Quantization

Manual

AIX 2024

서울대학교 차세대반도체 혁신공유대학

Xuan-Truong Nguyen (응웬트렁)

truongnx@snu.ac.kr

If you have no background in **YOLO** or **Quantization**, please refer to the links below:

YOLO:

https://www.notion.so/aix2024tutorial02/02-YOLO-network-AIX-tutorial-3b4314c624bf4007a4268df33b0d6b82

Quantization:

https://www.notion.so/aix2024tutorial02/03-Quantization-AIX-tutorial-5a03d7366197428cbf1454e4be97057c

1. Source code related to quantization

(in C:\skeleton\src)

- yolov2_forward_network_quantized.c

  = Functions for quantization, saving of the quantized model, and the forward pass of quantized yolo model

  (↑ You should mainly edit this file for quantization!)

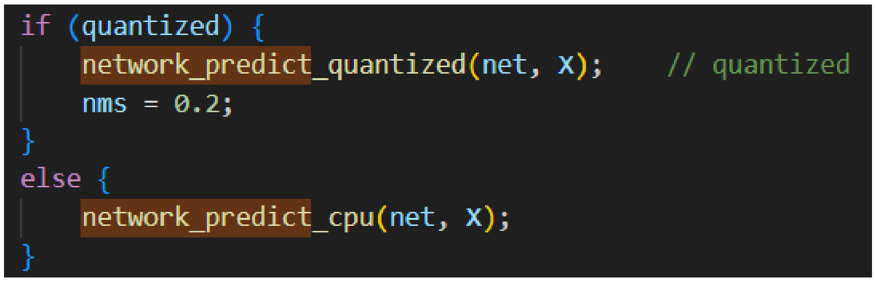
- main.c   *// The main functions*

In "main.c", there is a function named "test_detector_cpu".

[test_detector_cpu] □ main.c

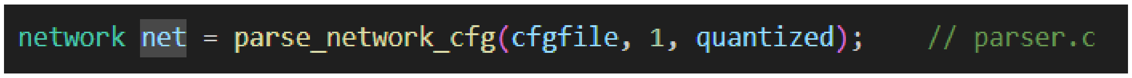
```
void test_detector_cpu(
    char **names,          // List of all items (bin/yolohw.names)
    char *cfgfile,         // Configuration file (bin/aix2024.cfg)
    char *weightfile,      // Configuration file (bin/aix2024.weights)
    char *filename,              // Input image file
    float thresh,                // Hierarchical threshold
    int quantized,               // On/off quantization
    int save_params,             // On/off save output
    int dont_show                // Don't show
    )
```

This function **parse the configuration file** and **call an inference function**.



```
if (quantized) {
    network_predict_quantized(net, X);    // quantized
    nms = 0.2;
}
else {
    network_predict_cpu(net, X);
}
```

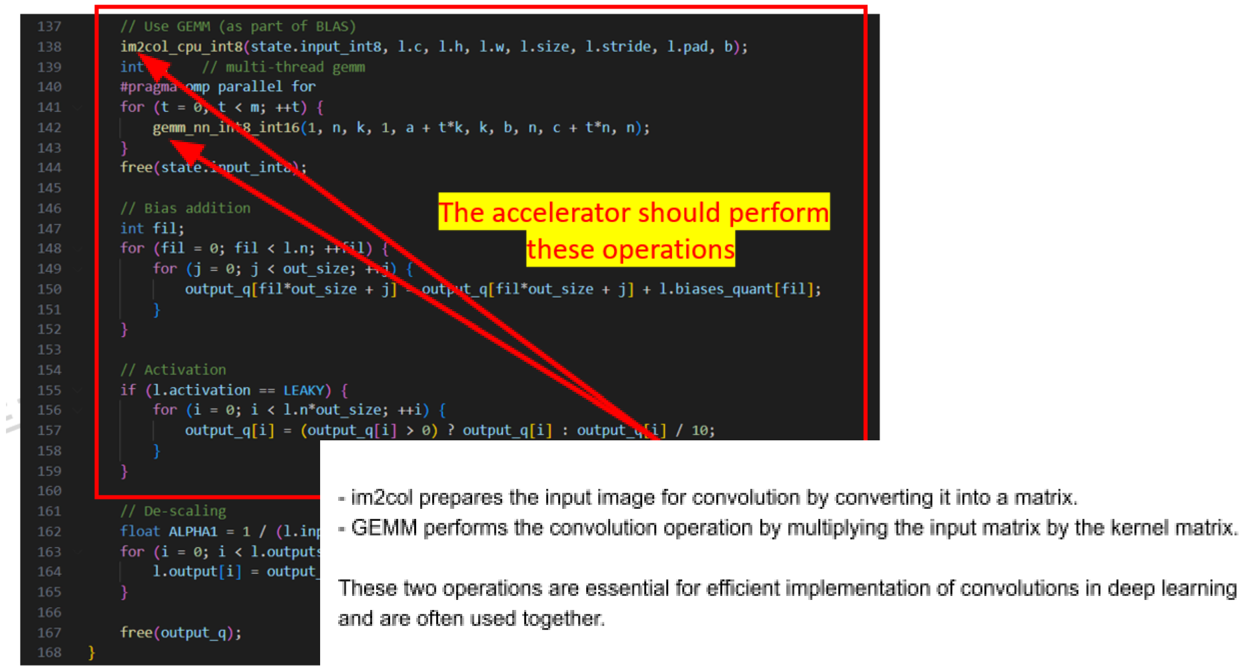
A network **architecture** is stored in the variable **"net"**.



```
network net = parse_network_cfg(cfgfile, 1, quantized);    // parser.c
```

