uDNN: Micro Deep Neural Networks with Variable Precision Computations

Department of Computer Science, Illinois Institute of Technology

Cohen, Raphaël
Illinois Institute of Technology
Chicago, USA
A20432518

Jhumkhawala, Milap
Illinois Institute of Technology
Chicago ,USA
A20409159

Abstract—**Deep learning algorithms are playing an important role in computer vision and image recognition tasks. Their applications range from self-driving cars to intelligent voice assistants. Deep learning algorithms are approximations of the target functions(actual properties), thus they cannot be truly accurate but accurate enough. From the engineering point of view, most important metrics of a DNN are speed of training, prediction, accuracy, performance and energy consumption. Recent trends in deep learning involves exploring lower precision operations for increasing speed, usability and energy efficiency without losing too much quality of prediction. If computing the model using lower precision gives us similar accuracy, that might save us a lot of training time, energy and can be compiled on various lower end processors and GPUs, thus saving dedicated resources in the process.**

I. Introduction

Deep learning models are a complicated network of layers of neurons. The more complex a problem gets, deeper the model needs to be in order to perform better, this also means more amount of computation is required.

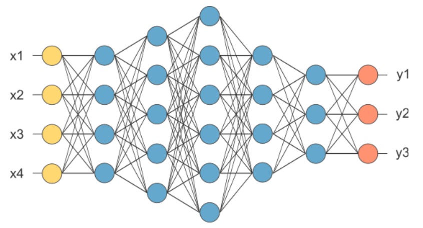


Fig. 1. Example of a dense deep learning network

The current deep learning frameworks are more like a black box, they do not really tell how they work. They only expose a few parameters we can tweak for our implementation. That is why we need a simpler and more customizable, transparent program to work with. The purpose of the project is to evaluate how does using variable precision(8-bit, 16-bit, 32-bit and 64-bit) on different kind of neural network model affect the quality, the performance and the speed the of model.

II. Problem Statement

Most of popular the deep neural network frameworks used usually focus on optimizing the model itself and not the underlying driver functions, like changing the precision or modifying the runtime environment or instruction set architecture of the hardware. Since these make up for the foundation of any processor based computation, they are kind of left unexplored in terms of how it might affect the neural network performance. Thus a question that arises is "Are we computing more than required"?

III. Related Work

[1] In this paper they evaluate the increase in compute density with change in precision. They estimate till what point determine till what point we can approximate the precision without losing performance.
[2] [7] In these papers they suggest that using different precision at different layers linear increase in performance with right combination of precision.
[3] [4] These papers talk about how using fixed point instead of floating point without much loss of accuracy.

[4] There are many Nvidia GPUs that have support for lower precision. The performance statistics(FLOPS) are also provided
[5] This paper talks about optimizing the parallel routines for optimal compilation and runtime for the deep learning models.

IV. Proposed Solution

We want to measure the impact of variable precision on both, the speed and the precision of the neural network. To do so, we need to build our own neural network with both, C++ and Tensorflow in order to be able to configure it as we want. We have classified possible areas in which we can focus our research to get valuable results and perform comparison. The three main aspects we will focus on to optimize the results:

- Software Optimization
- Model Optimization
- Hardware Optimization

Development Environment/ Language : C++, Python
Performance Standard: Tensorflow, Keras
We decided to work with C++ as it is a low level language and fast therefore it allows extreme flexibility and customization. We used Python to run Tensorflow and Keras on various datasets and models to create benchmarks that would allow us to detect optimal parameters to maintain a good accuracy while minimizing the training time. These are the areas we will be targeting for each of the three main aspects:

A. *Software Optimization(Driver Function Modification)*

- Effect on accuracy of DNN model with respect to various precision(quarter, single, half and double).
- Effect of change in precision on speed with which the model can be trained. Provided the hardware remains the same even before changing the precision.
- Data loss the number of bits is reduce. This might be a tradeoff for faster training time. How does that impact the quality of the model and its capability to discern patterns?

B. *Model Optimization(Neural Net Modification)*

- Reaction of different DNN algorithms to change in the precision.
- Reaction of different kind of datasets to change in precision. Is there a specific combination of dataset type, precision and algorithm that can yield fast training time with high accuracy?
- Effect of adding more layers in the neural network(8 -bit or 16-bit or 32-bit or 64-bit layers), on the training time positively or negatively One of the proposed work suggest that using different precision at different layers linear increase in performance with right combination of precision.
- Effect of the optimization algorithms on the accuracy of the model and how each of them behave when we change the number of bits(Stochastic gradient descent, AdaGrad, RMSprop, Adam)
- Vanishing Gradient: The gradient can vanish if it is too small, leading to an incapacity for the neural network to learn anything. How worse can it get with smaller precision?

C. *Hardware Optimization(CPU/GPU/TPU)*

- Effect of changing the hardware architecture on the performance of the model.
- Parallelising and multi-threading the computation of training of data set.

