Wet 2: Model Optimizations with PyTorch

EE 046853, Advanced Computer Architectures, Winter 2022/23
Technion — Israel Institute of Technology

1 Introduction

In this wet exerecise, you will learn some basic PyTorch functionalities, and implement simple optimizations such as quantization and pruning, which are highly relevant for hardware. Quantization reduces the parameters' precision. Quantized models, therefore, can be deployed on relatively cheap hardware (e.g., operations in INT8 instead of FP32). Pruning reduces the memory footprint and MAC operations by removing parameters. Both optimizations can be implemented in many different ways. In this assignment, you will implement a post-training symmetric quantization and a magnitude-based weight pruning without model fine-tuning.

1.1 Getting Started

Besides using PyTorch, you are more than welcomed to work in any environment you'd like. Fortunately, you don't have to install anything or SSH to a remote server, you can just use Google Colab. We prepared a template for you to use throughout this assignment, so you can focus on the more important things. Having said that, things are not perfect and you may need to add additional pieces of code beyond what's currently written.

1. Go to the Google Colab file.
2. Select *File* → *Save a copy in Drive.*
3. Select *Runtime* → *Change runtime type*, and then select *GPU* under the *Hardware accelerator* drop-down menu.

2 Home Assignments

2.1 Instructions

1. Final submission deadline: 30/01/2023.
2. Submissions are in *couples.*
3. Please submit your report and your colab file to the appropriate homework submission box in Moodle as a zip file.
4. Clearly explain your observations and insights. When presenting a graph, add axes labels and units. Feel free to ask questions in the appropriate Moodle forum. We encourage you to discuss all course material with your peers. However, the final solution must be yours, that is, a direct collaboration on assignments is strictly prohibited.

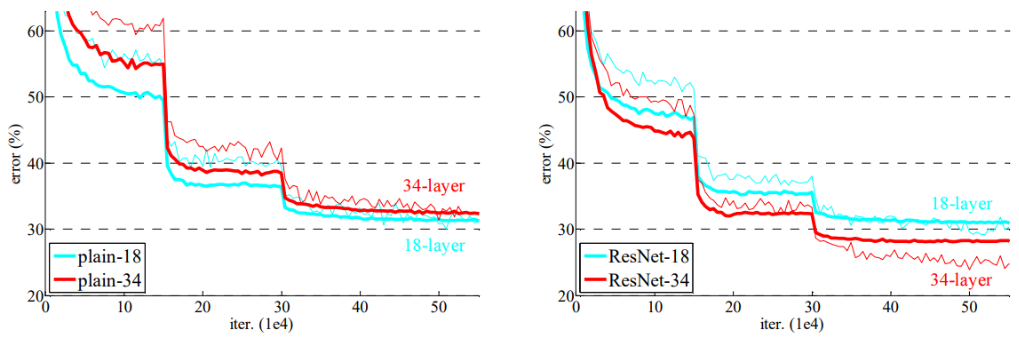

Figure 4. Training on **ImageNet**. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

Figure 1: From He et al., "Deep Residual Learning for Image Recognition", CVPR, 2016.

2.2 Questions

2.2.1 Getting Started with PyTorch

1. Modify the code to support the CIFAR-100 dataset. Obviously, you will have to load CIFAR-100 instead of CIFAR-10, buy you will also need to modify the network. Report your modifications.

2. Change the CNN structure to comply with the following rules, with your id numbers as id_1 and id_2:

   - conv layer 1 kernel size:
   $$\max_i\{id_1[i]\}$$
   - conv layer 2 kernel size:
   $$9 - \min_i\{id_2[i]\}$$
   - number of training epoches:
   $$\max\{\max_i\{id_1[i] * id_2[i]\}, 30\}$$

3. Report the following:

   (a) Your model architecture (do not copy-paste the code, but a table would be great).

   (b) The memory footprint during inference at each layer (activations and weights).

   (c) The memory footprint during backpropagation (activations and weights).

   (d) The amount of computations during inference in each layer.

   (e) A graph of the training and validation errors (on the same graph) as a function of the iteration (Figure 2, for example).

4. Plot histograms of the weights of every convolutional and fully-connected layer. Record any observation you make about the distribution of the values.

2.2.2 Quantization

For simplicity's sake, we will only tackle weight quantization. There are many quantization methods, we will a simple quantization method which works quite well, at least with 8-bit quantization. It is a post-training quantization method, that is, not further training is conducted after quantization.

The quantization is symmetric and uniform, thereby it may be easily implemented in hardware. For B bits quantization, the quantization range is $\left[-2^{B-1}, 2^{B-1}-1\right]$. For example, for 8-bit quantization, the range is $[-2^7, 2^7-1] = [-128, 127]$.

What symmetric uniform quantization does is normalizing the weight range (symmetrically) and then puts each weight into the appropriate bins which are uniform in size. That is, assume a weight tensor

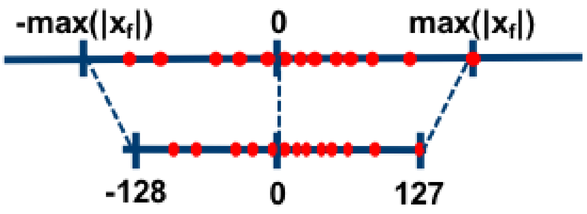

Figure 2: Symmetric quantization example. Figure from here.

W, and without loss in generality, assume $W_{min} < W_{max}$. The scale factor is, therefore, $\Delta = \frac{2 \cdot W_{max}}{2^B - 1}$. Now, for each weight w_i in W, the quantized weight is equal to $w_i^{(q)} = \text{round}\left(\frac{w}{\Delta}\right)$. The "full precision quantized" weight is equal to $w_i^{(q)}\Delta$.

What makes it easy to implement in hardware? Well, the nice thing is that the entire layer can be performed based on INT hardware, and only at the end we need to multiply the *result* with the FP scaling factor.

1. Implement the quantization function. Report the model accuracy as a function of the quantization bits (1,2,3,4,...,16).
2. Implement another quantization function where the quantization is *per-channel*. That is, given weight tensor dimension of (C_{out}, C_{in}, H, W), per-channel quantization means that each C_{out} has a different scale factor. Intuitively, it means that each filter has a different quantization. Again, report the model accuracy as a function of the quantization bits.
3. What is the memory footprint of your model during inference for 2, 4, and 8 bits compared to the FP32 model?

2.2.3 Pruning

There are many pruning methods. Pruning is usually followed by fine-tuning (i.e., training of several epochs). However, for simplicity's sake, we will not perform fine-tuning. Also, we can prune weights as well as activations. We will focus on weights.

How do we choose which weights to prune? again, there are all kind of methods. We assume that the impact of small weights on the model accuracy is negligible. So we (you) will prune weights locally, i.e., per-layer.

1. Implement the pruning function which prunes a percentage of the weights. That is, for example, 20% means that 20% of the weights with the smallest magnitude are pruned (can be set to zero).
2. Report the model accuracy as a function of the pruning percentage.
3. Report the model accuracy as a function of the number of entire model MAC operations.
4. Prune each layer *in isolation* (while the other layers are not pruned). For each layer plot the model accuracy as a function of the number of entire model MAC operations. Do all layers behave the same? what are you observations?
5. Will conventional hardware benefit from this pruning method? explain. If it wont, suggest a way to exploit it (we learned a couple of methods in class).