# Deep Learning with Multiple Precision using GPUs

Pooja Mehta, Prateek Parab
Department of Computer Science, Illinois Institute of Technology, Chicago, IL, USA
pmehta27@hawk.iit.edu, pparab@hawk.iit.edu

**ABSTRACT**

At present, available computational resources have limited the training of extensive deep neural networks. Collection and manipulation of data items to produce meaningful information has become more complex and expensive in terms of data intensive computing compared to computation resources. A novel approach is to use deep learning to extract features instead of manual feature extraction. In this work, we investigate the effect of multiple precision data representation and computation on neural network training. Our main contribution is thorough evaluation of inspecting performance using single floating-point precision and double floating-point precision on text and images (black and white). The demonstration will be done by passing multiple floating-point precision data as an input and executing it with the help of GPUs. In literature, it has been observed that the performance achieved after execution of fixed-point precision had showcased better accuracy on GPU's compared to CPU. The utilization of GPUs allows parallel processing and reduce the computation time resulting in improved performance.

**INTRODUCTION**

Recent years have seen a resurgence of interest in deploying large-scale computing infrastructure designed specifically for training deep neural networks. Some notable efforts in this direction include distributed computing infrastructure using thousands of CPU cores [1], or high-end graphics processors (GPUs) [2, 3], or a combination of CPUs and GPUs scaled-up to multiple nodes [4, 5].

At the same time, the natural error resiliency of neural network architectures and learning algorithms is well documented, setting them apart from more traditional workloads that typically require precise computations and number representations with high dynamic range. It is well appreciated that in the presence of statistical approximation and estimation errors, high-precision computation in the context of learning is rather unnecessary [6]. Moreover, the addition of noise during training has been shown to improve the neural network's performance [7, 8, 9, 10]. Except for employing the asynchronous version of the stochastic gradient descent algorithm [11] to reduce network traffic, the state-of-the-art large-scale deep learning systems fail to adequately capitalize on the error-resiliency of their workloads. These systems are built by assembling general-purpose computing hardware designed to cater to the needs of more traditional workloads, incurring high and often unnecessary overhead in the required computational resources.

**BACKGROUND INFORMATION**

Deep learning has emerged as a new area of machine learning research since 2006 [12,13,14]. Deep learning (or sometimes called feature learning or representation learning) is a set of machine learning algorithms which attempt to learn multiple-layered models of inputs, commonly neural networks. The deep neural networks are composed of multiple levels of non-linear operations. Before 2006, searching the parameter space of deep architectures is a nontrivial task, but recently deep learning algorithms have been proposed to resolve this problem with notable success, beating the state-of the-art in certain areas [14]. A graphics processing unit (GPU) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device. A graphics processing unit can render images more quickly than a central processing unit because of its parallel processing architecture, which allows it to perform multiple calculations at the same time. In terms of precision, single precision floating-point arithmetic deals with 32-bit floating point numbers whereas double precision deals with 64 bits. The number of bits in double precision increases the maximum value that can be stored as well as increasing the precision (i.e. the number of significant digits).

**MOTIVATION**

To a large extent, the success of deep learning technique is contingent upon the underlying hardware platform's ability to perform fast, supervised training of complex networks using large quantities of labeled data. Such a capability enables

rapid evaluation of different network architectures and a thorough search over the space of model hyperparameters. Current research has resulted for low-precision fixed-point computations where adopting stochastic rounding during deep neural network training delivers results nearly identical as 32-bit floating point computations using CPU's. While taking real world applications into consideration, it is evident that the computations are complex and to be more precise it is essential for the computations to be in floating-point values. Instead of using CPU's with fixed-point precision, an effective strategy would be to use GPU's and examine the computation time for single and double precision.

## PROPOSED SOLUTION

We present an effective deep learning approach to test the single floating-point and double floating-point precision using GPU's to achieve low computation time. The project will be about exploring various technologies to change the numerical precision thus evaluating the precision and recall parameters for various types of data such as text, image etc. Data from MINIST and CIFAR-10 dataset would be trained using GTX 960M GPU present in the local machine then testing the data on the same GPU. Thus, the project is to test whether running the test at half precision really provides the same level of accuracy as that of the full precision datatype. If the full precision datatypes provide more accuracy as compared to the half precision datatype, then it would be contradictory according to current results. Technologies use will be Python with CUDA, CUDA toolkit, Keras at the backend for supporting complex numerical operation for the deep neural net.