2. Weight/bias/activation Quantization

The code below represents a series of operations that the accelerator should perform for detection. Whether you use a full precision model or a quantized model, detection goes through the following process.

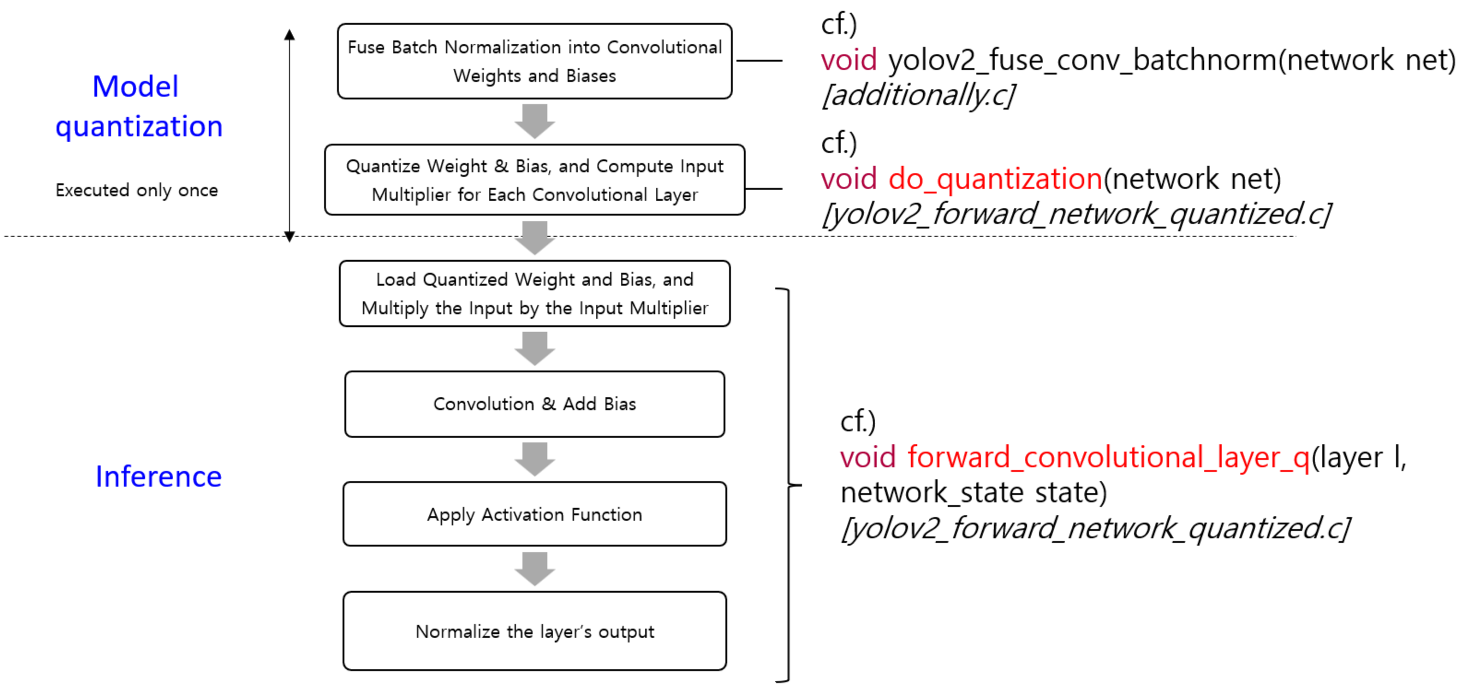


```c
    // Use GEMM (as part of BLAS)
    im2col_cpu_int8(state.input_int8, l.c, l.h, l.w, l.size, l.stride, l.pad, b);
    int t;     // multi-thread gemm
    #pragma omp parallel for
    for (t = 0; t < m; ++t) {
        gemm_nn_int8_int16(1, n, k, 1, a + t*k, k, b, n, c + t*n, n);
    }
    free(state.input_int8);

    // Bias addition
    int fil;
    for (fil = 0; fil < l.n; ++fil) {
        for (j = 0; j < out_size; ++j) {
            output_q[fil*out_size + j] = output_q[fil*out_size + j] + l.biases_quant[fil];
        }
    }

    // Activation
    if (l.activation == LEAKY) {
        for (i = 0; i < l.n*out_size; ++i) {
            output_q[i] = (output_q[i] > 0) ? output_q[i] : output_q[i] / 10;
        }
    }

    // De-scaling
    float ALPHA1 = 1 / (l.inp
    for (i = 0; i < l.outputs
        l.output[i] = output_
    }

    free(output_q);
}
```

