Final Report

Deep Learning on GPU with Limited precision

Chetan Gupta
cgupta5@hawk.iit.edu
A20378854

Abstract: - *Deep learning (also known as deep structured learning or hierarchical learning) is part of a broader family of machine learning methods based on learning data representations, as opposed to task-specific algorithms. Learning can be supervised, partially supervised or unsupervised. The training of deep neural networks is very often limited by hardware. Training of large-scale deep neural networks is often constrained by the available computational resources. We will going study the effect of limited precision data representation and computation on neural network training. Within the context of low precision floating-point computations, we will observe the rounding scheme to play a crucial role in determining the network's behavior during training. Lots of previous works address the best exploitation of general-purpose hardware, typically CPU clusters (1), FPGAs and GPUs (2). We are planning to train a set for state of art neural network on various benchmark MNIST dataset. They will be get trained for distinct formats like floating point (16, 32, 64 bit). For each dataset and for each those formats we will going to train them and test for each precision (like half precision, single precision and double precision).*

Introduction

To a large extent, the success of deep learning techniques 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. It should therefore come as no surprise that recent years have seen a resurgence of interest in deploying large-scale computing infrastructure designed specifically for training deep neural networks. The training of deep neural networks is very often limited by hardware. Traditional statistical machine learning operates with atable of data and a prediction goal. The rows of the table correspond to independent observations and the columns correspond to hand crafted features of the underlying data set. Then a variety of machine learning algorithms can be applied to learn a model that maps each data row to a prediction. More importantly, the trained model will also make good predictions for unseen test data that is drawn from a similar distribution as the training data.

Lots of previous works address the best exploitation of general-purpose hardware, typically CPU clusters (Dean et al., 2012), GPUs (Coates et al., 2009; Krizhevsky et al., 2012a) and FPGAs (Coates et al. Suyog Gupta, Ankur Agrawal and Kailash Gopalakrishnan, 2015). Faster implementations usually lead to state of the art results (Dean et al., 2012; Krizhevsky et al., 2012a).

Background

Deep learning is the name we use for "stacked neural networks"; that is, networks composed of several layers. The layers are made of nodes. A node is just a

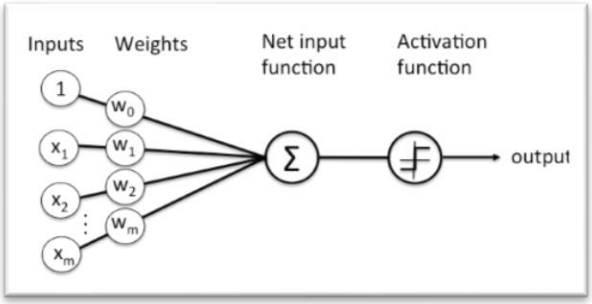


Figure 1:- One node description

place where computation happens, loosely patterned on a neuron in the human brain, which fires when it encounters sufficient stimuli. A node combines input from the data with a set of coefficients, or weights, that either amplify or dampen that input, thereby assigning

significance to inputs for the task the algorithm is trying to learn. (For example, which input is most helpful is classifying data without error?) These input-weight products are summed, and the sum is passed through a node's so-called activation function, to determine whether and to what extent that signal progresses further through the network to affect the ultimate outcome, say, an act of classification. Here's a diagram (Figure 1) of what one node might look like.

A node layer is a row of those neuronlike switches that turn on or off as the input is fed through the net. Each layer's output is simultaneously the subsequent layer's input, starting from an initial input layer receiving your data.

Pairing adjustable weights with input features is how we assign significance to those features regarding how the network classifies and clusters input. Such approaches always consist in adapting the algorithm to best exploit state of the art general-purpose hardware. Nevertheless, some dedicated deep learning hardware is appearing as well. Deep learning involves huge amount of matrix multiplications and other operations which can be massively parallelized and thus speed up on GPU-s.

A single GPU might have thousands of cores while a CPU usually has no more than 12 cores. Although GPU cores are slower than CPU cores, they more than make up for that with their large number and faster memory if the operations can be parallelized. Sequential code is still faster on CPUs. CPUs are designed for more general computing workloads. GPUs in contrast are less flexible, however GPUs are designed to compute in parallel the same instructions. Deep Neural Networks (DNN) are structured in a very uniform manner such that at each layer of the network thousands of identical artificial neurons perform the same computation. Therefore, the structure of a DNN fits quite well with the kinds of computation that a GPU can efficiently perform.

