by convolve itself was now insignificant. Next we optimized upsample by creating a 4D version (by flattening the array into 2D and using a bunch of reshaping). This provided an overall speed-up of 20 times with GPU (and 6.5 times without GPU) compared to Numpy.

The dominant time is spent in the rotation operation used in filter_convolve and grad_convolve(), and it accounts for about 40% of the CPU. Another 30% is accounted by pool and upsample operations (the reshaping operation in the upsample being the dominant contributor). Once we implement these 3 operations in C-CUDA, we expect to see an even larger gain in performance. FIgure 6 gives a run time comparison of CNN across Numpy, Theano (no GPU) and Theano with GPU. The speedup we obtain in the last case validates the usefulness of the 4 most important API operations we have identified with respect to optimization.

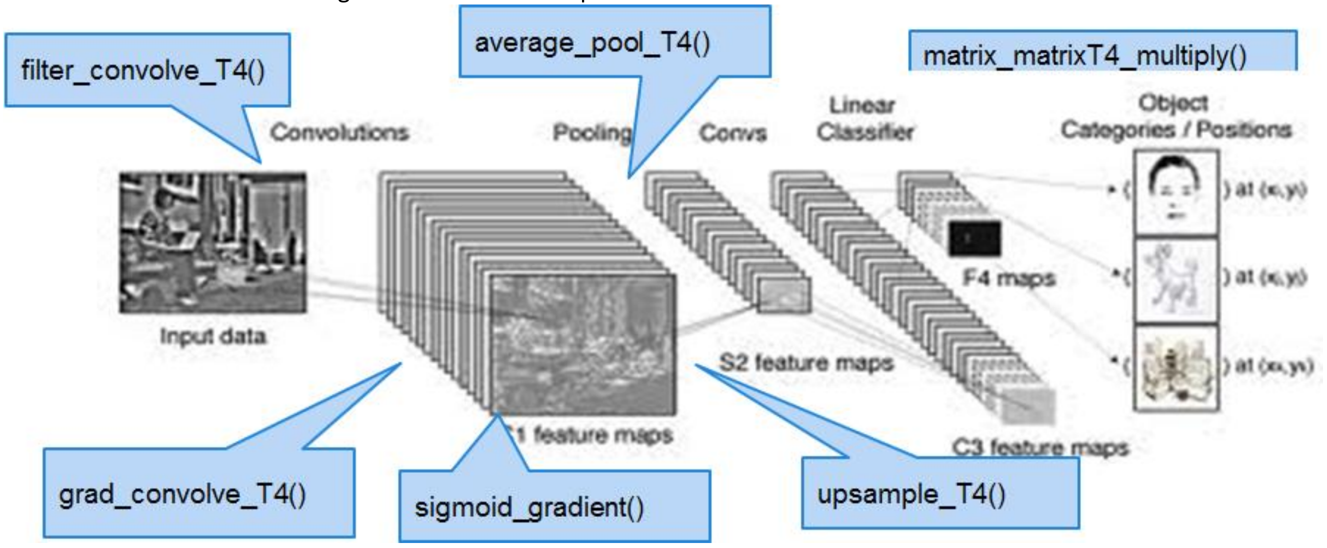
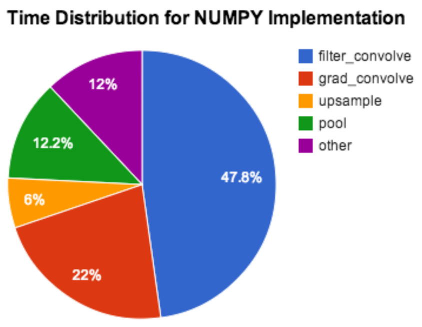


