Deep Learning in Object Detection

Shringa Bais, Prasanna Shanmuganathan

Abstract: Deep-learning neural networks have proven to be very successful for a wide range of recognition tasks across modern computing platforms. However, the computational requirements associated with such deep nets can be quite high. Here, we are going to investigate the effect of the complex network depth on its accuracy in the large-scale image recognition setting. The training of deep neural networks is very often limited by hardware. We explore techniques to analyse and compute efficiency and performance of Deep Convolutional Neural Networks with limited precision data representation (E.g. quarter precision and half precision data). Our motive is to train a neural network on certain benchmark datasets like CIFAR-10, MNIST and ImageNet with four distinctive formats of precision (8-bit,16- bit,32-bit and 64-bit) and assess the impact of the limited precision data types on the predictive accuracy and quality of those trained models. We have discussed a specific technique called Quantization to reduce the precision of the data type and ensure the consistency in the efficiency of the model.

1. Introduction:

Deep Learning has achieved unparalleled success with large-scale machine learning. Deep Learning models are used for achieving state-of-the-art results on a wide variety of tasks including Computer Vision, Natural Language Processing, Automatic Speech Recognition and Reinforcement Learning. Mathematically this involves solving a complex non-convex optimization problem with order of millions or more parameters. Solving this optimization problem - also referred to as training the neural network is a compute-intensive process that for current state-of-art networks requires days to weeks. Once trained, the DNN is used by evaluating this many-parameter function on specific input data - usually referred to as inference. While the compute intensity for inference is much lower than that of training, inference also involves significant amount of compute. Moreover, owing to the fact that inference is done on a large number of input data, the total computing resources spent on inference is likely to dwarf those that are spent on training. Due to the large and somewhat unique compute requirements for both deep learning training and inference operations, it motivates the use of non-standard customized arithmetic and specialized compute hardware to run these computations as efficiently as possible . Furthermore, there some theoretical evidence and numerous empirical observations that deep learning operations can be successfully done with much lower precision.

Several works have addressed the reduction of model size, through the use of quantization , low-rank matrix factorization , pruning, architecture design , etc. Recently, it has been shown that weight compression by quantization can achieve very large savings in memory, reducing each weight to as little as 1 bit, with a marginal cost in classification accuracy . However, it is less effective along the computational dimension, because the core network operation, implemented by each of its units, is the dot-product between a weight and an activation vector. Hence, substantial speed ups are possible if, in addition to the weights, the inputs of each unit are quantized to low-bit.

Training neural networks is done by applying many tiny nudges to the weights, and these small increments typically need floating point precision to work .Taking a pre-trained model and running inference is very different. One of the magical qualities of deep networks is that they tend to cope very well with high levels of noise in their inputs. If you think about recognizing an object in a photo you've just taken, the network has to ignore all the CCD noise, lighting changes, and other non-essential differences between it and the training examples it's seen before, and focus on the important similarities instead. This ability means that they seem to treat low-precision calculations as just another source of noise, and still produce accurate

results even with numerical formats that hold less information.

In this paper we discuss about a advanced technique, referred to as Quantized Neural Network (QNN), for quantizing the neurons and weights during inference and training. We have conducted experiments on two real time datasets : CIFAR-10 and MNIST.

2. Background Information :

A convolutional neural network can have tens or hundreds of layers that each learn to detect different features of an image. Filters are applied to each training image at different resolutions, and the output of each convolved image is used as the input to the next layer. The filters can start as very simple features, such as brightness and edges, and increase in complexity to features that uniquely define the object as the layers progress. When working with large amounts of data and complex network architectures, GPUs can significantly speed the processing time to train a model. One method for creating a convolutional neural network to perform object recognition is to train a network from scratch. The architect is required to define the number of layers, the learning weights, and number of filters, along with other tunable parameters. Training an accurate model from scratch also requires massive amounts of data, on the order of millions of samples, which can take an immense amount of time to train.