GPUs have additional advantages over CPUs, these include having more computational units and having a higher bandwidth to retrieve from memory. Furthermore, in applications requiring image processing (i.e. Convolution Neural Networks) GPU graphics specific capabilities can be exploited to further speed up calculations.

The primary weakness of GPUs as compared to CPUs is memory capacity on GPUs are lower than CPUs. The highest known GPU contains 24GB of RAM, in contrast, CPUs can reach 1TB of RAM. A secondary weakness is that a CPU is required to transfer data into the GPU card. This takes place through the PCI-E connector which is much slower than CPU or GPU memory. One final weakness is GPU clock speeds are 1/3rd that of high end CPUs, so on sequential tasks a GPU is not expected to perform comparatively well.

In summary, GPUs work well with DNN computations because (1) GPUs have many more resources and faster bandwidth to memory (2) DNN computations fit well with GPU architecture. Computational speed is extremely important because training of Deep Neural Networks can range from days to weeks. In fact, many of the successes of Deep Learning may have not been discovered if it were not for the availability of GPUs.

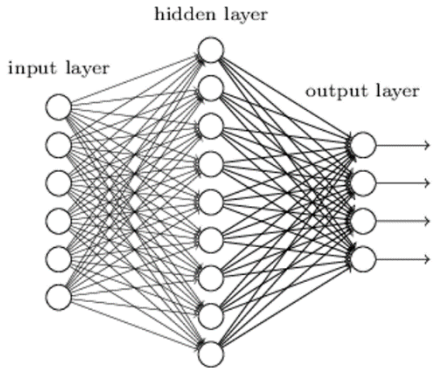


Figure 2 Basic neural network with 3 layer.

As we are doing our experiments on floating point precision (16, 32, 64 bit), so let's discuss a bit on floating point. Floating point formats are often used to represent real values. They consist in a sign, an exponent, and a mantissa, as illustrated in figure 1. The exponent gives the floating-point formats a wide range, and the mantissa gives them a good precision. One can compute the value of a single floating-point number using the following formula:

$$value = (-1)^{sign} \times \left(1 + \frac{mantissa}{2^{23}}\right) \times 2^{(exponent-127)}$$

Fig 3 shows the exponent and mantissa widths associated with each floating point format. In our experiments, we use half precision, single precision and double precision floating point format as our reference because it is the most widely used format in deep learning, especially for GPU computation. We show that the use of half precision floating point

format has little to no impact on the training of neural networks.

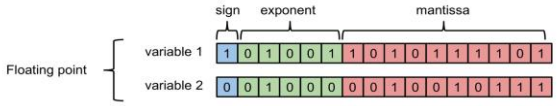


Figure 3: Floating point

Format	Total bit-width	Exponent bit-width	Mantissa bit-width
Double precision floating point	64	11	52
Single precision floating point	32	8	23
Half precision floating point	16	5	10

Table 1: Definitions of double, single and half precision floating point formats.

Motivation

Machine learning is one of the fastest-growing and most exciting fields out there, and deep learning represents its true bleeding edge. In this project, I developed a basic understanding of the motivation for deep learning, and design intelligent systems that learn from basic-scale datasets. Deep learning on neural networks and then on GPUs what fascinates me about this project idea. As Deep learning is topic on which lots of research is going on. After completion, we get that idea how any neural network learn and apply its learning to demonstrate the efficiency, As GPUs itself a new topic for me to study so this will help me in understanding the working of GPU over deep neural networks. MetaMind also presented a new model called Dynamic Coattention Network (DCN) for the question answering problem, which builds on a pretty intuitive idea. In September 2016, Google presented a new model used by their translation service called Google Neural Machine Translation (GNMT). This model is trained separately for each pair of languages like Chinese-English. GNMT results show that training it on multiple pairs of languages is better than training on a single pair, demonstrating that it is able to transfer the "translation knowledge" from one language pair to another. Several corporations and entrepreneurs have created non-profits and partnerships to discuss about the future of Machine Learning and making sure that these impressive technologies are used properly in favor of the community. OpenAI is a non-profit organization that aims to collaborate with the research and industry community, and releasing the results to public for free. It was created in late 2015, and started delivering the first results (publications like InfoGAN, platforms like Universe and (un)conferences like this one) in 2016. The motivation behind it is to make sure that AI technology is reachable for as many people as possible, and by doing so, avoiding the creation of AI superpowers. On the other hand, a partnership on AI was signed by Amazon, DeepMind, Google, Facebook, IBM and Microsoft. The goal is to advance public understanding of the field, support best practices and develop an open platform for discussion and engagement. It is a great time to be part of the recent Machine Learning developments. There are numerous projects and events are running on daily basis in this field. And this project is my first step towards it.

