Quantization of Deep Neural Networks for Large-scale Connectomics

Jonathan Hu

Harvard University

Abstract. The field of connectomics aims to solve all connections between neurons to shed light on the understanding of animal brains. Since manual annotation is super time-consuming and requires expertise, deep learning approaches have become the de facto methodology and significantly speed up the annotation. However, with exponentially increasing data scale and model size, the time and memory efficiency of existing models also need to be improved. In this reading and research project, we first discuss the big-data challenges of connectomics and then focus on the model quantization technique that uses low-precision or integer weights to alleviate the computational demands. Our implementation of post-training static quantization on a ResNet18 model shows significantly reduced inference time and memory usage, with only a marginal decrease in terms of accuracy. We will apply the technique to heavier 3D connectomics segmentation models in the future.

Keywords: Deep Learning · Quantization · Connectomics · PyTorch

1 Introduction

The field of connectomics endeavors to provide a more complete rendering of the neural circuit synaptic connectivity [1]. The applications are numerous. With a complete map of the human brain, "comparisons between healthy and diseased brains might point to the physical underpinnings of psychiatric diseases"[1]. Another exciting application of a connectional map is the potential ability to understand how the network changes as humans age [1].

Assuming that connectomics has a future, there are three main technical obstacles: faster image acquisition, faster image processing, and better statistical and analytic tools [1]. In their paper "The big data challenges of connectomics" published in *Nature* in 2014, Lichtman et al. contextualize the scale of the big data challenges surrounding connectomics. For instance, at 2014 computing and image acquisition speeds, they estimate that it would take 1,000 years to map the cerebral cortex of a rat, and over 10 million years to complete a map of every synapse in the human brain [1, 2].

Since then, developments in electron microscopy technology and an automated tissue slicing set-up developed in the Lichtman Lab have substantially sped up the image acquisition process. For instance, acquiring the 3D image of 1mm^3 of brain tissue used to take 6 years, but with the acquisition of a new imaging microscope from Carl Zeiss, imaging the same 1mm^3 only takes one month now [2].

On the image processing side, developments in deep learning model optimizations techniques, particularly quantization, have reduced the computational demands of deep neural networks with a marginal impact on accuracy, making their use in computing-resource-constrained environments much more attractive.

2 Big Data Challenges of Connectomics

Right now, the main bottleneck in connectomics is image processing. Indeed, Professor Lichtman et al. outline that "a number of analytic problems stand between the raw acquired digitized images and having access to this data in a useful form" [2].

The first problem in image processing is accuracy [1]. Processing brain tissue images is much more challenging than processing images of other types of human tissues. The main factors that make processing brain tissue images challenging are as follows: the number of types of cells (i.e. 50 in the retina alone vs. 5 in the liver), the complicated geometry of nervous tissue, the directionality of neural circuits, the fine structure of neural circuits with no structural redundancies like in the renal system, and how the cellular structure of neural tissue is a product of both genetic instruction and experience [1]. These are the main reasons why even the best computer vision models struggle with accuracy in the task of image segmentation.[1]

The second problem in image processing is scale. The datasets used in connectomics are some of the largest in the world of big data. According to Lichtman et al., "acquiring images of **a single cubic millimeter** of a rat cortex will generate about 2 million gigabytes or 2 petabytes of data" [2]. The volume of a rat cortex is approximately 500 mm^3, which would require 1,000 petabytes of data [2]. To put one petabyte in context, the complete database system of Walmart is a few petabytes [2]. And a complete human cortex is 1000 times larger than the cortex of a rat, and would require a zetabyte which is about the total amount of recorded data in the entire world back in 2014 [2]. Given that CPU and GPU computing is cost and energy-intensive, just processing this amount of data is a huge challenge, even without considering the accuracy of the output.

To summarize, Lichtman et al. explain that the "greatest challenge facing connectomics at present is to obtain saturated reconstructions of very large (for example, 1 mm^3) brain volumes in a fully automatic way, with minimal errors, and in a reasonably short amount of time" [2]. Indeed, connectomics faces challenges on three fronts: automation, scale, and accuracy. The challenge on each of these fronts is daunting.