Reducing precision for weights and activations of a neural network has significant power-performance implication on system design. Low-precision not only allows increasing compute density, but also reduce pressure on the memory sub-system. Most of the current solutions are focused on compressing the model , going as low as binary weights, which allows storing the model on the limited on-chip local memory. However, activations need to be fetched from external memory or I/O-device . Fetching data contributes to majority of the system power consumption. Hence reducing the size of activations is essential for more efficient utilization of the available computational resources. There have been a few solutions using lower precision representation for activations, however they necessitate specialized hardware for efficient implementation.

Today, DNNs are almost exclusively trained on one or many very fast and power-hungry Graphic Processing Units (GPUs). GPUs is very efficient for processing neural network as compare to CPUs because they have hundreds of simpler cores, thousands of hardware concurrent threads and have maximize floating point throughput. GPUs has been proven better option for Deep learning neural networks. DNNs achieve higher accuracy by building more powerful models consisting of greater number of layers. The greater compute requirements make DNNs harder to deploy, which has led to a lot of interest recently in specialized hardware solutions. As per study, Low-precision convolutional networks claim to reduce compute requirements of the networks.

The rest of the paper is organized as follows, Section 2 discusses the implementation of our model on Object detection and quantization of weights to lower the precision . Section 3 discusses the evaluation results of the experiments we ran on both CPU and GPU on two real time datasets under low precision . This is followed by Section 4, which includes related work, summary of our implications on the model, and future research directions.

3. Model Description:

To learn about thousands of objects from millions of images, we need a model with a large learning capacity. However, the immense complexity of the object recognition task means that this problem cannot be specified even by a dataset as large as ImageNet, so our model should also have lots of prior knowledge to compensate for all the data we don't have. Convolutional neural networks (CNNs) constitute one such class of models. Their capacity can be controlled by varying their depth and breadth, and they also make strong and mostly correct assumptions about the nature of images (namely, stationarity of statistics and locality of pixel dependencies). Thus, compared to standard feedforward neural networks with similarly-sized layers, CNNs have much fewer connections and parameters and so they are easier to train, while their theoretically-best performance is likely to be only slightly worse. Despite the attractive qualities of CNNs, and despite the relative efficiency of their local

architecture, they have still been prohibitively expensive to apply in large scale to high-resolution images. Luckily, current GPUs, paired with a highly-optimized implementation of 2D convolution, are powerful enough to facilitate the training of interestingly-large CNNs, and recent datasets such as CIFAR-10 contain enough labeled examples to train such models without severe overfitting.

When modern neural networks were being developed, the biggest challenge was getting them to work at all. That meant that accuracy and speed during training were the top priorities. Using floating point arithmetic was the easiest way to preserve accuracy, and GPUs were well-equipped to accelerate those calculations, so it's natural that not much attention was paid to other numerical formats. These days, we actually have a lot of models being deployed in commercial applications. The computation demands of training grow with the number of researchers, but the cycles needed for inference expand in proportion to users. That means pure inference efficiency has become a burning issue for a lot of teams. That is where quantization comes in. It's an umbrella term that covers a lot of different techniques to store numbers and perform calculations on them in more compact formats than 32-bit floating point. Neural network models can take up a lot of space on disk, with the original AlexNet being over 200 MB in float format for example. Almost all of that size is taken up with the weights for the neural connections, since there are often many millions of these in a single model. Because they're all slightly different floating point numbers, simple compression formats like zip don't compress them well. They are arranged in large layers though, and within each layer the weights tend to be normally distributed within a certain range.

Neural networks have been known for decades to contain the expressive power to approximate arbitrary real valued functions . To run neural networks on real hardware, we must represent real network weights and activations in a finite way, amenable to efficient computation and storage. They are typically implemented using floating point. Floating point directly corresponds to real numbers and handles large dynamic ranges without conceptual complexity. However, simplicity and elegance come at a cost. Large floating point matrix multiplies are more complex and less efficient in hardware than analogous integer operations. Companies using Neural networks at scale have been exploring ways to use quantization to improve inference efficiency.

