AI Hardware and Programming
EE 4953/5453

Lab 5

Group 4

Allan Aanonsen, Mauricio Figueroa, Ivan Hernandez,
Enrique Alvarez

Objective

This lab aims to implement 8-bit quantization-aware training (QAT) for the LeNet-5 neural network architecture. The original LeNet-5 model uses 64-bit floating-point values. To enable QAT, quantization and dequantization functions are first developed to accurately convert between floating-point and 8-bit integer formats. The forward pass of each network layer is then modified to quantize and dequantize weights and biases before performing multiply-accumulate (MAC) operations, simulating 8-bit behavior during training. After training, all learned weights and biases are fully converted to 8-bit integer representations.

Introduction

Quantization is a key technique for reducing the computational and memory requirements of deep neural networks, making them more suitable for deployment on resource-constrained devices. Instead of relying on high-precision floating-point representations, quantized networks use lower-precision formats, such as 8-bit integers, to perform inference with minimal loss in accuracy. Quantization-aware training (QAT) further improves the performance of quantized models by simulating quantization effects during training, allowing the network to adapt to the reduced precision.

In this lab, 8-bit QAT is applied to the LeNet-5 convolutional neural network, a classic architecture originally developed for handwritten digit recognition. The baseline LeNet-5 implementation uses 64-bit floating-point values for all computations. To implement QAT, new quantization and dequantization functions are introduced, and the forward pass of each layer is modified to apply quantization to weights and biases during training. After training, the final model parameters are fully converted to 8-bit integer format, enabling efficient inference without further floating-point operations.

This project highlights the practical steps involved in implementing quantization, the challenges of maintaining model accuracy with reduced precision, and the benefits of QAT for real-world deployment scenarios.

Method

For detailed steps see out GitHub for this lab.

We continued extending the source code we used in Lab 4, from the LeNet5 implementation by fan-wenjie. First, we decided to develop a stable version of the file, *pc_training.c*, that had quantization implemented. This meant adding *quantize* and *dequantize* functions to the lenet.c and lenet.h files (renamed to lenet_quantized).

```c
int8_t quantize(double value)
{
    if (value > QUANT_SCALE) return QUANT_SCALE;
    if (value < -QUANT_SCALE) return -QUANT_SCALE;
    return (int8_t)(value * QUANT_SCALE);
}

double dequantize(int8_t value)
{
    return (double)value / QUANT_SCALE;
}
```

Then in the lenet_quantized.h file we add a struct to hold quantized weights/biases. We *do not* delete the regular struct we used in the previous lab since we still want the floating-point model (we use that in backprop). There were a lot of conflicting opinions online, some people we're saying that we're supposed to quantize/dequantize all the way through the training loop, so during forward and during backprop, but we found that didn't work at all.

```c
typedef struct LeNet5Quant {
    int8_t weight0_1[INPUT][LAYER1][LENGTH_KERNEL][LENGTH_KERNEL];
    int8_t weight2_3[LAYER2][LAYER3][LENGTH_KERNEL][LENGTH_KERNEL];
    int8_t weight4_5[LAYER4][LAYER5][LENGTH_KERNEL][LENGTH_KERNEL];
    int8_t weight5_6[LAYER5*LENGTH_FEATURE5*LENGTH_FEATURE5][OUTPUT];

    int8_t bias0_1[LAYER1];
    int8_t bias2_3[LAYER3];
    int8_t bias4_5[LAYER5];
    int8_t bias5_6[OUTPUT];
} LeNet5Quant;
```

Instead, we only quantize during the forward pass and update floating point during backprop, then store the latest quantized weights/biases. The forward pass ended up looking like this

```c
#define CONVOLUTE_VALID(input, output, weight)
{
    FOREACH(o0, GETLENGTH(output))
    FOREACH(o1, GETLENGTH(*(output)))
    FOREACH(w0, GETLENGTH(weight))
    FOREACH(w1, GETLENGTH(*(weight)))
        (output)[o0][o1] +=
            (input)[o0 + w0][o1 + w1] * qat_weight((weight)[w0][w1]);
}
```

```
#define CONVOLUTION_FORWARD(input, output, weight, bias, action)
{
    for (int in_ch = 0; in_ch < GETLENGTH(weight); ++in_ch)
        for (int out_ch = 0; out_ch < GETLENGTH(*(weight)); ++out_ch)
            CONVOLUTE_VALID(
                (input)[in_ch],
                (output)[out_ch],
                (weight)[in_ch][out_ch] );

    FOREACH(j, GETLENGTH(bias))
        FOREACH(i, GETCOUNT(output[j]))
            ((double*)(output)[j])[i] =
                action(
                  ((double*)(output)[j])[i] + qat_bias((bias)[j]) );
}
```

```
#define DOT_PRODUCT_FORWARD(input, output, weight, bias, action)
{
    for (int x = 0; x < GETLENGTH(weight); ++x)
        for (int j = 0; j < GETLENGTH(*(weight)); ++j)
            ((double*)(output))[j] +=
                ((double*)(input))[x] * qat_weight( ((weight)[x][j]) );
    FOREACH(j, GETLENGTH(bias))
        ((double*)(output))[j] =
            action( ((double*)(output))[j] + qat_bias( (bias)[j] ) );
}
```

where QAT related operations can be seen in the yellow text.

The backpropagation was not touched. The training loop was modified to store quantized weights/biases at the end of the loop.

For the *pc_training_quantized.c* test the results were

```
FP accuracy:        9624 / 10000 (96.24%)
FP inference: 41.93 s total, 4.19 ms/image
QAT-int8 accuracy:  9624 / 10000 (96.24%)
QAT-int8 inference: 17.07 s total, 1.71 ms/image
```

For the *pc_jetson_test.c* test the results were

```
FP  Accuracy: 9547 / 10000 (95.47%)
FP  Time:     345.27 s total, 34.53 ms/image

QAT Accuracy: 9547 / 10000 (95.47%)
QAT Time:     150.48 s total, 15.05 ms/image
```

For the jetson_test_quantized.cu the results were

```
QAT-int8 CUDA Accuracy: 9547 / 10000 (95.47%)
Elapsed Time: 0.15 seconds
```

This lab was especially challenging because of all the conflicting opinions available online. There are also more nuanced details that don't particularly get mentioned such as selecting an adequate batch size and learning rate combination (in our case 0.5 was way too high and 0.1 worked well, with a batch of 300).

Results

Model	Accuracy	Time (standard | quantized)		
PC Training	9624/10000	406 KB	5m 47s | 10m 20s	51 KB
PC Test	...	...	42s | 17s	...
Jetson Single-thread	...	...	5m 45s | 2m 30s	...
Jetson Multi-thread	...	...	0.1s | 0.15s	...

Discussion

The results demonstrate that 8-bit quantization significantly reduces model file size and inference time without impacting accuracy. The quantized LeNet-5 model achieved the same 95% test accuracy as the original 64-bit floating-point version, while reducing the file size from 406 KB to 51 KB. Inference time using a single thread decreased from 5 minutes 45 seconds to 2 minutes 30 seconds. With multi-threading enabled, inference time further improved to 0.15 seconds. These results highlight that quantization not only improves storage and computational efficiency but also enables faster model deployment, making it well-suited for AI hardware applications.