V. Testbed

- [1] CPU: 2.5 Ghz Intel i5 core processor, 4 cores
- [2] CPU: i7-8650U, 4 cores, 8 threads
- [3] GPU: GTX 1080 2560 NVIDIA CUDA cores
- [4] GPU: Tesla K80, 4492 NVIDIA CUDA cores
- [5] GPU: Tesla P100, 3584 NVIDIA CUDA cores
- [6] Tensor Processing Unit

VI. Performance Evaluation

A. *CPU Benchmarks*

1) Variable Precision Training Time and Accuracy

The most important metrics to focus are the model accuracy and training time taken by the model. Since using variable precision it is expected that training time should decrease with decrease in precision and accuracy should increase with increase in precision. To test that theory, one of the simplest dataset, the MNIST Handwriting was selected. It is a dataset of 64000 training binary(black and white) images and 10000 test images.. A two-layered network is used with 300 neurons in each layer. The activation functions used for the two layers is logistic and identity. The data type for weights and biases is floating point(16,32,64). We trained this model using a Testbed[1] CPU. The benchmark of model accuracy for 16-bit, 32-bit and 64-bit over an epoch of 100 is shown in the Fig 2.

It can be inferred from the Fig 2 that 32-bit and 64-bit perform very well and identical. They both learn very quickly and achieve a stagnant accuracy fast. Since the machine on which the model was run is 32-bit architecture, it performs the best. The 16-bit does not perform as good as others, it learns much slowly, but over 100 epoch it eventually achieves identical accuracy. If there was a cap of 25 on epoch, then the accuracy of 16-bit would be very low compared to 32-bit and 64-bit.

This conclusion is very much dependent on the dataset. If the dataset was different (grey-scale), then the results would be much different.

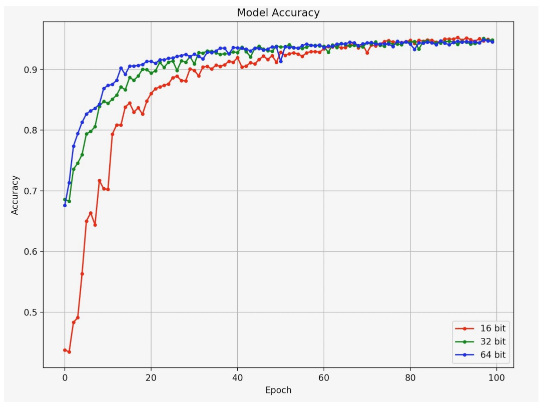


Fig. 2. 2-Layered Model Accuracy with variable precision

The other important metric is training time. It is expected of lower precision to consume less time. To test that theory we use the same model we used to benchmark accuracy and the same dataset. The benchmark of model training time(in milliseconds) for 16-bit, 32-bit and 64-bit over an epoch of 100 is shown in the Fig 3.

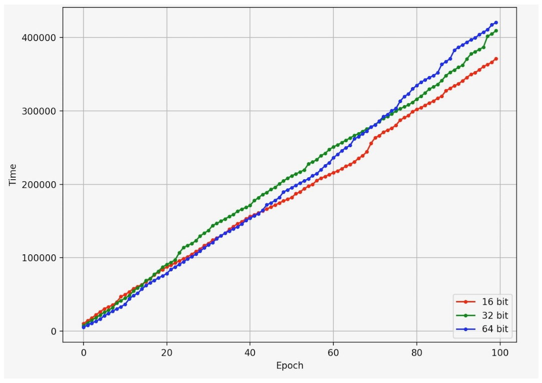


Fig. 3. 2-Layered Model Training Time with variable precision

It can be inferred from the figure 3 that there is not much difference between the precisions. They take identical amount of time to train. With 16-bit being least and 64-bit highest, with minor difference.

2) *Variable Activation Functions*

Activation function is an important part of the deep learning model. They determine how values will be assigned to weights and biases. They also define how efficiently back propagation will work. We ran the same model and dataset used in previous metric evaluation. A combination of three activation functions is used, hyperbolic_tangent, logistic, identity for the two deep layers. The benchmark of accuracy of the model over epoch of 100 with 32 bits precision and for a combination of activation functions is shown in the Fig 2.1.

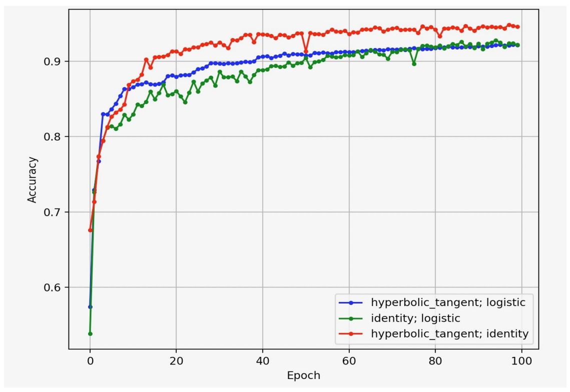


Fig. 4. Model Accuracy with variable activation functions

From the above Fig 4, it can be inferred that all combinations learn fast and identical rate in the start with a fixed precision. From the graph it is evident that combination of hyperbolic_tangent for first layer and identity for second layer as activation functions achieves maximum accuracy, higher than the rest of the combinations. Choosing the right activation functions for the model is tricky, as each model comes with their own set of advantages and disadvantages. The reason to pick activation functions as metric to evaluate was to prove that activation functions have an effect on the performance of the model.