- im2col prepares the input image for convolution by converting it into a matrix.
- GEMM performs the convolution operation by multiplying the input matrix by the kernel matrix.

These two operations are essential for efficient implementation of convolutions in deep learning and are often used together.

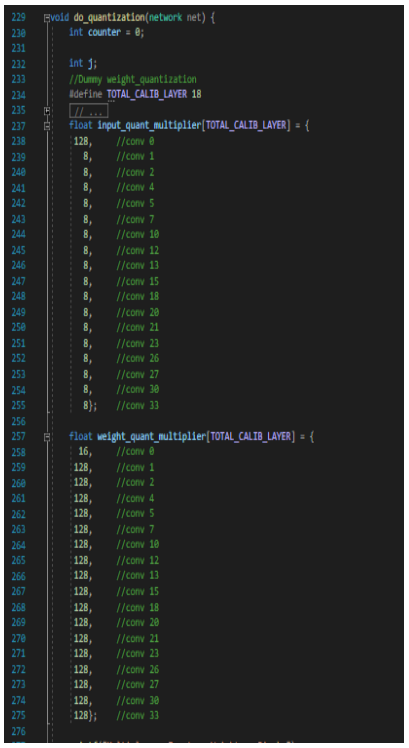
The difference between the two is determined by whether or not the data is quantized prior to detection. The process of using a quantized model is as follows.


Model quantization
Executed only once
Fuse Batch Normalization into Convolutional Weights and Biases
Quantize Weight & Bias, and Compute Input Multiplier for Each Convolutional Layer
cf.) void yolov2_fuse_conv_batchnorm(network net) [additionally.c]
cf.) void do_quantization(network net) [yolov2_forward_network_quantized.c]
Inference
Load Quantized Weight and Bias, and Multiply the Input by the Input Multiplier
Convolution & Add Bias
Apply Activation Function
Normalize the layer's output
cf.) void forward_convolutional_layer_q(layer l, network_state state) [yolov2_forward_network_quantized.c]

Let’s see how quantization is implemented in the code.

[do_quantization] □ @ yolov2_forward_network_quantized.c

- **Input:**
  - Network
- **Output**
  - Quantized model



```c
void do_quantization(network net) {
    int counter = 0;

    int j;
    //Dummy weight_quantization
    #define TOTAL_CALIB_LAYER 18
    // ...
    float input_quant_multiplier[TOTAL_CALIB_LAYER] = {
     128,     //conv 0
       8,     //conv 1
       8,     //conv 2
       8,     //conv 4
       8,     //conv 5
       8,     //conv 7
       8,     //conv 10
       8,     //conv 12
       8,     //conv 13
       8,     //conv 15
       8,     //conv 18
       8,     //conv 20
       8,     //conv 21
       8,     //conv 23
       8,     //conv 26
       8,     //conv 27
       8,     //conv 30
       8};    //conv 33

    float weight_quant_multiplier[TOTAL_CALIB_LAYER] = {
      16,     //conv 0
     128,     //conv 1
     128,     //conv 2
     128,     //conv 4
     128,     //conv 5
     128,     //conv 7
     128,     //conv 10
     128,     //conv 12
     128,     //conv 13
     128,     //conv 15
     128,     //conv 18
     128,     //conv 20
     128,     //conv 21
     128,     //conv 23
     128,     //conv 26
     128,     //conv 27
     128,     //conv 30
     128};    //conv 33