Proposed solution

We have created our deep neural network and tested for MNIST dataset. We have used Tensorflow to model our system and tested accuracy level for different precision, running over GPUs - Tesla V100 having 16 GB 8 vCPUs 61 GB memory(AWS) and Nvidia GPU Tesla P100 chipset(Chameleon) is GP100 having 3584 cores per GPU with RAM of 16 GiB GDDR5. Logic behind our model is that we have

```
def neural_network_model(data):
    # input_data * weights + biases
    hidden_l1 = {'weights': tf.Variable(tf.random_normal([784, n_nodes_hl1])),
                 'biases': tf.Variable(tf.random_normal([n_nodes_hl1]))}

    hidden_l2 = {'weights': tf.Variable(tf.random_normal([n_nodes_hl1, n_nodes_hl2])),
                 'biases': tf.Variable(tf.random_normal([n_nodes_hl2]))}

    hidden_l3 = {'weights': tf.Variable(tf.random_normal([n_nodes_hl2, n_nodes_hl3])),
                 'biases': tf.Variable(tf.random_normal([n_nodes_hl3]))}

    output_l = {'weights': tf.Variable(tf.random_normal([n_nodes_hl3, n_classes])),
                'biases': tf.Variable(tf.random_normal([n_classes]))}
```

Figure 5: All layer definitions in Code

created 3 layers deep neural network which functions as follows:

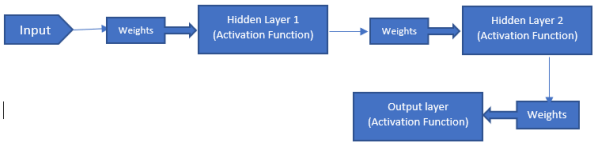


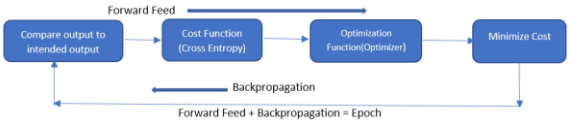
Figure 4: Iteration through layer one to layer 3

```
n_nodes_hl1 = 500
n_nodes_hl2 = 500
n_nodes_hl3 = 500
```

Where these are 3 different levels for deep network and 500 are the number of nodes per level. Definition for different layers and how they are connected

through each other with respect to weights and biases.

After getting the output we will test it with intended output and try to optimize the accuracy using softmax function (tf.nn.softmax_cross_entropy_with_logits) and Adam optimizer (tf.train.AdamOptimizer) while training and testing of the model at different precision. Currently, we have tested the model for 16, 32, 64-bit precision. We have checked for model for 30 epochs and tested the accuracy. Soft function measures the probability error in discrete classification tasks in which the classes are mutually exclusive (each entry is in exactly one class).



And Adam's Optimizer is an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. The method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. The method is also appropriate for non-stationary objectives and problems with very noisy and/or sparse gradients. The hyper-parameters have intuitive interpretations and typically require little tuning.

Adam Optimizer

An algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. The method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. The method is also appropriate for non-stationary objectives and problems with very noisy and/or sparse gradients. The hyper-parameters have intuitive interpretations and typically require little tuning. Some connections to related algorithms, on which Adam was inspired, are discussed. Empirical results demonstrate that Adam works well in practice and compares favorably to other stochastic optimization methods.

Algorithm : Adam, our proposed algorithm for stochastic optimization and for a slightly more efficient (but less clear) order of computation. gt2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are α = 0.001, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β1 and β2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Figure 6: Basic Algorithm for Adam Optimizer

Rectified linear unit Function(ReLU)

Instead of sigmoids, most recent deep learning networks use rectified linear units (ReLUs) for the hidden layers. A rectified linear unit has output 0 if the input is less than 0, and raw output otherwise. That is, 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. Most frameworks like TensorFlow and TFLearn make it simple to use ReLUs on the hidden layers.

The sigmoid function can be applied easily, the ReLUs will not vanish the effect during your training process. However, when you want to deal with classification problems, they cannot help much. Simply speaking, the sigmoid function can only handle two classes, which is not what we expect.