3) *Reduced Precision Accuracy*

Benchmarking variable precision generally mean the standard supported precision(8,16,32,64). These are the only allowed type of precision variables can have. There is no defined way to create custom precision, for example there is no 12-bit floating point, not only it is not directly supported by the hardware architecture but also by most compilers. One of the method to create custom precision is to limit the decimal point of the floating point values. Clipping or rounding decimal point values leads to loss of precision and information. Although the effect of this on behaviour of the model is not certain and definite when compared to switching from 32-bit precision to 16-bit precision. But the accuracy of the model is affected and that is a metric to evaluate. The time required by the model to train is not a metric worthy enough to evaluate in this case, even though decimal point is limited or clipped off, it still occupies same amount of space in the memory. The arithmetic operation during run time might be faster but it is very minor. One of the easiest to achieve reduced precision is to use round() function in python which takes in two arguments, the floating point value to round and number of decimal points to round to. Using this function we limited the forward feed and backward propagation values that are communicated between the layers. The values of weights and biases were also limited using this function to result in reduced precision. Using the same model and dataset used in the first metric evaluation, the floating point values in the model are limited to maximum of five decimal points and minimum of zero decimal points, which essentially becomes an integer. The following figure displays the benchmark of the model accuracy over an epoch of 100 with decreasing decimal points from five to zero.

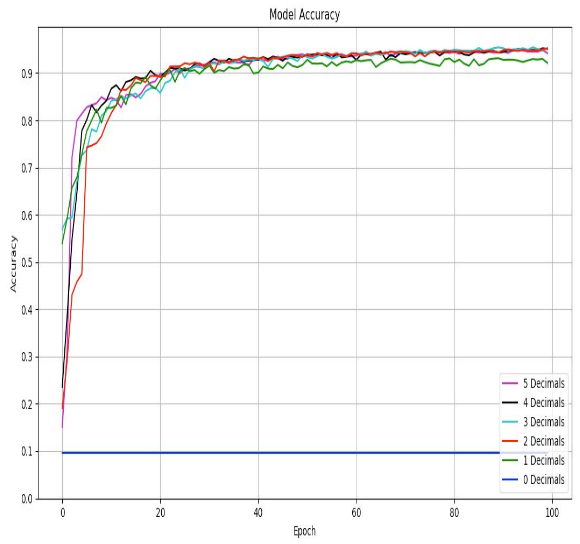


Fig. 5. Model accuracy with reduced precision

From the above Fig 5 it can be inferred using round() function to create reduced precision does have an effect on the model. The model accuracy decreases with decrease in decimal points. For zero decimal point, the floating point becomes an integer. Integers do not generate great accuracy as weights and biases are limited to only whole numbers.Lesser decimal point models also learn slowly as compared to greater decimal points. Thus this is the result of using reduced precision on original 16-bit floating point data type. The precision and information loss would be greatly reflected in a dataset of colored images where the range of values of weights and biases are much greater.

There are other ways to truly create custom precision using bitwise operators. Benchmarking the model using those precision would be major next steps in future work. The following figure Fig 6 shows the zoomed in portion of the graph to display accuracies achieved by all the decimal point values.

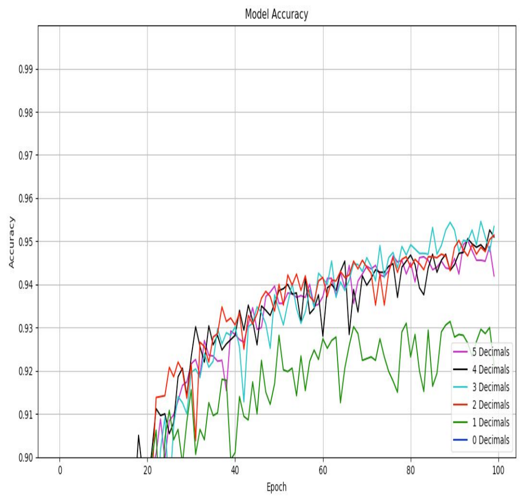


Fig. 6. Enlarged version of model accuracy with reduced precision

B. *GPU Benchmarks on different models with Tensorflow*

In this part, we will focus on benchmarking different model architectures with Tensorflow, allowing us to compute our calculations on different GPUs.

1) *Visualizing the differences*

First of all, we thought that it would be interesting to start our model diversification with something visual.

Auto-encoders are models that are used to automatically compress images or data without using well known compression algorithms. These models are constituted of two parts, the first one is the 'encoding' one and its goal is to compress the input. The second one is the 'de-encoder', it takes the compressed data as input and has to output a decompressed version of it while minimizing the loss.

Here is a simplified diagram of how it is supposed to work:

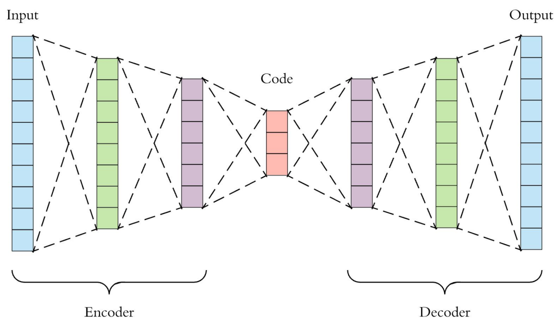


Fig. 7. Autoencoder model architecture

Based on this architecture, two models has been build. One very simple, only fully connected dense layers to use as a reference (we will call it the shallow model), and another one more complex, with convolutional layers (CNN model). Convolutional neural networks are very popular when it

comes to image recognition and this is why we decided to use them in this context. As we need to define a comparison metric, we will use the loss. It is a measure of how much data has been lost during the compression and decompression phases.
All those tests were performed using an Nvidia GTX 1080, no training times were measured as we only wanted to visualize the difference and they would all have been equal since this gpu only natively supports 32 bits.
Here are the results for the shallow model, it uses 6 dense layers, half of them for the encoding part and half for decoding. This network compresses the image size by four.