The simplest motivation for quantization is to reduce the computational resources you need to do the inference calculations, by running them entirely with eight-bit inputs and outputs. This is a lot more difficult since it requires changes everywhere you do calculations, but offers a lot of potential rewards. Fetching eight-bit values only requires 25% of the memory bandwidth of floats, so you'll make much better use of caches and avoid bottlenecking on RAM access. You can also typically use SIMD operations that do many more operations per clock cycle. In some case you'll have a DSP chip available that can accelerate eight-bit calculations too, which can offer a lot of advantages.Moving calculations over to eight bit will help you run your models faster, and use less power (which is especially important on mobile devices). It also opens the door to a lot of embedded systems that can't run floating point code efficiently, so it can enable a lot of applications in the IoT world.

Another reason for quantization to shrink file sizes by storing the min and max for each layer, and then compressing each float value to an eight-bit integer representing the closest real number in a linear set of 256 within the range. For example with the -3.0 to 6.0 range, a 0 byte would represent -3.0, a 255 would stand for 6.0, and 128 would represent about 1.5. I'll go into the exact calculations later, since there's some subtleties, but this means you can get the benefit of a file on disk that's shrunk by 75%, and then convert back to float after loading so that your existing floating-point code can work without any changes.

We have implemented quantization by writing equivalent eight-bit versions of operations that are commonly used during inference. These include convolution, matrix multiplication, activation functions, pooling operations and concatenation. The conversion script first replaces all the individual ops it knows about with quantized

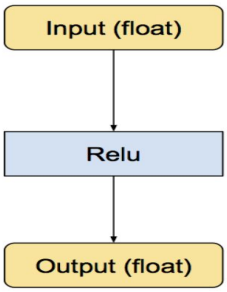

Fig. 1

equivalents. These are small sub-graphs that have conversion functions before and after to move the data between float and eight-bit as shown in Fig.1 .Then, this is the equivalent converted subgraph, still with float inputs and outputs, but with internal conversions so the calculations are done in eight bit as shown in Fig. 2 .The min and max operations actually look at the values in the input float tensor, and then feeds them into the Dequantize operation that converts the tensor into eight-bits. There are more details on how the quantized representation works later on. Once the individual operations have been converted, the next stage is to remove unnecessary conversions to and from float. If there are consecutive sequences of operations that all have float equivalents, then there will be a lot of adjacent Dequantize/Quantize ops. This stage spots that pattern, recognizes that they cancel each other out, and removes them as shown in Fig. 3. Applied on a large scale to models where all of the operations have quantized equivalents, this gives a graph where all of the tensor calculations are done in eight bit.

4. Evaluation:

This section answer the following questions:

1. How does the model perform under different precisions (e.g. 64 bit, 32 bit, 16 bit and 8 bit)?
2. How does the model perform on two different datasets (We have considered the MNIST and the Cifar10 datasets)?
3. How much performance improvement can be achieved by quantization with lower precision?

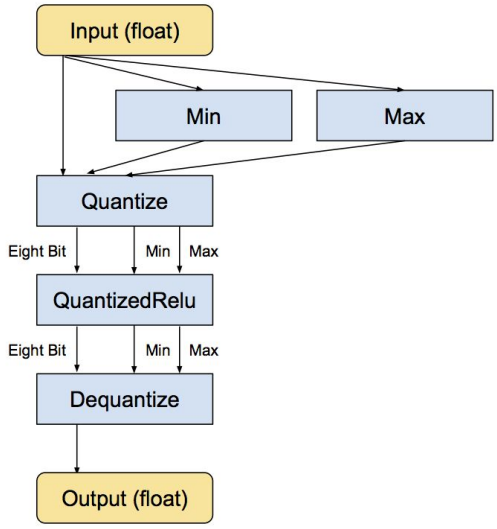

Fig. 2

We start with briefing over the dataset and the setup for the object detection experiment. We then progress through the section and compare the results to answer the above questions.

