Lab 1: Quantization

EE 290-2 Hardware for Machine Learning
UC Berkeley, Spring 2021
Instructor: Prof. Sophia Yakun Shao
Teaching Assistant: Abraham Gonzalez

Due: February 5, 2020 @ 11:59 PM PST

Contents

1 Introduction

This lab will teach you how to generate machine learning models which can be run efficiently on limited hardware resources. We will do this with a technique called *post-training quantization*, where we reduce the precision of the weights and inputs that correspond to a model which has already been trained at a high precision. Additionally, we will change the data format of our inputs and weights from expensive *floating-point* numbers to cheap *fixed-point* numbers.

2 Background

Machine learning models are typically trained using 32-bit floating-point data. However, floating-point arithmetic is very expensive (in terms of area, performance, and energy) to implement in hardware. Additionally, 32 bits of precision may be needed during training to capture very small gradient steps, but that much precision is usually unnecessary during inference. Lowering arithmetic precision can save both circuit area and memory bandwidth, helping hardware designers to save costs on several fronts.

Therefore, DSPs and, more recently, ML accelerators typically implement fixed-point arithmetic at much lower precisions. 8-bit unsigned integers are a common target, but some accelerators will go all the way down to single-bit binary arithmetic.

There are two broad approaches to quantization: *post-training quantization*, and *quantization-aware training*. Quantization-aware training can preserve more accuracy, but we explore only post-training quantization in this lab. Interested students can read this paper[1], which provides an accessible introduction to quantization-aware training.

There are two major forms of fixed-point arithmetic which we can target. One uses *unsigned* integers, along with a *zero-point* which describes where the floating-point zero maps to the unsigned integer range. The other approach uses *signed* integers, where the zero-point has essentially been fixed to 0. The approaches are nearly identical, but in this lab, we target signed integers (also called *symmetric quantization*), because it makes some equations a little simpler and easier to understand. Interested students can check out this link[2], which describes the math behind both methods.

With fixed-point arithmetic, we are always faced with the question of what to do with floating-point values that don't perfectly map to any fixed-point number. Floating-point values that are too large, or too low to be expressed, are typically mapped to our maximum and minimum fixed-point values respectively. Floating-point values that are within range are usually scaled to their fixed-point representation and rounded to the nearest integer. But what happens if the fixed-point representation is equally between two adjacent fixed-point numbers? For example, what if the unrounded fixed-point representation is something like 32.5, which could be rounded to either 32 or 33 in an 8-bit integer? In that case, a special *rounding rule* must be determined. In PyTorch, and in most implementations, the rounding rule used will be *round-to-even*, where we round to the nearest even number in the case of a tie (32 in the previous example). This may seem unintuitive, but we do it because it helps us to reduce bias—the mean error is expected to be 0 because numbers have an equal chance of being even or odd. Other rounding rules, such as rounding to the highest number (as is often taught in elementary schools) will instead bias your network to larger magnitudes, so that all your predictions get shifted up closer and closer to your maximum fixed-point numbers. This can gradually destroy your accuracy.

3 Your Assignment

This lab will require a beginner's level of proficiency with PyTorch. If you're unfamiliar with PyTorch then be sure to complete these tutorials first:

[1] "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference" by Jacob et al.
[2] https://nervanasystems.github.io/distiller/algo_quantization.html

- PyTorch Deep Learning Blitz: https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html

Fortunately, you will not be required to install PyTorch on your own laptop, or to ssh into some obscure server to get access to GPUs. Instead, we use Google Colaboratory, which you can think of as Google Docs for IPython notebooks. We have prepared a template Colaboratory notebook which you will edit for this lab; you can set it up with these steps:

- Go to https://colab.research.google.com/drive/12Unx635MbZ_7GoCACRxXIg49hn5SZ3TF?usp=sharing
- Select *File* → *Save a copy in Drive*.
- Select *Runtime* → *Change runtime type*, and then select *GPU* under the *Hardware accelerator* drop-down menu.

Complete the Colaboratory notebook which contains code that will load the CIFAR10 dataset and train a simple convolutional neural network (CNN) to classify it. You will fill out this notebook (questions and code blocks) to quantize sections of the CNN one-at-a-time.

4 Lab Report Structure

Submit a PDF writeup of your responses to the questions from the Colaboratory notebook on Gradescope. Make sure to include your name and student ID number in the writeup.

For questions which ask for plots, please put the plots directly in your writeup. For questions which ask you to write code, submit the code sample which you wrote as well.

Please copy the entire codebase, including the template code we wrote, and put it in an Appendix. For all code sections, make sure that the code is nicely formatted and properly wraps at the end of line if necessary. Finally, remember to show your work and explain your answers (this includes comments within the code).

5 Parting Thoughts

In this lab, we used floating point values for the scaling factors between layers. This is not as bad as it might first seem, because a multiplication by a constant floating point value can be mapped to a multiplication by a constant integer followed by a bit-shift. However, we could improve our area and power efficiency even further (at the cost of accuracy) by requiring that all our scaling factors be powers-of-2, so that they can be implemented with simple bit-shifters.

If you're up for an extra challenge, you could also change your code so that your scaling factors are powers-of-2. With a dataset as simple as CIFAR10, you should still see negligible accuracy degradation.

Finally, 8-bit integers are a common fixed-point target because they can preserve accuracy with only a few percentage points of loss for the majority of popular neural networks, but are still small enough to see dramatic power, performance, and area gains. But you could go lower than that for some networks, and you are free to experiment with 4-bit, 2-bit or variable bitwidth quantization as well. You might even want to reduce the bitwidth of your biases—32 bits was probably too conservative.