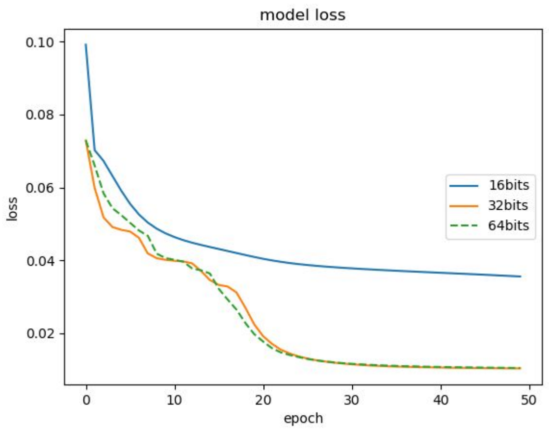


Fig. 8. Model loss for shallow auto-encoder

At first, the three precisions were very close in terms of loss value. However, we can clearly see that both, 64 bits and 32 bits have stood out by achieving a much better loss function minimization than the 16 bits precision. During the training, 16 bits suffered from gradient vanishing because of underflow issues. Here is a representation of some numbers that were encoded and de-encoded using this network:

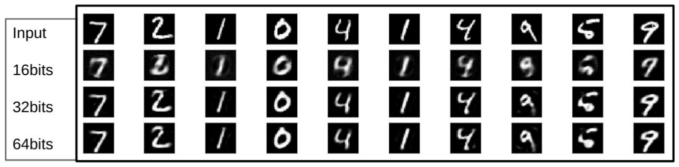


Fig. 9. Visual Representation of MNIST dataset images encoded and decoded using the shallow auto-encoder

It appears very clearly that 32 bits and 64 bits are reaching approximately the same precision while the 16 bits precision is struggling to return a clear image of the input. The following figure shows model loss using the CNN network and our intention is to see if it performs better.

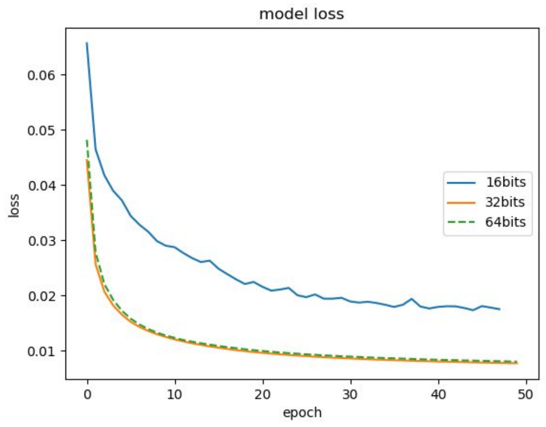


Fig. 10. Model loss for deep CNN auto-encoder

We can immediately see that the learning curve are much smoother and that the three final losses are lower than with the previous model. Again, 64 and 32 bits are almost identical whereas the 16 bits precision is still having difficulties to further minimize the loss. Here are the visual results:

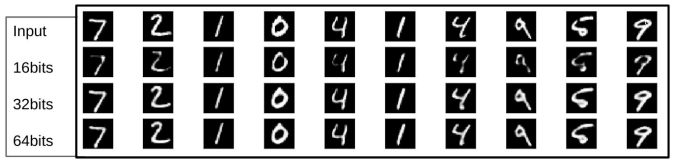


Fig. 11. Visual Representation of MNIST dataset images encoded and decoded using the deep CNN auto-encoder

Same overall conclusion as for the previous model, however, we can see that the 16 bits is achieving a sharper reconstruction of the output. Finally, we decided to use the two exact same models but to perform a denoising task instead of an encoding one. We added some noise to the MNIST dataset to be able to do this benchmark.

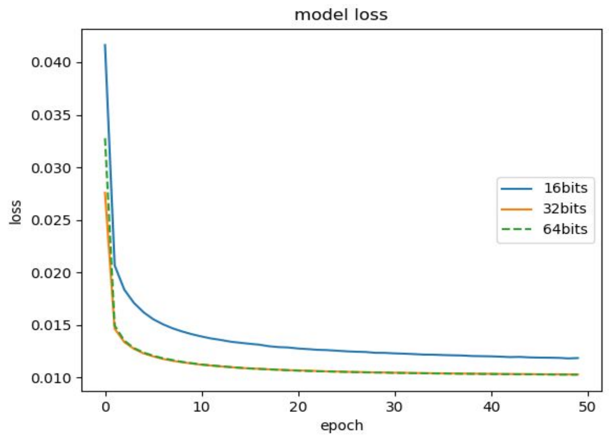


Fig. 12. Model loss for deep CNN denoising

This time, the gap between the three values is much smaller and the learning seems to happen at the same rate as well.

It would appear that for this kind of task, 16 bits could be preferable depending on the training time.