In this work, we study the effect of limited precision data representation and computation on neural network training to analyze and compare with the existing results. Determining the precision of the data representation and the compute units is a critical design choice in the hardware (analog or digital) implementation of artificial neural networks. At present, we have developed a system that trains the neural net and evaluates the performance of multiple datatypes – floating point (16-bit, 32-bit and 64-bit) and integer (8-bit) on local GPU. Implementation is done using python and CUDA which is NVIDIA's language/ API for programming. To detect the files extracted and train the neural net cuDNN library has been imported to provide GPU accelerated functionality for common operations in deep neural nets. The MINIST handwritten dataset is used for training and testing purpose. Also, testing of one test case for floating point and integer data type is completed and the results have been evaluated to compute the accuracy of the system. Block Diagram for our model is as follows:

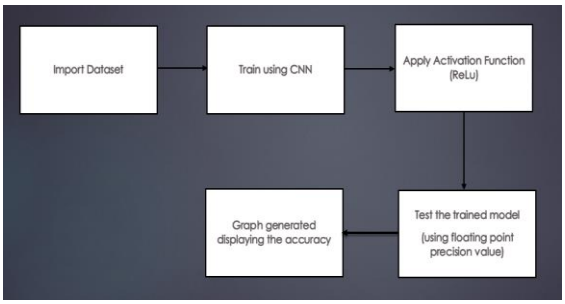


Fig. 1: Block Diagram of proposed solution

In the block diagram as shown above, initially we import two datasets individually, once the dataset is loaded it will be provided as input to our Convolution Neural Network (CNN) model which comprises of three layers deep. This CNN model trains the training data using the activation function Rectified Linear Unit (ReLu). If the input is greater than 0, the output is equal to the input. ReLUs' machinery is more like a real neuron in your body.

$$f(x) = max(x, 0)$$

ReLU activations are the simplest non-linear activation function you can use, obviously. When you get the input is positive, the derivative is just 1, so there isn't the squeezing effect you meet on backpropagated errors from the sigmoid function. Research has shown that ReLUs result in much faster training for large networks. After the data is trained, test datasets are provided as input and using floating point precision (16-bit, 32-bit and 64-bit) accuracy and loss is calculated.

## EVALUATION

In this empirical performance evaluation, we have used loss and accuracy as the evaluation parameters to test the efficiency of each precision value. Loss value implies how good or bad a certain model is behaving after each iteration of optimization. Ideally, reduction is expected in loss after several iterations. Loss is calculated on training and validation data. Accuracy of a model is usually determined after the model parameters are learned. Loss is not in percentage as opposed to accuracy. To achieve better results, loss should be least, and accuracy should be high.

MINIST handwritten dataset comprises of training set consisting of 60,000 examples, and a test set of 10,000 examples. The digits have been size-normalized and centered in a fixed-size images. To optimize the gradient descent, learning rate is configured as 0.01 and to train the model epochs is configured as 25. To test the result, we have taken the batch size as 100.

Execution for 2500 images and calculating the average cost (epoch) for five times results in following graphs that display the results of accuracy that is achieved.

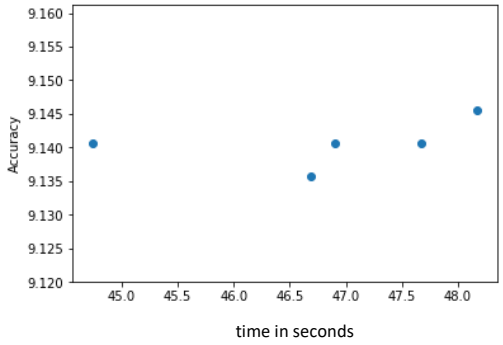


Fig. 2: Accuracy achieved for MINIST dataset using floating point(16-bit)

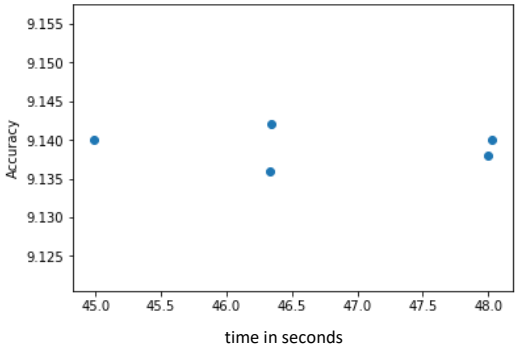


Fig. 3: Accuracy achieved for MINIST dataset using floating point(32-bit)