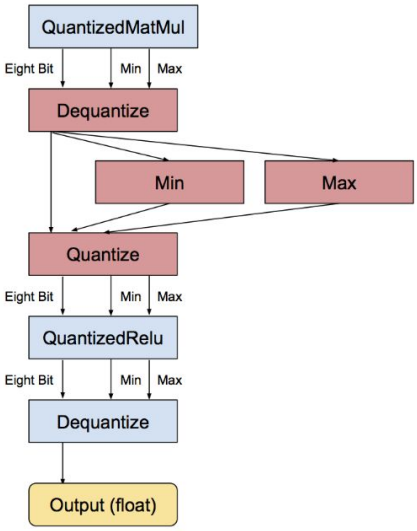

Fig. 3

4.1 Experimental Setup:

This section explains the initial settings of the CNN model and the datasets used to test the model performance.

4.1.1 Datasets:

We have used the below datasets to train and test the CNN model and check the performances.

MNIST: The MNIST[1] database of handwritten digits, 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.

CIFAR10: The CIFAR-10[2] dataset we have used of 35000 32x32 colour images in 2 classes (cats and dogs), with 12500 images per class. There are 25000 training images and 10000 test images.

4.1.2 Packages and Model:

Keras: Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. This is one of the best packages available online for the training of neural networks and allows fast experimentation.

We have used Convolution 2D to deal with images as they are in 2D space. We have also used MaxPooling2D for pooling purposes and used Flatten for flattening in which we convert the feature maps, that we have obtained through Convolution and Pooling, to large feature vector.

CNN parameters:

32 - Number of feature detectors

3 - Rows of Feature Detectors

3 - Columns of Feature Detectors

Input shape - To ensure all the input images are of the same format. We have taken the input matrix as 64x64x3 where, 64,64 represents the image matrix in number of pixels and 3 represents that it is a colour image. We set the last parameter of input_shape to 1 if we are dealing with a black and white images.

Activation function - Relu

pool_size - This is set to [2,2] to reduce the size of the feature maps by 2.

Flatten() - This function converts the pooling output to one dimensional matrix. This matrix will contain pixel pattern of the input image.

The single matrix from Full Connection is fed into an conventional neural network (a fully connected NN).

Dense() - Function to create 128 nodes for the hidden layers and creating a single output node.

Number of epochs = 25, So training of 25000 images will happen over 25 epochs.

one epoch = one forward pass and one backward pass of all the training examples

batch size = The number of training examples in one forward/backward pass is set to 32.

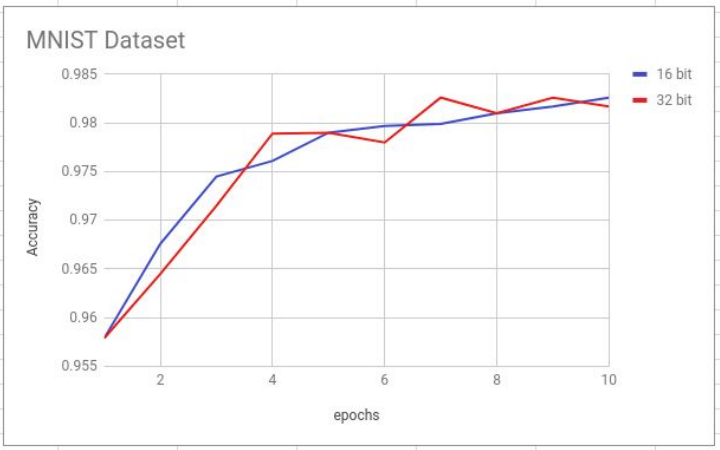

Fig. 4

4.2 Model Performance Comparison:

The base CNN model structure set up using the above parameters was trained against the aforementioned two datasets and the results are as shown in the graphs Fig. 4 and Fig.5.

The accuracy in the initial epochs is lower and the model then builds on the already trained settings, thus increasing the accuracy towards the final epochs.

As seen in Fig. 4, the maximum accuracy achieved for the MNIST dataset is 0.983 for 16 bit which is almost same as for 32 bit. The model accuracy increases more in the initial epochs and we then see a steady increase in the accuracy.