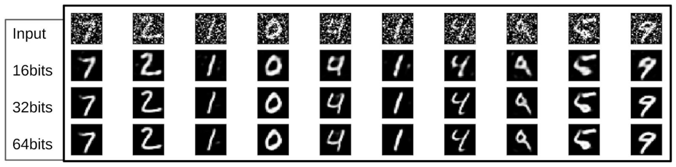


Fig. 13. Visual representation of denoising MNIST dataset images

As suggested by the loss function graph, the three precisions are performing greatly when it comes to denoising images even though the 16 bits is still behind.

2) *Simple predictive model for MNIST*

Now that we have a good understanding of what to expect with this precision change, we can build a more classic, predictive model for the same MNIST dataset. To be coherent with our previous part, we again, build two models, a shallow one with only dense layers and a deep convolutional one. These are the results in accuracy for our test set using the shallow model.

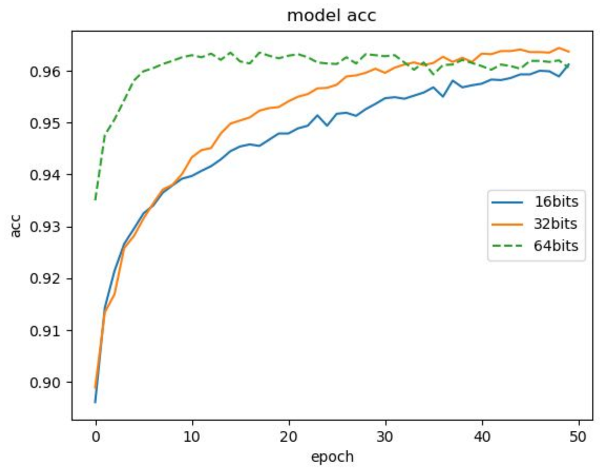


Fig. 14. Shallow Model Accuracy

The 64 bits precision clearly takes the lead at first when it comes to learning speed, followed by the 32 bits. However, at the end we observe that the curves are confounding themselves. As this is the accuracy for the test set and not the train one, we cannot precisely say which one is performing best at the end. We can only observe tendencies. Convolutional models are again, very popular for image recognition and classification. Here are the results for the deep CNN network:

We can observe immediately from Fig 15 that all differences that were present in the shallow model disappeared. There is still a minor lead at the start for the 64 bits precision but it quickly vanishes.

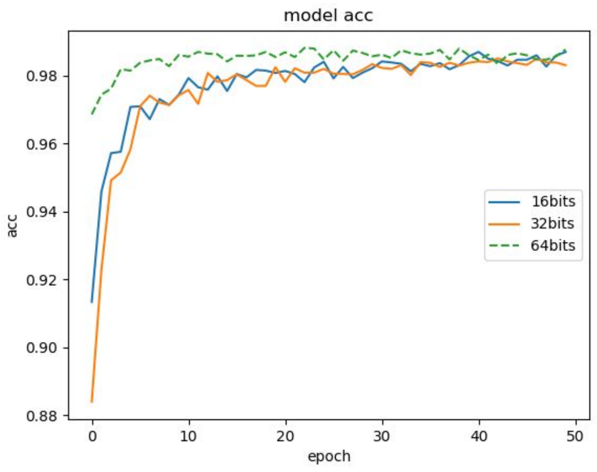


Fig. 15. Deep CNN Model Accuracy

It is worth mentioning as well that the final overall accuracy also increased. It would then appear that when it comes to classification, using a 16 bits precision and a well conceived model, can lead to very similar if not identical results than when using higher precisions.

3) *Using CIFAR10 to make results more robust*

Our previous results were hinting us that it could be possible to achieve great performances with lower precision, but only with well thought out models. We want to make this claim more robust by testing the same models with a new, more complex dataset, CIFAR10. As for the metrics, we will be using the training time on two different GPUs (K80 and P100) and the accuracy to compare our results. Varying variable precision is the major goal of this project and this is why we wanted to find a way of testing float 8 bits precision or lower. Keras framework has some very interesting built-in features that allowed us to clip each weights, biases, gradients and values while training the network. For this project we focused on simulating 8 bits precision but we could have tried more unusual values if we had had more time to do so. As a reference for our simulated 8 bits code we used this definition:

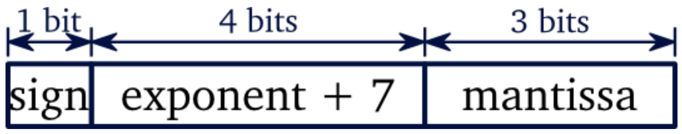


Fig. 16. Definition of standard 8 bit floating point data type

This allow us to calculate the minimum and maximum values possible for an 8 bits float.
The smallest non-zero number would be: 0.03125 and the largest value would be 01011111 and have value of 3.3875. With the sign bit we can also use negative values that have the same range. For the remainder of this report all benchmarks for the 8 bits precision has been constructed using this

clipping method for the weights, biases, gradients and values of the network. As the dataset is more complex than the previous one, we automatically decided to go for a convolutional model. We used dropouts to be able to train the model for longer while avoiding overfitting.