[1] See Appendix B for an example of image segmentation.

3 Applying Deep Neural Networks to Connectomics

In connectomics, the required combination of accuracy and speed makes the image processing problem particularly challenging. If lower accuracy was acceptable, then we could make a trade-off and employ less accurate, but more computationally efficient computer vision models. However, accuracy matters because building a useful connectional map requires that each cell in each cross-section of the brain tissues be identified correctly to build the 3D map of the neural circuit network.

Deep neural networks (DNNs) attempt to address the image processing challenges of connectomics. More specifically, computer vision and deep learning research aims to address these challenges by fully automating the process of annotating each layer of brain tissue and by working to match the accuracy of professional biomedical annotators, thereby addressing the problems of speed and accuracy.

Although deep neural networks have made many advances in accuracy and delivered improvements over previous computer vision models, Lichtman et al. observe that deep neural networks "require on the order of 1 million instructions per pixel and therefore cannot be applied to the workflow as described. Moreover, the results of these programs are still inadequate because they require human proofreading to correct mistakes"[2].

Indeed, the accuracy improvements from using a DNN might not justify the cost of additional computing power. A typical $1M computing system, consisting of 500 4-core 3.6GHz processors equipped with 16GB of RAM per processor, can match the image acquisition speed of the Lichtman's lab set-up with 10,000 computational instructions per image pixel, which is far below the 1M instructions per pixel required to run a deep neural network [2].

Thus, there are many research efforts to make deep learning models more efficient in their memory usage and their computation speed. Memory usage is important because GPUs are expensive. Computation speed is important because computation speed is directly proportional to the number of instructions needed per image pixel. So for instance, a 2x speedup in inference speed would be the equivalent of reducing the required computing resources from 1M to 0.5M instructions per pixel.

In the next sections, we will briefly discuss the current research efforts to optimize deep learning models and do a deep dive into model quantization.

4 Optimization Techniques for Deep Learning Models

One direction of research in the field of computer vision is the pursuit of human-like accuracy in tasks such as image segmentation and object classification. In the world of deep neural networks, high accuracy is usually accompanied by high complexity, and high complexity means high memory and computing usage. Indeed, Bianco et al. (2018) benchmark all the major deep neural networks for image recognition and find that as model accuracy increases, the computing power required and model complexity increase (see figure 1) [3].

Thus, given that higher accuracy implies higher complexity and higher computing power, a parallel direction of research is focused on making current models computationally more efficient without sacrificing accuracy.