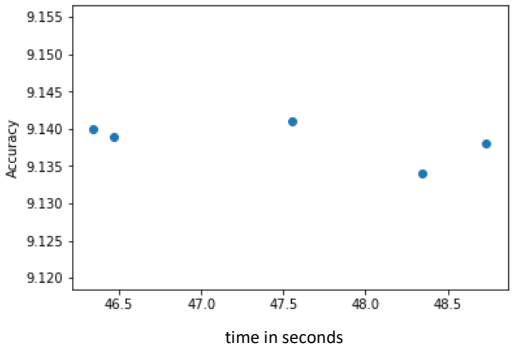


Fig. 4: Accuracy achieved for MINIST dataset using floating point(16-bit)

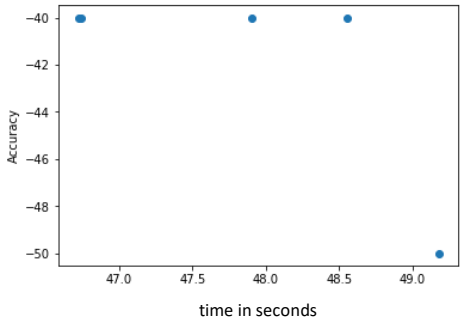


Fig. 5: Accuracy achieved for MINIST dataset using integer (8-bit)

As observed, from figure 2, 3 and 4 the average computation time is 21 215.25 seconds to 218.5 seconds and accuracy achieved is 99.01%. In case of figure 5, we observe that the accuracy plotted shows negative value.

A typical implementation of our Neural Network would be as follows:

- Define Neural Network architecture to be compiled
- Transfer data to our model
- Under the hood, the data is first divided into batches, so that it can be ingested. The batches are first preprocessed, augmented and then fed into Neural Network for training
- The model then gets trained incrementally
- Display the accuracy for a specific number of timesteps
- After training save the model for future use
- Test the model on a new data and check how it performs

For convenience we pickled the dataset to make it easier to use in python. The pickled file represents a tuple of three lists: the training set, the validation set and the testing set. Each of the three lists is a pair formed from a list of images and a list of class labels for each of the images.

An image is represented as numpy 1-dimensional array of 784 (28 x 28) float values between 0 and 1 (0 stands for black, 1 for white). The labels are numbers between 0 and 9 indicating which digit the image represents. The code block below shows how to load the dataset.

When using the dataset, we usually divide it in mini batches. We stored the dataset into shared variables and accessed it based on the minibatch index, given a fixed and known batch size. The reason behind shared

variables is related to using the GPU. There is a large overhead when copying data into the GPU memory.

If we would have copied the data on request (each minibatch individually when needed) as the code will do if you do not use shared variables, due to this overhead, the GPU code will not be much faster than the CPU code (maybe even slower). the training set, validation set, and testing set to make the code more readable (resulting in 6 different shared variables).

Since now the data is in one variable, and a minibatch is defined as a slice of that variable, it comes more natural to define a minibatch by indicating its index and its size. In our setup the batch size stays constant throughout the execution of the code, therefore a function will require only the index to identify on which data points to work.

The CIFAR-10 dataset consists of 60000 32x32 color images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class.

The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.

CIFAR 10 dataset comprises of 10 classes which include airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck. The classes are completely mutually exclusive. There is no overlap between automobiles and trucks. "Automobile" includes sedans, SUVs, things of that sort. "Truck" includes only big trucks. Neither includes pickup trucks.

In the dataset layout, the archive contains the files data_batch_1, data_batch_2, ..., data_batch_5, as well as test_batch. Each of these files is a Python "pickled" object produced with cPickle.

When the file is loaded, each of the batch files contains a dictionary with the following elements:

*data* -- a 10000x3072 numpy array of uint8s. Each row of the array stores a 32x32 color image. The first 1024 entries contain the red channel values, the next 1024 the green, and the final 1024 the blue. The image is stored in row-major order, so that the first 32 entries of the array are the red channel values of the first row of the image.

*labels* -- a list of 10000 numbers in the range 0-9. The number at index i indicates the label of the $i^{th}$ image in the array data.