Another advantage of using this architecture is that it is a network that has less weights than a fully-connected dense one. It means that we will have less clipping to perform while training. Here is a summary of our model and we can clearly see that the highest density of weights are at the end, where the dense layer is. We used a softmax activation function as the output because we have a multi-class problem and we need a function that we will be able to interpret as probabilities that sum to 1.

```
       OPERATION           DATA DIMENSIONS   WEIGHTS(N)   WEIGHTS(%)

           Input   #####      3   32   32
          Conv2D    \|/  -------------------       896     0.0%
            relu   #####     32   32   32
         Dropout    | || -------------------         0     0.0%
                   #####     32   32   32
          Conv2D    \|/  -------------------      9248     0.4%
            relu   #####     32   32   32
    MaxPooling2D   Y max -------------------         0     0.0%
                   #####     32   16   16
          Conv2D    \|/  -------------------     18496     0.8%
            relu   #####     64   16   16
         Dropout    | || -------------------         0     0.0%
                   #####     64   16   16
          Conv2D    \|/  -------------------     36928     1.5%
            relu   #####     64   16   16
    MaxPooling2D   Y max -------------------         0     0.0%
                   #####     64    8    8
          Conv2D    \|/  -------------------     73856     3.1%
            relu   #####    128    8    8
         Dropout    | || -------------------         0     0.0%
                   #####    128    8    8
          Conv2D    \|/  -------------------    147584     6.2%
            relu   #####    128    8    8
    MaxPooling2D   Y max -------------------         0     0.0%
                   #####    128    4    4
         Flatten   ||||| -------------------         0     0.0%
                   #####        2048
         Dropout    | || -------------------         0     0.0%
                   #####        2048
           Dense   XXXXX -------------------   2098176    87.6%
            relu   #####        1024
         Dropout    | || -------------------         0     0.0%
                   #####        1024
           Dense   XXXXX -------------------     10250     0.4%
         sigmoid   #####          10
```

Fig. 17. CNN model used with simulated 8-bits and all GPU comparisons

We will now compare our results for different GPUs and different precisions. Figure 18 is the training accuracy evolution over 100 epochs. Figure 19 is the test accuracy evolution over 100 epochs. With these new benchmarks, we observe the same phenomenon for 64, 32 and 16 bits as for the MNIST dataset. For those first three precisions we have extremely similar results even though we had to clip gradients values for the 16 bits to avoid underflow. However, the 8 bits precision network has a much slower learning curve and it appears to reach its maximum at approximately 45% accuracy when the other ones can go up to 95% for the train set and 80% for the test set.

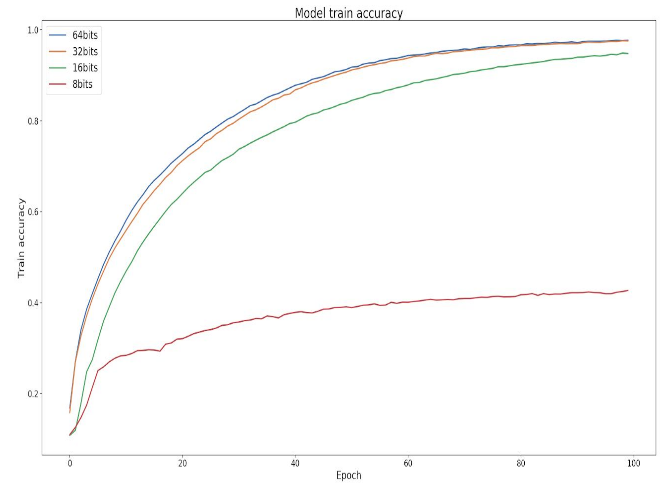
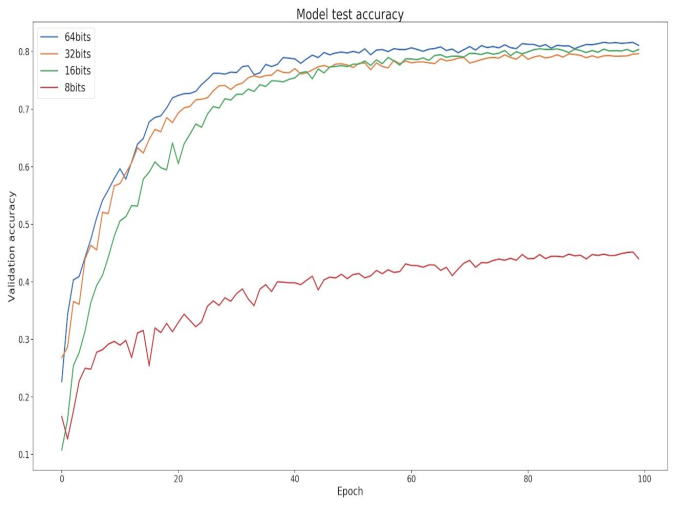


Fig. 18. Simulated 8-bits with 16,32,64 bits precision model Train accuracy



Now considering training time. Training time benchmark for K80 GPU:

Fig. 19. Simulated 8-bits with 16,32,64 bits precision model Test accuracy

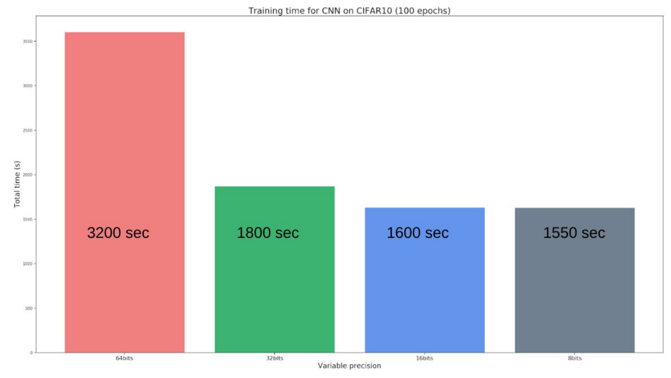


Fig. 20. Model training time for CIFAR10 dataset on Tesla K80 GPU

In the figure 20, can see that there is a huge, but expected, gap in training time when going down from 64 bits precision to 32 bits. We are able to observe this because both GPUs, the K80 and P100 natively support 64 and 32 bits operations. The training time doesn't change much when using 16 bits because the K80 doesn't support 16 bits operation, which means that we have the same training time for 32, 16 and 8 bits (as the latter is only simulated 8 bits with 16 bits float).