While on the Cifar10 dataset, the maximum accuracy achieved is around 0.79 for 16 bit as well as 32 bit precision. The accuracy increases steadily over the epochs for this dataset as can be seen in Fig. 5

Fig. 5

4.3 Quantization:

Deep networks can be trained with floating point precision, a quantization algorithm can be applied to obtain smaller models and speed up the inference phase reducing memory requirements. Fixed-point compute units are typically faster and consume far less hardware resources and power than floating-point engines. Low-precision data representation reduces the memory footprint, enabling larger models to fit within the given memory capacity.

We tested the Quantized model for 8 bit precision and below are the results:

Fig. 6 represents the accuracy for the Cifar10 dataset following three runs using original model and Quantized model. It can be seen that the Quantized model accuracy is almost similar to the original model accuracy with smaller model and faster inference phase.

We similarly compared the two models for MNIST dataset and it can be seen that the accuracies are almost equal.

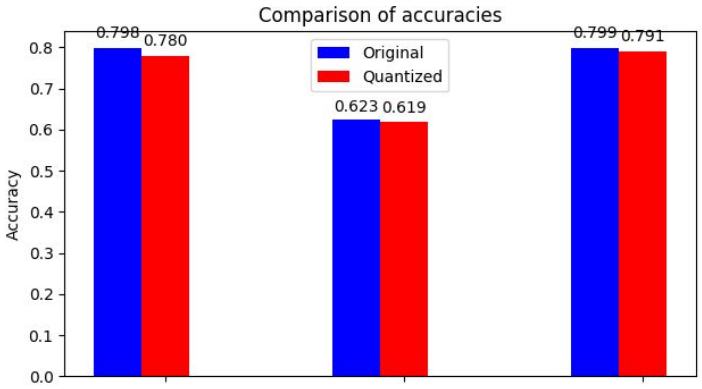

Fig. 6

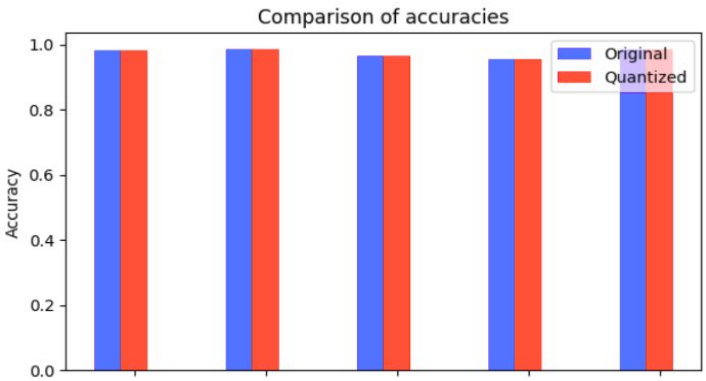

Fig. 7

5. Related Work:

Deep Neural Network(DNN) are typically over-parameterized in that there is significant redundancy in the learning model. On the one hand, the sparsity of the learnt representations offer DNN the high algorithmic performance, but at the same time, it is easy to arrive at a DNN that is a waste in both computation and memory. Recent research efforts have started to address DNN size and complexity. Some of these efforts are motivated from the need to reduce training time. One basic approach to low precision deals with approximation and quantization techniques. Vanhoucke

et al converted 32-bit floating point activations to 8-bit integer fixed point implementations. Gong et al compressed DNN weights using vector quantization. Gupta et al explore stochastic rounding techniques to implement DNN from 32-bit floating point to 16-bit fixed point integers on FPGA. Courbariaux et al shows a training method using binary weights to avoid the forward and backward propagation multiplications. Until recently, the use of extremely low-precision networks (binary in the extreme case) was believed to substantially degrade the network performance (Courbariaux et al., 2014). Soudry et al. (2014) and Cheng et al. (2015) proved the contrary by showing that good performance could be achieved even if all neurons and weights are binarized to ±1 . 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. However, our work is focussed on lowering the precision further more and evaluating the performance of the DNN at different scenarios.

Lin et al. (2015) carried over the work of Courbariaux et al. (2015) to the back-propagation process by quantizing the representations at each layer of the network, to convert some of the remaining multiplications into binary shifts by restricting the neurons values of power-of-two integers. Lin et al. (2015)'s work and ours seem to share similar characteristics . However, their approach continues to use full precision weights during the test phase. Moreover, Lin et al. (2015) quantize the neurons only during the back propagation process, and not during forward propagation.