Figure -3 Convoluted Neural Networks



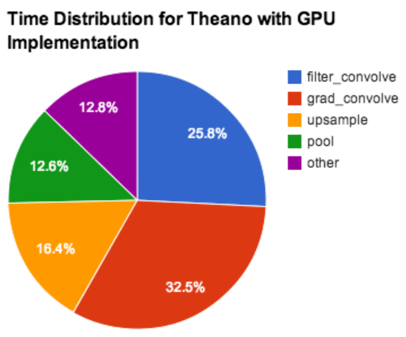


Figure 4a and 4b - Time Distribution by Functions for CNN

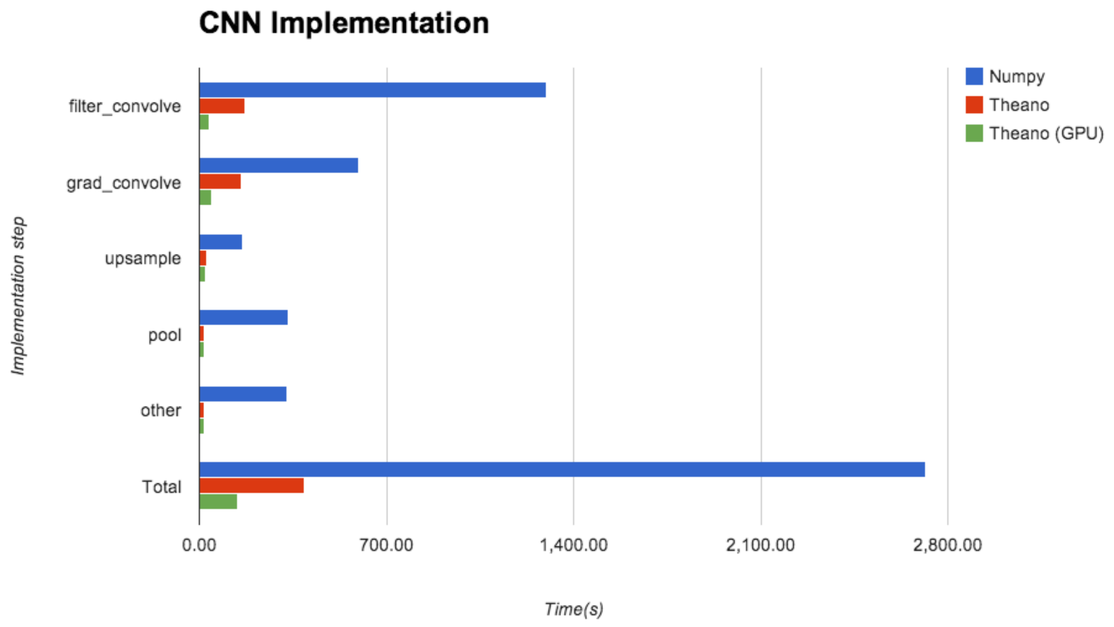


Figure 5 - CNN implementation comparison for Numpy, Theano and Theano with GPU

**Sparse Auto Encoders**: The figure and the pie-charts below identify the most expensive operations in our API for AE. Figure 6 highlights the most important operations in the AE API and their location in the AE pipeline. The pie-chart in figure 7a, 7b and 8 shows the 5 most expensive operations for AE in Numpy and Theano (with GPU) respectively. In the AE context, Theano with GPU performed 2 times faster than Theano without GPU and 3.5 times faster than Numpy. We optimized all the 5 operations to use GPU in theano and got descent performance gain. Now significant time is spent in transferring data between host and GPU. We still haven't figured out a clean way of forcing results of intermediate operations to stay on the GPU in Theano. Once we achieve this (or implement these operations in C-CUDA or PY-CUDA) we expect to see better resultsimplement these operations in C-CUDA or PY-CUDA) we expect to see better results.