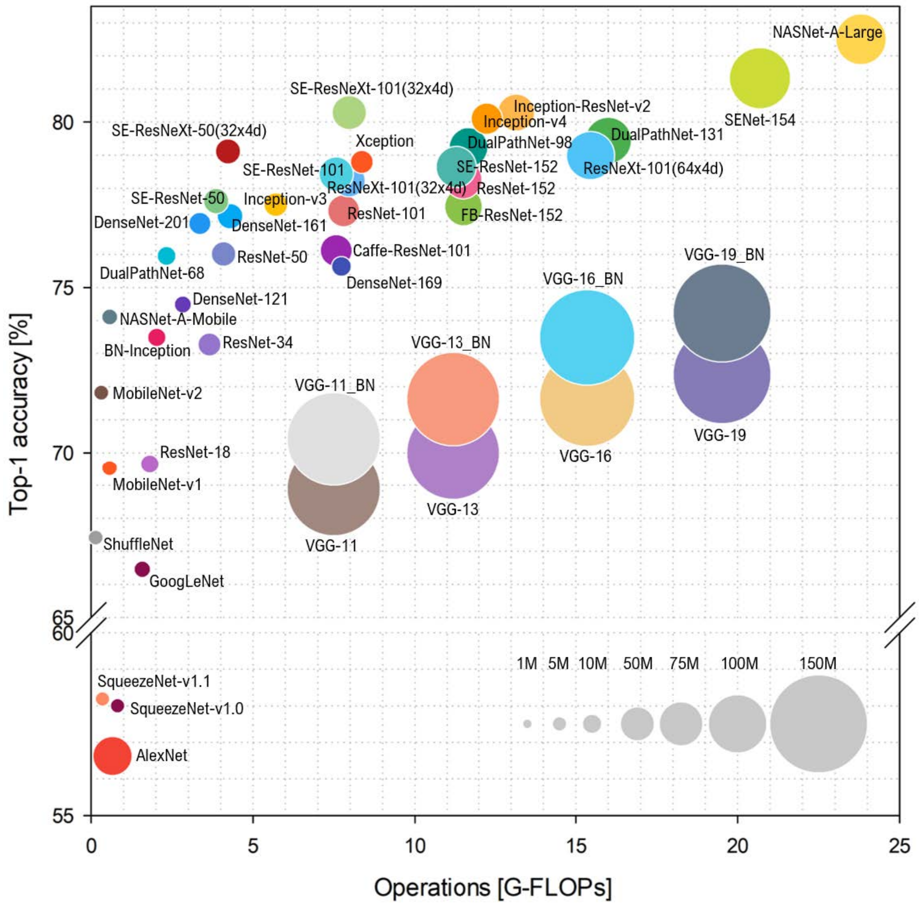

Fig. 1. Accuracy vs. Computational Complexity. G-FLOPs required for a single forward pass. Size of ball corresponds to model complexity [3].

Currently, both academics and the research divisions of technology companies are simultaneously working on different fronts to optimize deep learning models. In a seminar on model optimization, Chris Gottbrath, Group Technical Program Manager at Facebook, outlines the current directions of optimization research: better hardware accelerators (i.e. TPUs), optimized kernels (i.e. Cudnn, Intel MKL-DNN), efficient deep network architectures (i.e. Nasnet, Mobilenet, FBNet), and optimization techniques that do not require a redesign of the model architecture or new hardware [4].

In the past decade, there have been many new developments in the area of optimization techniques that do not require new architectures or hardware. Two such techniques are pruning and quantization.

Han et al. (2016) combine pruning, quantization, and Huffman coding to compress a VGG-16 model by 49x without loss of accuracy and with a 4x faster inference speed. They find that the accuracy of the network drops when the quantization process alone or the pruning process alone compresses the network by more than 8% the size of the original model. But when pruning and quantization are combined, they were able to compress the network to 3% of the original size before losing accuracy [5].

More recently, Wang et al. (2021) exploit a variant of L2 regularization that grows over time to serve as a mechanism for pruning the network during the training process. Their method of pruning via growing regularization is simple to implement and scalable to large datasets. Pruning 50% of the weights of the ResNet56 + CIFAR10 model resulted in a speed-up of 1.99x and a memory reduction of 2x, with accuracy falling from 93.36% to 93.06% [6].

5 Quantization

5.1 What is Quantization?

The process of quantizing a model refers to the process of converting the weights and activations of a deep neural network from 32-bit floats (FP32) to 8-bit integers (INT8). The idea is that reducing the precision of the weights and activations will reduce the model size in memory (also known as model complexity) and will speed up the computation during inference [7]. Intuitively, quantization can be thought of as building an approximation of the original model by replacing the FP32 weights with INT8 weights.

Quantizing a model can potentially result in a 4x reduction in model size, 2-4x reduction in memory bandwidth and 2-4x faster inference; the faster inference is due to the smaller required memory bandwidth and faster compute with INT8 arithmetic [7].

5.2 The Different Types of Quantization

There are three types of quantization: dynamic quantization, post-training static quantization, and quantization aware training (QAT).

Dynamic Quantization. Dynamic quantization consists of converting the weights to INT8s. All the activations get dynamically converted to INT8 at runtime when computing the forward pass of inference; however, the activations are still read and written to memory as FP32s [7]. This optimization technique works best for dynamic models such as LSTMs, MLPs and Transformers and results in good accuracy [4]. Dynamic quantization is the easiest to implement out of the three in PyTorch with a one-line API call to torch.quantization.quantize_dynamic(model, args).