The Softmax Function

The softmax function squashes the outputs of each unit to be between 0 and 1, just like a sigmoid

function. But it also divides each output such that the total sum of the outputs is equal to 1 (check it on the figure above).

$$\begin{bmatrix}1.2\\0.9\\0.4\end{bmatrix} \rightarrow \boxed{\text{Softmax}} \rightarrow \begin{bmatrix}0.46\\0.34\\0.20\end{bmatrix}$$

The output of the softmax function is equivalent to a categorical probability distribution, it tells you the probability that any of the classes are true.

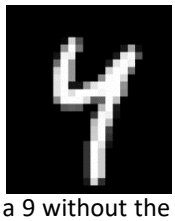
Mathematically the softmax function is shown below, where z is a vector of the inputs to the output layer (if you have 10 output units, then there are 10 elements in z). And again, j indexes the output units, so j = 1, 2, ..., K.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

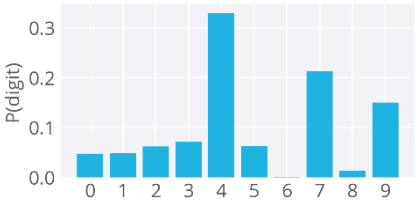
To understand this better, think about training a network to recognize and classify handwritten digits from images. The network would have ten output units, one for each digit 0 to 9. Then if you fed it an image of a number 4 (see below), the output unit corresponding to the digit 4 would be activated.

Building a network like this requires 10 output units, one for each digit. Each training image is labeled with the true digit and the goal of the network is to predict the correct label. So, if the input is an image of the digit 4, the output unit corresponding to 4 would be activated, and so on for the rest of the units.

For the example image above, the output of the softmax function might look like:



The image looks the most like the digit 4, so you get a lot of probability there. However, this digit also looks somewhat like a 7 and a little bit like a 9 without the loop completed. So, you get the most probability that it's a 4, but also some probability that it's a 7 or a 9.



The softmax can be used for any number of classes. It's also used for hundreds and thousands of classes, for example in object recognition problems where there are hundreds of different possible objects.

This is all about the basic functionalities of my model. Now finally comes with all the final modelling of my project. The graph containing the Neural Network (illustrated in the image above) should contain the following steps:

1. The input datasets; the training dataset and labels, the test dataset and labels (and the validation dataset and labels).
2. The test and validation datasets can be placed inside a tf.constant(). And the training dataset is placed in a tf.placeholder() so that it can be feeded in batches during the training (Adam Optimizer).
3. The Neural Network model with all of its layers. This can be a simple fully connected neural network consisting of only 3 layers, or a more complicated neural network consisting of 5, 9, 16 etc layers.
4. The weight matrices and bias vectors defined in the proper shape and initialized to their initial values. (One weight matrix and bias vector per layer.)
5. The loss value: the model has as output the logit vector (estimated training labels) and by comparing the logit with the actual labels, we can calculate the loss value (with the softmax with cross-entropy function). The loss value is an indication of how close the estimated training labels are to the actual training labels and will be used to update the weight values.

An optimizer, which will use the calculated loss value

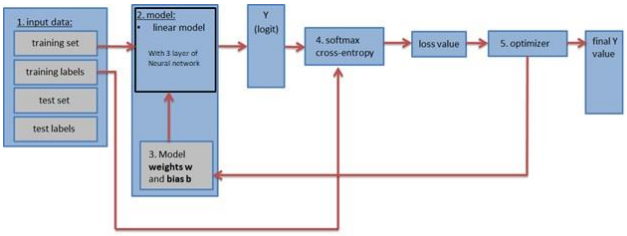


Figure 7: Neural network model

to update the weights and biases with backpropagation.

I have modeled my system over Python language having code of 83 lines for each bit precision (i.e. 16, 32 and 64 bit). We have used Tensorflow Library for

establishment our model, used CUDA 8 API to work on GPUs and CUDNN library for deep neural model.