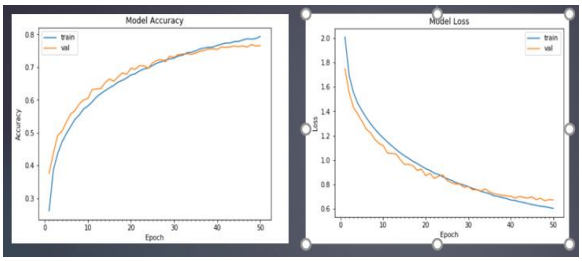


Fig. 6: Accuracy and Loss reported for CIFAR-10 using 16-bit floating point

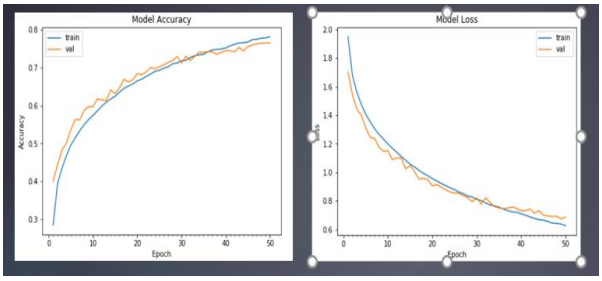


Fig. 7: Accuracy and Loss reported for CIFAR-10 using 32-bit floating point

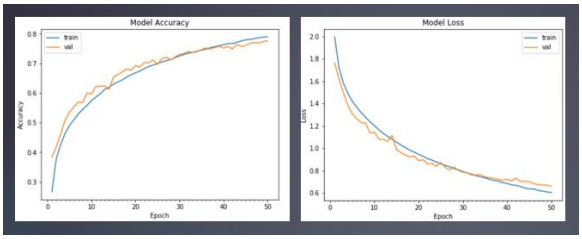


Fig. 8: Accuracy and Loss reported for CIFAR-10 using 64-bit floating point

As observed, the computation time to train the CIFAR-10 dataset consisting of 50000 images for figure 6 is 1092.01 seconds, figure 7 is 1109.41 seconds and figure 8 is 1119.01 seconds. From the above graphs it can be observed that the accuracy achieved 76.58% for 16-bit floating point, 76.45% for 32-bit floating point and 77.64% for 64-bit floating point. Also, the loss observed for all three cases is less than 0.8.

## RELATED WORK

Exploring the use of low-precision fixed-point arithmetic for deep neural network training with a special focus on the rounding mode has been adopted while performing operations on fixed-point numbers.

The motivation to move to fixed-point arithmetic (from the conventional floating-point computations) is two-fold. Firstly, fixed-point compute units are typically faster and consume far less hardware resources and power than floating-point engines. The smaller logic footprint of the fixed-point arithmetic circuits would allow for the instantiation of many more such units for a given area and power budget. Secondly, low-precision data representation reduces the memory footprint, enabling larger models to fit within the given memory capacity and lowering the bandwidth requirements. Cumulatively, this could provide dramatically improved data-level parallelism [16].

Previous studies have also investigated neural network training using different number representations. Iwata et al. [17] implements the back-propagation algorithm using 24-bit floating-point processing units. Hammerstrom [18] presents a framework for on-chip learning using 8 to 16-bit fixed-point arithmetic. In [8], the authors perform theoretical analysis to understand a neural network's ability to learn when trained in a limited precision setting. Results from empirical evaluation of simple networks indicate that in most cases, 8-16 bits of precision is sufficient for backpropagation learning. In [20], probabilistic rounding of weight updates is used to further reduce (< 8 bits) the precision requirements in gradient based learning techniques. While these studies provide valuable insights into the behavior of the limited precision training of neural networks, the networks considered are often limited to variants of the classical multilayer perceptron containing a single hidden layer and only a few hidden units.

Extrapolating these results to the state-of-the-art deep neural networks that can easily contain millions of trainable parameters is non-trivial. Consequently, there is a need to reassess the impact of limited precision computations within the context of more contemporary deep neural network architectures, datasets, and training procedures.

A recent work [21] presents a hardware accelerator for deep neural network training that employs fixed-point computation units, but finds it necessary to use 32-bit fixed-point representation to achieve convergence while training a convolutional neural network on the MNIST dataset. In contrast, few results show that it is possible to train these networks using only 16-bit fixed-point numbers, so long as stochastic rounding is used during fixed-point computations. To our knowledge, this work represents the first study of application of stochastic rounding while training deep neural networks using low precision fixed-point arithmetic. In our results, it is observed that training these networks is possible using 16-bit floating-point numbers.

## CONCLUSION

We present a novel deep learning approach towards computing the computation time for multiple floating-point precision values using GPUs. The experimental results on the datasets prove that 16-bit floating point require less computation time than 64-bit floating point value, although there was no significant bump in the accuracy from 16 to 64 bits. The positive outcome will serve as a basis for new advancements in terms of taking colored image as input or analyzing a video. Also, it can be used in real-world scenarios such as self-driving cars, finance applications, etc. The implementation of data-intensive computing on a larger scale combined with deep learning will prepare us not only for handling large datasets but also assist in analyzing and researching in the Data Analytics field.