Post-Training Static Quantization. In contrast to dynamic quantization which makes use INT8 arithmetic only, statically quantizing a deep neural network after it has been trained allows the network to make use of INT8 arithmetic **and** memory access [7]. This is because for static quantization, the FP32 activations are not dynamically converted to INT8s; instead, the activation modules are quantized prior to inference.

Quantizing the activation modules requires a calibration process where we compute the distribution of the activations in each layer in advance. This is done by placing observer modules on top of the activation modules, feeding the deep neural network batches of training data, and observing the resulting statistical distribution of activation values [7]. Then, we create a map from the observed distribution to the range of 8-bit integers. The most common method for mapping FP32 values to INT8 values is uniform quantization, where we divide the range of activation values into 256 equally sized intervals. By default, PyTorch implements the uniform quantization method, but it supports more sophisticated methods [7].

Quantization works best for CNNs and provides the best performance for static models [4]. It is important to emphasize the post-training aspect of this optimization technique, meaning that when the FP32s weights are being converted to INT8s, the weights are already the optimal weights learned after performing stochastic gradient descent.

Static Quantization Aware Training. QAT results in the highest accuracy of the three types of quantization. During training, all the weights and activations are computed and accessed as FP32s, but they are rounded to mimic INT8 values. This technique works best for CNNS and models that are to be deployed on mobile devices [4].

5.3 Mathematical Underpinning of Uniform Quantization

As mentioned above, post-training static quantization relies on the uniform quantization method, which provides a map between the entire range of FP32 activations to a set of 256 integers.

Formula for Quantizing x_f within $[-L, L]$ using b bits [8]. Let
L: Boundary of the quantization interval
sf: Scale factor
b: Bitwidth
x_f: Floating-point input
x_c: Clamped version of x_f
$\lfloor . \rfloor$: Floor function
x_q: Quantized x_f

Step 1 :

$$sf = \frac{L}{2^{b-1} - 1}$$

Step 2 :

$$x_c = clamp(x_f, -L, L)$$

Step 3 :

$$x_q = \lfloor \frac{x_c}{sf} + 0.5 \rfloor sf$$

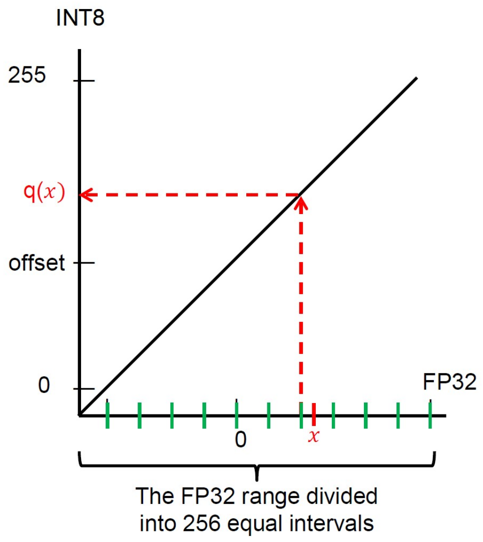

Fig. 2. A Visual Representation of Uniform Quantization [8].

6 Experiments

My independent research focuses on implementing post-training static quantization for a ResNet18 model trained on the CIFAR-10 dataset.[2]
Model: an untrained ResNet18 model from the Torchvision library [9].
Dataset: CIFAR-10. 60,000 32x32 colour images in 10 classes. 6000 images/class. 50,000/10,000 training-test split. Each image has been labeled by a human [10].
Training: 200 Epochs. Helper functions and code provided by Kuang Liu [11].
Computing Set-up: Harvard Research High-Performance Cluster. Linux Bash Shell. 2 CPUs. 1 GPU. 20GB of memory.