MNIST dataset

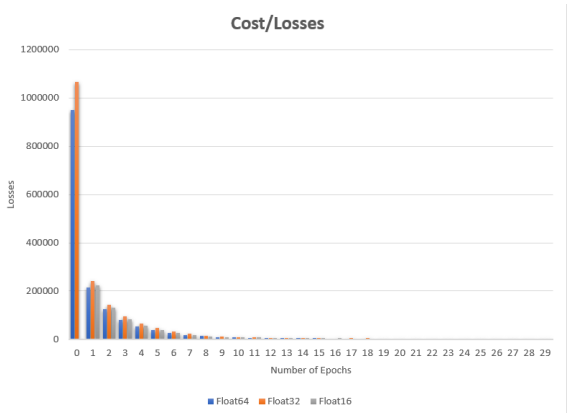
MNIST database of handwritten digits which has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image. The most commonly used sanity check. Dataset of 28x28, centered, B&W handwritten digits. We do not use any data augmentation (e.g. distortions) nor any unsupervised pre-training. We simply use minibatch Adam optimizer with momentum. We use a linearly decaying learning rate and a linearly saturating momentum. We regularize the model with dropout and a constraint on the norm of each weight vector.

We train our models on MNIST. The first is a permutation invariant (PI) model which is unaware of the structure of the data. It consists in two fully connected maxout layers followed by a softmax layer.

Evaluation

Cost/Loss evaluation

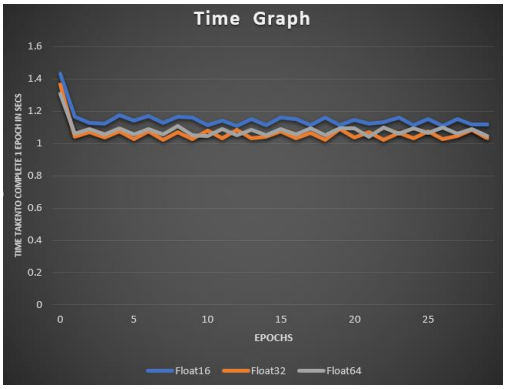
A cost function is a measure of "how good" a neural network did with respect to its given training sample and the expected output. It also may depend on variables such as weights and biases. A cost function is a single value, not a vector, because it rates how good the neural network did as a whole. We have evaluated the loss/error (i.e. compared output to the intended output) functionality while training over neural network. It basically shows how well our network is learning and after how many epochs it came to normal. There are things to point out while training our system. Firstly, it shows infinity errors on 1st epoch for 16-bit precision, Secondly, errors in 32 bit precision is more that 64 bit at 1st epoch. And lastly after 10 epochs error came to minimize.



Graph 1: Cost and Loss graph

Time Consumption

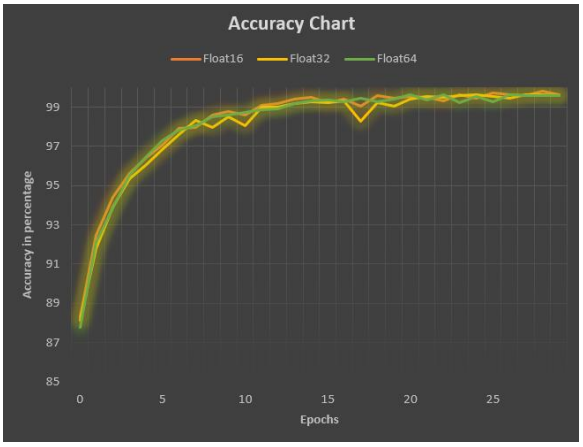
while training our system we found out time taken to complete 30 epochs of training are 34.443904, 31.857684 and 32.421133 secs for 16, 32, and 64 bit precision respectively, as well as it is taking more time for every epoch as compared to float 32 and float 64 which shows learning time for 16 bit is more than 32 and 64 bit because it takes time to convert 32 bit data to 16 bit while training data.



Graph 2:-Time taking by network while learning

Learning Accuracy

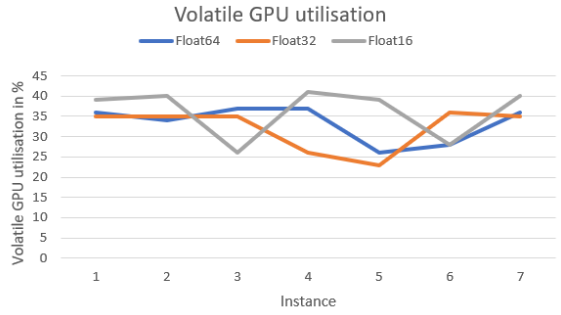
Learning accuracy means while training the system it calculates loss and after giving output it forwards to optimizing function (Adam optimizer in my case) to vary weights and biases to control the loss and improve training accuracy. In our system found out at 1st epoch 88.29%, 88.13% and 87.75%, and at last epoch 99.61%, 99.58% and 99.60% for 16, 32 and 64 bit respectively which doesn't showed much difference while training out system.



