Lab 1: Quantization

EE 290-2 Hardware for Machine Learning
UC Berkeley, Spring 2020
Instructor: Prof. Sophia Yakun Shao
Teaching Assistants: Alon Amid and Hasan Genc

Due: February 5, 2020

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.

1.1 Getting Started

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:

- 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/12ETP3DfjvmL3C_J3MzCYQO6kx63xchet
- Select *File* → *Save a copy in Drive*.
- Select *Runtime* → *Change runtime type*, and then select *GPU* under the *Hardware accelerator* drop-down menu.

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

[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

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

The Colaboratory notebook that we provide already has code which will load the CIFAR10 dataset and train a simple convolutional neural network (CNN) to classify it. The network provided is similar to LeNet-5, and it has the following architecture:

Layer	**Type**	**Input Shape**	**Output Shape**	**Activation**
conv1	Convolutional	3x32x32	6x28x28	ReLU
pool1	Max pool	6x28x28	6x14x14	None
conv2	Convolutional	6x14x14	16x10x10	ReLU
pool2	Max pool	16x10x10	16x5x5	None
fc1	Fully-connected	400	120	ReLU
fc2	Fully-connected	120	84	ReLU
fc3	Fully-connected	84	10	None

None of the layers in the network have a bias associated with them. This makes them easier to quantize. Towards the end of the assignment, we will add biases to the final layer and quantize it as well.

Question 1: Visualize Weights

1. Plot histograms of the weights of every convolutional and fully-connected layer. Record any observations you make about the distribution of the values.
2. Record the range of the weights, as well as their 3-sigma range (the difference between $\mu + 3\sigma$ and $\mu - 3\sigma$). For which layers is the 3-sigma range larger or smaller than the actual range? Then explain which range you would prefer to use if you were to quantize each layer's weights and wanted to strike a balance between the range of values that could be expressed, and your precision.

Question 2: Quantize Weights

Any convolution or fully-connected layer pass, without a bias, can be described by the equation:

$$W * In = Out \tag{1}$$

where W is the weight tensor, In in the input tensor, and Out is the output tensor.

For this question, your task is to find a *scaling factor*, called n_W for each convolutional and fully connected layer, which would fit inside an 8-bit signed integer. Then,

$$n_W W * In = n_W Out \tag{2}$$

You might wonder: "Isn't it a problem that the output of the layer has now changed? Wouldn't quantizing the weights change the output of the neural net?"

The answer, of course, is: "Yes". However, what we care about is not the *absolute* values output by the CNN, but the relative difference between the probabilities it assigns to different classes for its predictions. Quantizing the weights only scales this relative difference up or down, but it does not affect which class the network assigns the most probability to. Therefore, it does not affect the final predictions that the neural net makes.

1. Fill in the `quantized_weights` function. The template code we provide will then call this function on the weights of every layer in the CNN that we just trained at 32-bit floating point precision, to lower them into 8-bit signed integer precision.

2. Record the accuracy degradation of the network after quantizing its weights. If you've done everything correctly, the accuracy degradation should be negligible.

Question 3: Visualize Activations

Now that we have quantized the weights of the CNN, we must also quantize the activations (inputs and outputs to layers) traveling through it. But before doing so, let's analyze what values the activations take when travelling through the network.

We provide convenience code which will record the values of every pixel of the outputs and inputs travelling through the neural network. (This is the initial CNN, where not even the weights had yet been quantized). We then profile these values when running on a subset of the training set.

1. Plot histograms of the input images and the outputs of every convolutional and fully-connected layer. Record any observations you make about the distribution of the values.

2. Additionally, record the range of the values, as well as their 3-sigma range (the difference between $\mu + 3\sigma$ and $\mu - 3\sigma$). For which layers is the 3-sigma range larger or smaller than the actual range? Then explain which range you would prefer to use if you were to quantize each layer's weights and wanted to strike a balance between the range of values that could be expressed, and your precision. Remember that you are plotting the activations *after* activation functions like ReLU have been applied, which means that you should not be worried if you find that your plots are asymmetric.

Question 4: Quantize Activations

Now it is time to quantize the activations (inputs and outputs to layers) traveling through the CNN. Our equation now becomes:

$$n_W W * n_{In} In * n_{Out} = n_W n_{In} n_{Out} Out \tag{3}$$

where n_{In} is the scaling factor which was applied to the input to the layer, and n_{Out} is the scaling factor which we decide to apply to the output of the layer. n_{Out} must be chosen such that the expected values of the elements of Out can be scaled down to fit within 8 bits.

1. Before performing any quantization at all, we could describe the output of the `conv1` layer as:

$$W_{conv1} * In = Out_{conv1} \tag{4}$$

   Suppose that we quantized the input matrix, In, scaling it down by n_{In}. Suppose that we also scaled the weight matrix, W_{conv1}, by $n_{W_{conv1}}$, and the output matrix, Out_{conv1}, by $n_{Out_{conv1}}$.

   (a) Write an equation describing the output of the `conv1` layer with these new scaling parameters.

(b) Write an equation describing the output of the `conv2` layer in terms of In, W_{conv1}, W_{conv2}, Out_{conv1}, Out_{conv2}, n_{In}, $n_{W_{conv1}}$, $n_{W_{conv2}}$, $n_{Out_{conv1}}$, and $n_{Out_{conv2}}$. You can pretend that the pooling layers do not exist.

2. Complete the `quantize_initial_input` and `quantize_activations` functions which calculate the scaling factors for the initial image which is input to the CNN, and the outputs of each layer, respectively.

3. Complete the `forward` function for the `NetQuantized` class. You will have to add code here to scale the outputs of each layer, and then to clamp the outputs of each layer to integers between -128 and 127 afterwards.

4. Finally, record the accuracy of your network after both weights and activations have been quantized. If you've done everything right, you should still find almost no accuracy degradation.

Question 5: Quantize Biases

Let us now update our CNN to include a bias in its final layer, fc3. We have already included code to create and train a new CNN called `net_with_bias`.

Consider how a bias affects the equation for an unquantized layer:

$$W * In + \beta = Out \tag{5}$$

where β is the bias.

1. Suppose that we again quantized a biased layer with the same scaling factors we used in previous questions: n_W, n_{In}, and n_{Out}. What would we scale β by in this case? Write an equation to describe the output of the quantized layer with a bias.

2. Fill in the `quantized_bias` function in the `NetQuantizedWithBias` class. This function is meant to quantize the bias on the final layer of the CNN. Keep in mind that biases are typically quantized to 32-bits, so your bias values do not all have to be between -128 and 127.

3. What is your accuracy before and after quantizing CNN with the bias? The accuracy degradation should ideally be negligible.

4 Lab Report Structure

Submit a PDF writeup of your responses to the questions in Section 3 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.

Finally, please copy the entire codebase, including the template code we wrote, and put it in an Appendix.

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.