[2] See Appendix A for implementation details

7 Results and Discussion

Table 1. Non-Quantized vs. Quantized ResNet18

ResNet	**Inference Speed**	**Model Complexity**	**Accuracy**
Non-Quantized	24.683 s	46.84 MB	88.540 %
Quantized	13.907 s	11.84 MB	88.450 %

The inference speed and accuracy were calculated by running the model on the CIFAR-10 testing set of 10,000 images. The model was moved from CUDA to CPU for inference.

Applying post-training static quantization to the ResNet18 model resulted in a 1.77x increase in inference speed and a 3.96x reduction in model size, with negligible impact on the accuracy.

The baseline accuracy of the ResNet18 trained in this experiment is lower than the maximum achieved accuracy in the literature of approximately 95%. More time could have been spent optimizing the hyperparameters, but the point of this experiment is not to achieve the highest accuracy, but rather to gather empirical data on the impacts of quantization on inference speed, on model complexity, and on model accuracy.

From these experiments, post-training static quantization shows a lot of promise for significantly reducing model complexity and increasing inference speed while resulting in little to no accuracy loss.

Current Limitations of Implementing Quantization in Pytorch. At the moment, PyTorch only supports model quantization on CPUs because the quantization module in PyTorch does not provide quantized operator implementations on CUDA.

There are a few implications. First, this means that once the model is trained, the model must be moved from the GPU to the CPU before model quantization can occur. This also means that inference can only happen on the CPU, which is much slower than inference on the GPU. For instance, for the non-quantized ResNet18, the inference speed was 3.84 seconds on the GPU compared to 24.683 seconds on the CPU. Even after quantization, the inference speed of the non-quantized model on GPU is still 3.63 times faster than the quantized model on CPU.

8 Conclusion

Applying post-training static quantization to the ResNet18 model resulted in a 1.77x increase in inference speed and a 3.96x reduction in model size, with negligible impact on the accuracy. One key limitation in PyTorch is that it only supports quantization on CPU. Thus, until PyTorch provides support for quantization on CUDA, the quantization feature on PyTorch will have limited use in real-world applications where inference speed matters such as image segmentation in connectomics.

9 Acknowledgement

I would like to thank Prof. Hanspeter Pfister, Zudi Lin, and Donglai Wei for their support, teaching and guidance in my exploration of connectomics, deep learning, cluster computing and model optimization this semester!

10 Appendix A: Implementation Details

10.1 Environment

1. Connect to the Harvard High Performance Research Computing via SSH.
2. Start an interactive session with 2 CPUs, 1 GPU and 20GB of memory by running: srun –pty -p cox –gres=gpu:1 -n 2 -N 1 –mem 20000 -t 12:00:00 /bin/bash
3. Load the CUDA module by running: module load cuda
4. Activate a Conda enviroment with PyTorch

10.2 Execution

1. To run the experiment code on the cluster, clone the following GitHub repository: https://github.com/jonyhu/am91r.git.
2. Change the directory to pytorch-cifar-quantized-static-predefined-200-newday
3. Run: python main.py