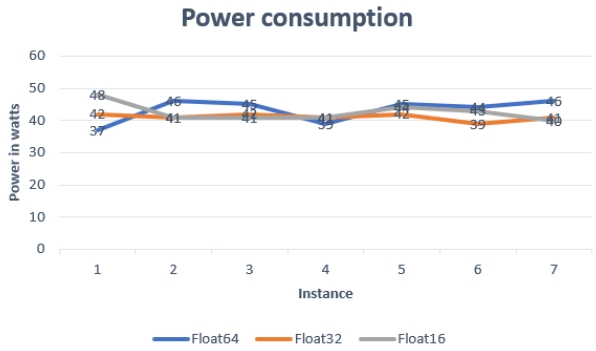
Graph 3: Accuracy while training

GPU utilization

This isn't directly related to the number of cores on the GPU, but it is related to the GPU performance in flops. While running Nvidia-smi command on random instances while running instances on the GPUs we found out for all cases it they are taking approximate equal utilization while running.



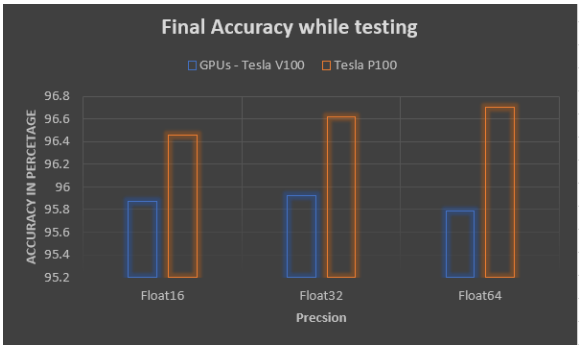
Power Consumption: - While running the instance we check power utilization for the GPUs the maximum limit was 300 Watts for one epoch and it uses 302, 288 and 298 Watts for 30 epochs for 64-bit, 32-bit and 16-bit precision. We can see that for our model and MNIST datasets its consumes almost equal power. Graph shows data for 7 random instances.



Graph 4: Power consumption on 7 random instances.

Accuracy while testing

This graph shows how accurate our model at different precision while testing and predicting random handwritten number images. We have run our model on 2 GPUs Tesla V100(AWS), Tesla P100(Chameleon). Its shows promising results for Float-16 precision and gave results which is approximately equal with other precision i.e. 32 Bit and 64 bit.

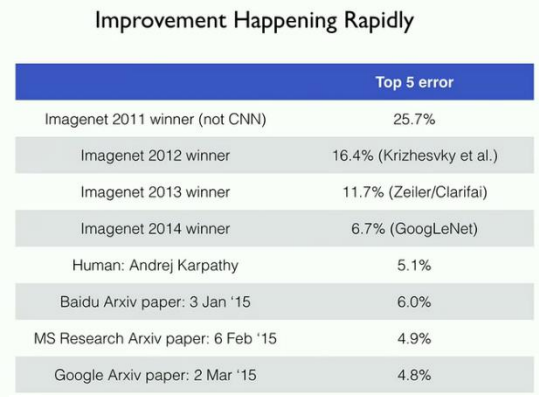


Graph 5: Final testing Accuracy

Related work: - Many works have been implements on deep learning on GPUs but in last 5 years growth in this field is exponentials. Below table said a lot about it.

There are one more paper which has been published (Ref#4) they have studied the effect of limited precision data representation and computation on neural network training. Within the context of low precision fixed-point computations, they observed the rounding scheme to play a crucial role in determining the network's behavior during training. Their results show that deep networks can be trained using only 16-bit wide fixed-point number representation when using stochastic rounding, and incur little to no degradation in the classification

accuracy. There is some other reference which basically tells about the part of deep learning like ref 5, 6. 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. Not surprisingly, a rich body of literature exists that aims to quantify the effect of this choice on the network's performance. However, a disproportionately large majority of these studies are focused primarily on implementing just the feed-forward (inference)