The next step is to compare these results with a more advanced GPU that supports 16 bits operations.

Training time benchmark for P100 GPU:

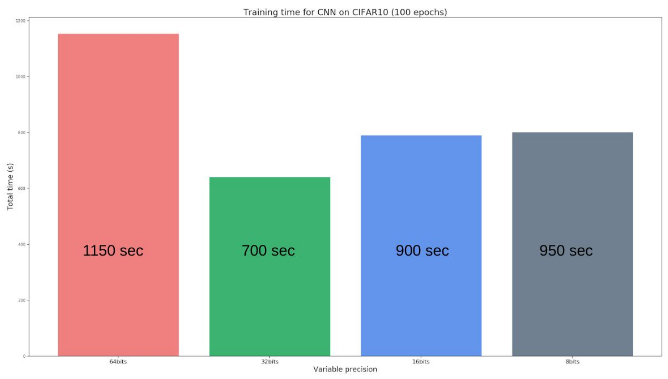


Fig. 21. Model training time for CIFAR10 dataset on Tesla P100 GPU

First of all, we can see that the overall training times for all precisions were almost divided by 2 ~ 3. Again, it is clear that training the model for 32 bits is much faster than for 64 bits (as expected, approximately 2 times faster). However, even though the P100 natively supports 16 bits operations, we do have an increase in training time for the 16 and 8 bits precision compared to the 32 bits. The justification we came up with to explain this anomaly is as follow: since value clipping has been used on the 16 bits as well as on the 8 bits network, it would be appropriate to think that this is what has penalized the training time for those two precisions.

But why wouldn't the same phenomenon be observed for the K80 benchmark ? We think that, proportionally, the time taken to clip values is much lower for the K80 configuration than for the P100 one. It means that value clipping is still impacting the K80 but it is less noticeable as this extra time taken is drowned into the overall slower performances of this GPU.

Finally, those benchmarks prove that it is possible to achieve good accuracy even with a low precision 16 bits model, but going down to 8 bits require some major adjustments to make it work. As for the training times, the results for 64 and 32 bits were expected but it would appear that it is more complex than anticipated to scale down training times for 16 bits.

VII. Exploratory Analysis

A. *Exploring custom precision*

One of the most important lead to explore with this project would be to not only configure our network to simulate an 8 bit precision but to try a large variety of in between precisions like 10 or 12 bits. By doing so, we wouldn't be able to compare training time but we could precisely measure the threshold for which the accuracy starts decreasing beyond reasonable.

B. *Bfloat16*

The bfloat16 floating-point format is a computer number format occupying 16 bits in computer memory; it represents a wide dynamic range of numeric values by using a floating radix point. This format is a truncated (16-bit) version of the 32-bit. The following figure shows the difference between bfloat 16 and 32-bit standard float.

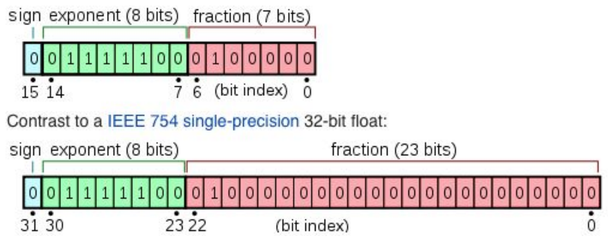


Fig. 22. Architecture of bfloat16 vs float32 data type

The bfloat16 is a compact version of the 32-bit float. The 32-bit float has 8 exponent bits, which are used for signed values. In the standard 16-bit float, the exponent bits are reduced to 5 bits, thus reducing the range of 16-bit float. But in bfloat16, there are 8 exponent bits which preserves 32-bit float range of ~1e-38 to ~3e38. This makes conversion between bfloat16 and floating-point 32 simple and efficient with minor loss in accuracy. The dynamic range of bfloat16 is similar to that of a IEEE single precision number. Relative to floating-point 32, bfloat16 sacrifices precision to retain range. Range is mostly determined by the number of exponent bits, though not entirely. Lower precision makes it possible to hold more numbers in memory, reducing the time spent swapping numbers in and out of memory. Also, low-precision circuits are far less complex. Together these can benefits can give significant speedup. The following graph shows the model accuracy when bloat16 is used compared to float32 on a Tensor Processing Unit and Tesla K80 GPU.