11 Appendix B

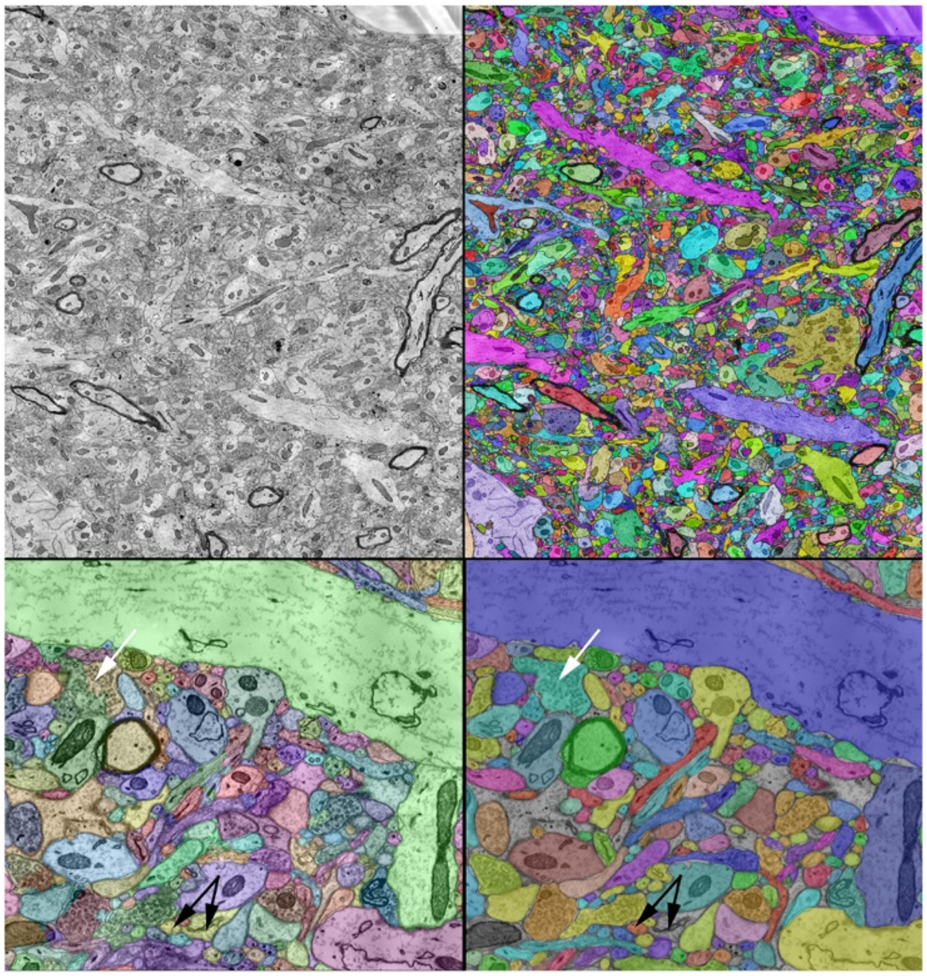

Fig. 3. Image Segmentation of Brain Images [2].

References

1. Morgan, Joshua L., and Jeff W. Lichtman. “Why Not Connectomics?” Nature Methods 10, no. 6 (June 2013): 494–500. https://doi.org/10.1038/nmeth.2480.
2. Lichtman, Jeff W, Hanspeter Pfister, and Nir Shavit. “The Big Data Challenges of Connectomics.” Nature Neuroscience 17, no. 11 (November 2014): 1448–54. https://doi.org/10.1038/nn.3837.
3. Bianco, Simone, Remi Cadene, Luigi Celona, and Paolo Napoletano. “Benchmark Analysis of Representative Deep Neural Network Architectures.” IEEE Access 6 (2018): 64270–77. https://doi.org/10.1109/ACCESS.2018.2877890.
4. PyTorch. Deep Dive on PyTorch Quantization - Chris Gottbrath, 2020. https://www.youtube.com/watch?v=c3MT2qV5f9w.
5. Han, Song, Huizi Mao, and William J. Dally. “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding.” ArXiv:1510.00149 [Cs], February 15, 2016. http://arxiv.org/abs/1510.00149.
6. Wang, Huan, Can Qin, Yulun Zhang, and Yun Fu. “Neural Pruning via Growing Regularization.” ArXiv:2012.09243 [Cs], April 5, 2021. http://arxiv.org/abs/2012.09243.
7. Krishnamoorthi, Raghuraman, James Reed, Min Ni, Chris Gottbrath, and Seth Weidman. “Introduction to Quantization on PyTorch.” Accessed May 6, 2021. https://www.pytorch.org.
8. Kung, HT. “Harvard CS242 Computing at Scale.” Lecture Slides. February 17, 2021.
9. GitHub. “Pytorch/Vision.” Accessed May 10, 2021. https://github.com/pytorch/vision.
10. “CIFAR-10 and CIFAR-100 Datasets.” Accessed April 22, 2021. https://www.cs.toronto.edu/ kriz/cifar.html.
11. Liu, Kuang. Kuangliu/Pytorch-Cifar. Python, 2021. https://github.com/kuangliu/pytorch-cifar.