Improvement Happening Rapidly

	Top 5 error
Imagenet 2011 winner (not CNN)	25.7%
Imagenet 2012 winner	16.4% (Krizhesvky et al.)
Imagenet 2013 winner	11.7% (Zeiler/Clarifai)
Imagenet 2014 winner	6.7% (GoogLeNet)
Human: Andrej Karpathy	5.1%
Baidu Arxiv paper: 3 Jan '15	6.0%
MS Research Arxiv paper: 6 Feb '15	4.9%
Google Arxiv paper: 2 Mar '15	4.8%

Figure 8: Ref from Large-Scale Deep Learning for Building Intelligent Computer Systems by Jeff Dean (Google)(http://www.ustream.tv/recorded/60071572)

stage, assuming that the network is trained off in e using high precision computations. Some recent studies that embrace this approach have relied on the processor's vector instructions to perform multiple 8 bit operations in parallel (Vanhouckeetal., 2011), or employ reconfigurable hardware (FPGAs) for high-throughput, energy-efficient inference (Farabet et al., 2011; Gokhale et al., 2014), or take the route of custom hardware implementations (Kimetal.,2014; Merollaetal., 2014). A recent work (Chen et al., 2014) 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, our 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.

Conclusion

In this project tried to check out if lower precision will be good or bad for deep neural network and their algorithms. Specifically, for lower precision that is for 16 bit. Additionally, we implement high throughput and energy efficient architecture for matrix multiplication. Which gives us failure for int8 precision bit. Our results show that deep networks can be trained using only 16-bit wide floating-point number representation when using typecasting the float values at 32 and 64, and incur little to no degradation in the classification accuracy, power consumption and memory utilization. If I pursue this project in future I will try to run this model by varying the precision at float8 even lower. Planning to run and analyze the model for more datasets.

References

1) Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Le, Q., Mao, M., Ranzato, M., Senior, A., Tucker,P. ,Yang, K. ,and Ng ,A.Y.(2012). Large scale distributed deep networks. InNIPS'2012
2) Krizhevsky, A., Sutskever, I., and Hinton, G. (2012a). ImageNet classification with deep convolutional neural networks. In Advances in Neural Information Processing Systems 25 (NIPS'2012)
3) Large-Scale Deep Learning for Building Intelligent Computer Systems by Jeff Dean (Google)(http://www.ustream.tv/recorded/60071572)
4) Deep Learning with Limited Numerical Precision http://proceedings.mlr.press/v37/gupta15.pdf Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, Pritish Narayanan.
5) TRAINING DEEP NEURAL NETWORKS WITH LOW PRECISION MULTIPLICATIONS, Matthieu Courbariaux & Jean-Pierre David, YoshuaBengio(https://arxiv.org/pdf/1412.7024.pdf)
6) LOW PRECISION STORAGE FOR DEEP LEARNING Matthieu Courbariaux & Jean-Pierre David Yoshua Bengio(https://pdfs.semanticscholar.org/851c/27d7cdb74b0b21bd84a9333bca106f486713.pdf)

7) A complex-valued neural dynamical optimization approach and its stability analysis Songchuan Zhanga, Youshen Xia b,∗, Weixing Zhengc
8) Video by By Robert Ober, Chief Platform Architect, NVIDIA https://atscaleconference.com/videos/gpus-and-deep-learning/
9) Wiki Page for Deep learning https://en.wikipedia.org/wiki/Deep_learning
10) Vanhoucke, V., Senior, A., and Mao, M. Z. Improving the speed of neural networks on CPUs. In Proc.Deep Learning and Unsupervised Feature Learning NIPS Workshop, 2011.
11) Building an Efficient and Scalable Deep Learning Training System Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman, Microsoft Research (https://pdfs.semanticscholar.org/043a/fbd936c95d0e33c4a391365893bd4102f1a7.pdf)
12) Adam: A Method for Stochastic Optimization Diederik P. Kingma, Jimmy Ba (Submitted on 22 Dec 2014 (v1), last revised 30 Jan 2017 (this version, v9)) (https://arxiv.org/pdf/1412.6980.pdf)
13) Gokhale, V., Jin, J., Dundar, A., Martini, B., and Culurciello, E. A 240 G-ops/s mobile coprocessor for deep neural networks. In Computer Vision and Pattern Recognition Workshops (CVPRW), 2014 IEEE Conference on, pp. 696–701. IEEE, 2014
14) Xavier Glorot, Antoine Bordes and Yoshua Bengio (2011). Deep sparse rectifier neural networks (PDF). AISTATS. (http://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf)
15) The MNIST database by Yann LeCun(The MNIST database by Yann LeCun)