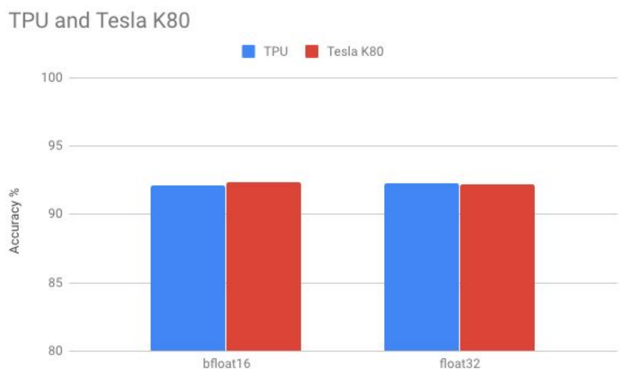


Fig. 23. Model Accuracy using bfloat16 and float32 running on TPU and Tesla K80

It can be inferred from the above figure that accuracies achieved by both data type are identical and thus bfloat16 can be used to replace float32 data type.

VIII. Issues Faced

A. Vanishing Gradient Problem

This is a problem faced when the model is trying to learn. It leads to difficulty in training and affects the performance of the model. It affects Backward Propagation the most. For example the sigmoid activation function is taken into consideration. The following figure shows the range of values of the function and range of values when taken derivative of it.

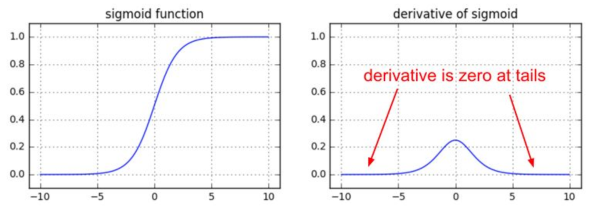


Fig. 24. Sigmoid and Derivative of Sigmoid Function

The derivative at the tails is zero and the max value is 0.2. Thus the gradient values during back propagation becomes so small with respect to the parameters that the early layers cannot learn properly as the weight correcting value in backpropagation is very small. Which means even a large change in the value of parameters for the early layers doesn't have a big effect on the output. This poses a problem for all gradient based models. One of the methods to solve this problem with tensorflow is to use gradient constraints (gradient clipping and norm adjustments) when dealing with gradient descent optimizer functions. The optimizer function requires some kind of user input as to how much optimization needs to be done (i.e. regulating values for the gradients). Overall, gradient based optimization algorithms require a good understanding of the activation functions used and their limitations, in order to tackle this problem. The syntax is tf.train.GradientDescentOptimizer().

B. 8-bit precision on 32/64-bit system architecture

The 8-bit precision is not natively supported by tensorflow, which has made its benchmarking very difficult. We had to find a workaround to be able to test its precision while keeping the structure of our model intact. We had to use gradient clipping and gradient norm constraints alongside our gradient optimizer function, as well as weights and values constraints/clipping on our layers. Two versions of the same model were created. One with weights, biases and values constraints and one without (for 64, 32, 16 bit precisions). In order to achieve these results, we created our own layer constraints with Keras Backend allowing us to monitor and tweak each variable in our network.

IX. Future Work

- Develop a neural net implementation in CUDA.
- Use PTX hardware instructions for NVIDIA GPU.
- Develop a more efficient model for TPU.Implement the model over colored image dataset.
- Develop an efficient multi threading implementation.
- Develop an implementation using integers to achieve similar accuracy.

X. Conclusion

This project helped us understand deep learning, neural networks in depth, since we had to implement own from scratch, even though we ended up using python frameworks to test more complex models. DNN are very complex and a breakthrough in field of artificial intelligence, but without optimal compilation, runtime environment and hardware they can be very time and resource hogging. But in this project we learnt how to optimize the neural network to run as fast as possible while maintaining great accuracy. Overall, we came to the conclusion that the most widespread precisions, 64, 32, and 16 bits are relatively easy to use for neural networks in general and that the difference in prediction accuracy between the three is not significant. However, when it comes to going further down to 8 bits, we were not able to get good performances and we think that 8 bits networks might only be useful in some very specific conditions as they require much more configuration adjustments.

Acknowledgment

Thank you to Prof Ioan Raicu and TA Alex Orhean for guiding us in the right direction for this project.

References

[1] Srivatsan Krishnan , Piotr Ratusziak , Chris Johnson , Duncan Moss , and Suchit Subhaschandra, "Accelerator Templates and Runtime Support for Variable Precision CNN", Intel Corporation University of Sydney.

[2] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on. IEEE, 2016, pp. 1–12.

[3] \Philipp Gysel, Mohammad Motamedi & Soheil Ghiasi, "HARDWARE-ORIENTED APPROXIMATION OF CONVOLUTIONAL NEURAL NETWORKS", Department of Electrical and Computer Engineering University of California, Davis.

[4] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, "cuDNN: Efficient Primitives for Deep Learning", NVIDIA, Santa Clara, USA.

[5] Matthieu Courbariaux & Jean-Pierre David, Ecole Polytechnique de Montreal and Yoshua Bengio, Universite de Montreal, CIFAR Senior Fellow, "TRAINING DEEP NEURAL NETWORKS WITH LOW PRECISION MULTIPLICATIONS".

[6] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, "Deep Learning with Limited Numerical Precision", IBM T. J. Watson Research Center, Yorktown Heights, NY 10598.

[7] Jürgen Schmidhuber, "Deep learning in neural networks: An overview".

[8] Tensorflow Playground, a visualization of how deep neural network work by Google.