Other approaches to handle complexity of deep networks include: (a) leveraging high complexity networks to boost performance of low complexity networks, as proposed in Hinton et al. (2014), (b) compressing neural networks using hashing (Chen et al., 2015), and (c) combining pruning and quantization during training to reduce the model size without affecting the accuracy (Han et al., 2015). These methods are complementary to our proposed approach and the resulting networks with reduced complexity can be easily converted to fixed point using our proposed method. In fact, the DCN model that we performed experiments with and report results on, (see Section 5.2), was trained under the dark knowledge framework by using the inception network (Ioffe & Szegedy, 2015) trained on ImageNet as the master network.

6. Conclusion:

In this work, we considered the efficiency of training high performance deep networks with low-precision. We've found that we can get extremely good performance on mobile and embedded devices by using eight-bit arithmetic rather than floating-point. You can see the framework we use to optimize matrix multiplications at gemmlowp. We still need to apply all the lessons we've learned to the TensorFlow ops to get maximum performance on the neural network models we have developed. The quantized implementation is a reasonably fast and accurate reference implementation that we're hoping will enable wider support for other higher and lower precision as high as to 128 bits and as low as 1bit. Our results have shown the impact of quantization in terms of performance of speedness and their aid in Shrinking the file size for the faster computeness of the matrix multiplications involved in the working. We also hope that this demonstration will encourage the community to explore what's possible with low-precision neural networks. The network that results from quantization technique proves to significantly outperform previous efforts at deep learning with low precision, substantially reducing the gap between the low-precision and full-precision various state of-the-art networks. These promising experimental results suggest that the Quantization can be very useful for the deployment of state-of-the-art neural networks in real world applications.

References:

[1] http://yann.lecun.com/exdb/mnist/

[2] https://www.cs.toronto.edu/~kriz/cifar.html

[3] https://github.com/EmilianoGagliardiEmanueleGhelfi/CNN-compression-performance

[4] https://github.com/bazelbuild/bazel

[5] https://www.tensorflow.org/

[6]
https://www.tensorflow.org/performance/quantization
[7]
http://www.wolfib.com/Image-Recognition-Intro-Part-1/
[8]
http://blog.stratospark.com/creating-a-deep-learning-ios-app-with-keras-and-tensorflow.html#creating-a-deep-learning-ios-app-with-keras-and-tensorflow
[9]
https://github.com/google/gemmlowp/blob/master/doc/low-precision.md
[10]
https://renatocunha.com/blog/2017/03/dlnd-image-classification/

Appendix

Deep Learning in Object Detection project has involved the studies of Deep Neural Network and how we can use this concept for Object detection on a Image. Given are the participation details of the project: All work have been done collaboratively, However we have mentioned the contributions made.

Prasanna Shanmuganathan - MCS at Illinoi Tech-A20378683
Contribution:

- Read a lot of research papers to understand the need for lowering the precision of data type on achieving high efficiency in deep learning neural network.
- Worked on building the neural network model in Keras and testing the model on 16-bit and 32-bit precision
- Reported the need for an alternate model to lower the precision further to 8-bit which demanded more of a reading from research papers and analysing different methodologies. However there were downsides with the various methods I came up with.
- Equal contributions on the write ups - Proposal Mid Term report and Final report.
- Worked on building the model using quantization technique as proposed in the implementation.(The idea of Quantization was proposed by my partner and I contributed on the implementation of the model).

Shringa Bais - MCS at Illinois Tech - A20382937

Contribution:

- Read a lot of research papers to understand the need for lowering the precision of data type on achieving high efficiency in deep learning neural network.
- Worked on building the neural network model in Keras and testing the model on 16-bit and 32-bit precision
- Through my reading, I was able to come up with the idea of Quantization which became the most empirical solution to lower down the datatype to different precisions.
- Ran the Code on GPU to test the results of the system on a high computing environment.
- Equal contributions on the write ups - Proposal Mid Term report and Final report.