## REFERENCE


[1] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Senior, A., Tucker, P., Yang, K., Le, Q. V., et al. Large scale distributed deep networks. In Advances in Neural Information Processing Systems, pp. 1223–1231, 2012.

[2] Ciresan, D. C., Meier, U., Gambardella, L. M., and Schmidhuber, J. Deep, big, simple neural nets for handwritten digit recognition. Neural computation, 22(12): 3207–3220, 2010.

[3] Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks. In Advances in neural information processing systems, pp. 1097–1105, 2012.

[4] Coates, A., Huval, B., Wang, T., Wu, D., Catanzaro, B., and Andrew, N. Deep learning with COTS HPC systems. In Proceedings of the 30th International Conference on Machine Learning, pp.1337–1345, 2013.

[5] Wu, R., Yan, S., Shan, Y., Dang, Q., and Sun, G. Deep image: Scaling up image recognition. arXiv preprint arXiv:1501.02876, 2015.

[6] Bottou, L. and Bousquet, O. The tradeoffs of large scale learning. In NIPS, volume 4, pp. 2, 2007.

[7] Murray, A. F. and Edwards, P. J. Enhanced MLP performance and fault tolerance resulting from synaptic weight noise during training. Neural Networks, IEEE Transactions on, 5(5):792–802, 1994.

[8] Bishop, C. M. Training with noise is equivalent to Tikhonov regularization. Neural computation, 7(1):108–116, 1995.

[9] An, G. The effects of adding noise during backpropagation training on a generalization performance. Neural Computation, 8(3):643–674, 1996.

[10] Audhkhasi, K., Osoba, O., and Kosko, B. Noise benefits in backpropagation and deep bidirectional pre-training. In Neural Networks (IJCNN), The 2013 International Joint Conference on, pp.1–8. IEEE, 2013.

[11] Recht, B., Re, C., Wright, S., and Niu, F. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In Advances in Neural Information Processing Systems, pp. 693-701, 2011.

[12] Arel, I., D. C. Rose, et al. (2010). "Deep Machine Learning - A New Frontier in Artificial Intelligence Research [Research Frontier]." Computational Intelligence Magazine, IEEE 5(4): 13-18.

[13] Bengio, Y. (2009). "Learning Deep Architectures for AI." Found. Trends Mach. Learn. 2(1): 1-127.

[14] Bengio, Y., A. Courville, et al. (2013). "Representation learning: A review and new perspectives."

[15] Bengio, Y., R. Ducharme, et al. (2003). "A Neural Probabilistic Language Model." Journal of Machine Learning Research 3: 1137-1155.

[16] Suyog Gupta, Ankur Agarwal, Kailash Gopalkrishnan, Pritish Narayanan Deep Learning with Limited Precision. ICML'15 Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37, July 06 - 11, 2015

[17] Iwata, A., Yoshida, Y., Matsuda, S., Sato, Y., and Suzumura, N. An artificial neural network accelerator using general purpose 24 bit floating point digital signal processors. In Neural Networks, 1989. IJCNN., International Joint Conference on, pp. 171–175. IEEE, 1989.

[18] Hammerstrom, D. A VLSI architecture for highperformance, low-cost, on-chip learning. In Neural Networks, 1990., 1990 IJCNN International Joint Conference on, pp. 537–544. IEEE, 1990.

[19] Holt, J. and Hwang, J.-N. Finite precision error analysis of neural network hardware implementations. Computers, IEEE Transactions on, 42(3):281–290, 1993.

[20] H¨ohfeld, M. and Fahlman, S. E. Probabilistic rounding in neural network learning with limited precision. Neurocomputing, 4(6):291–299, 1992.

[21] Chen, Y., Luo, T., Liu, S., Zhang, S., He, L., Wang, J., Li, L., Chen, T., Xu, Z., Sun, N., et al. Dadiannao: A machine-learning supercomputer. In Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on, pp. 609–622. IEEE, 2014.

**APPENDIX**

*Pooja Mehta:* Requirements to implement the project. Installation of cuDNN library. Documentation of the project. Coding in python to extract the dataset and train the neural net. Searching a dataset for colored images. Testing the performance for floating point precision (32-bits) and integer (8-bit) datatype.

*Prateek Parab:* Initial understanding of the project. Installation of CUDA and setting up the chameleon to run large datasets on GPU's. Coding in python to construct and test the model. Also, working on the parameter values to test and evaluate the results. Testing the performance for floating point precision (16-bits & 64-bits).