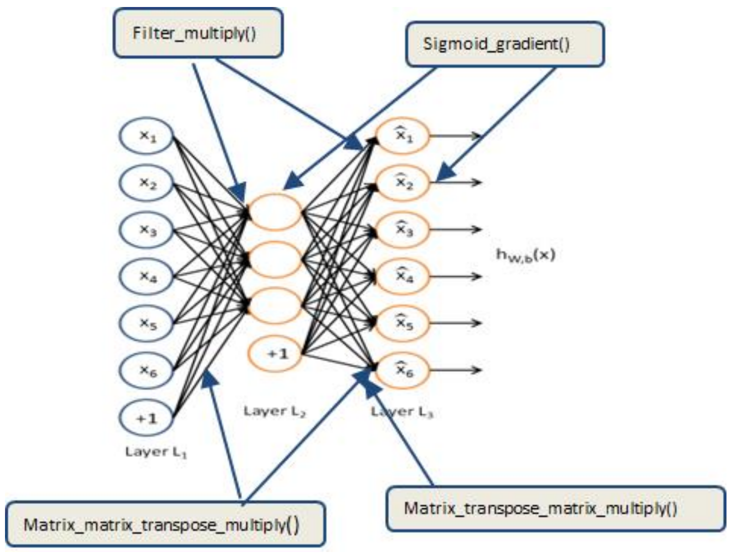
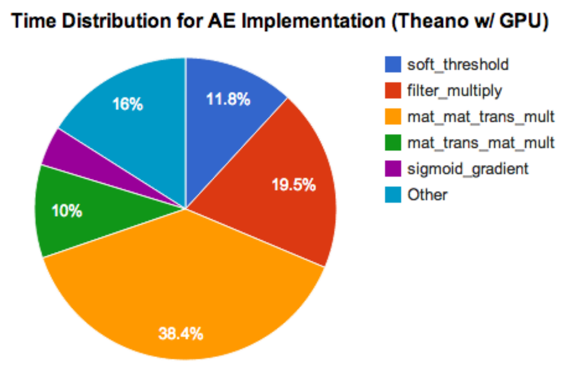


Figure 6 Sparse Auto Encoders



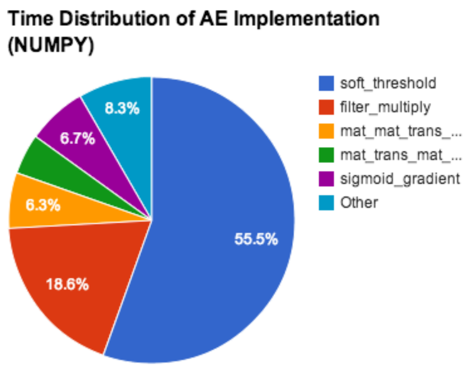


Figure 7a and 7b: Time Distribution by Functions for AE

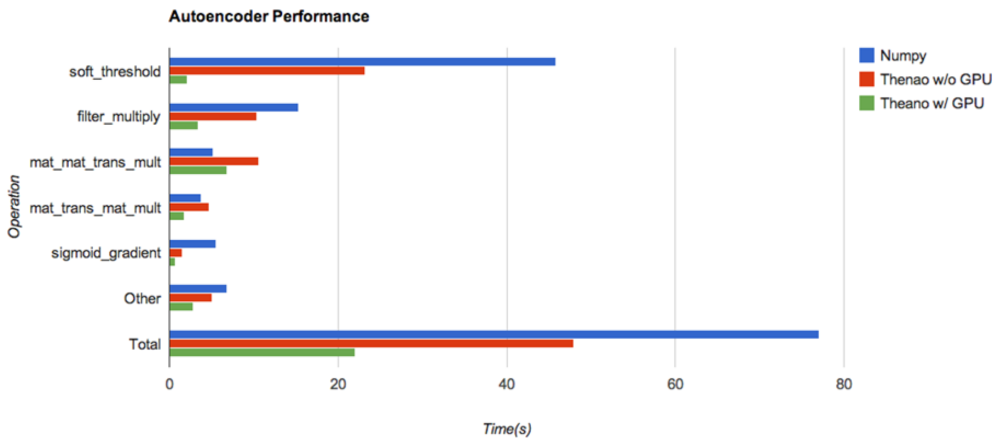


Figure 8 implementation comparison for Numpy, Theano and Theano with GPU

## Conclusion

We have completed the implementation of Autoencoders and CNN in Matlab and Python. We have identified a small set of basic operations which account for most of the CPU usage. These operations have been abstracted into an API. Reference implementations for this API were created using Numpy and Theano. The test results show that Theano on GPU performs 20x faster than Numpy, thus validating the utility of our API for optimization. Highly optimized implementation of these operations will allow Deep-Learning algorithms to be benchmarked for High-Performance Computing.

## References

[1]http://www.stanford.edu/~acoates/papers/CoatesHuvalWangWuNgCatanzaro_icml2013.pdf
[2] http://ufldl.stanford.edu/tutorial/index.php/UFLDL_Tutorial
[3] http://mechroom.technion.ac.il/~becka/papers/71654.pdf
[4] https://github.com/quaizarv/deeplearning-benchmark