```

In AIX 2024, it is our job to **find best set of multipliers** □ for quantization. It is now set to default values.

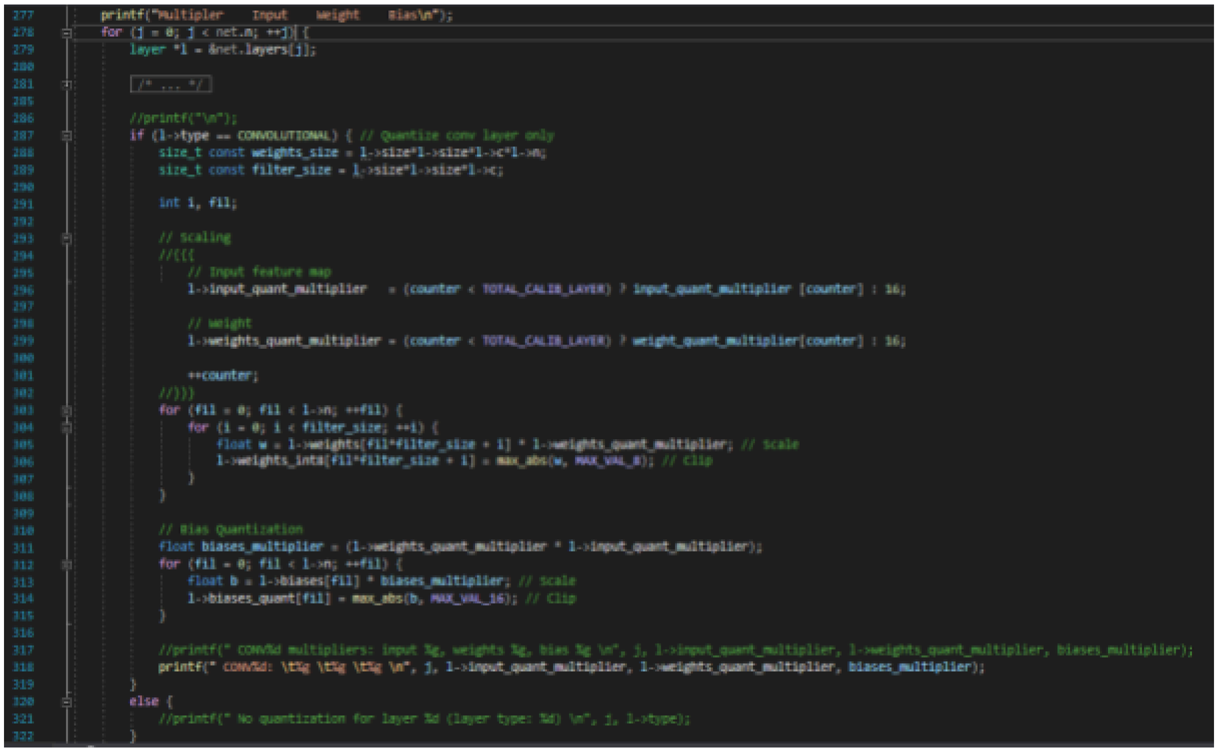
As described in the tutorial03, the goal of quantization is to convert **from FP32 to a fixed-point format.** For AIX 2024 quantization, we use **8 bits for weights and 16 bits for biases**.

- Two steps to quantize **weights and biases**
  1) **Scaling**: Multiply by a scale factor(multiplier)
  2) **Clipping**: avoid overflow

     Ex) int8□ if x>127, x=127 or if x<-128, x=-128

  For example, **multiply a given weight by 16 and apply the round-to-nearest on the result;** we can obtain a fixed-point number with four fractional bits, three integer bits, and one sign bit.

↓You can also **implement your own quantization** to minimize mAP degradation.



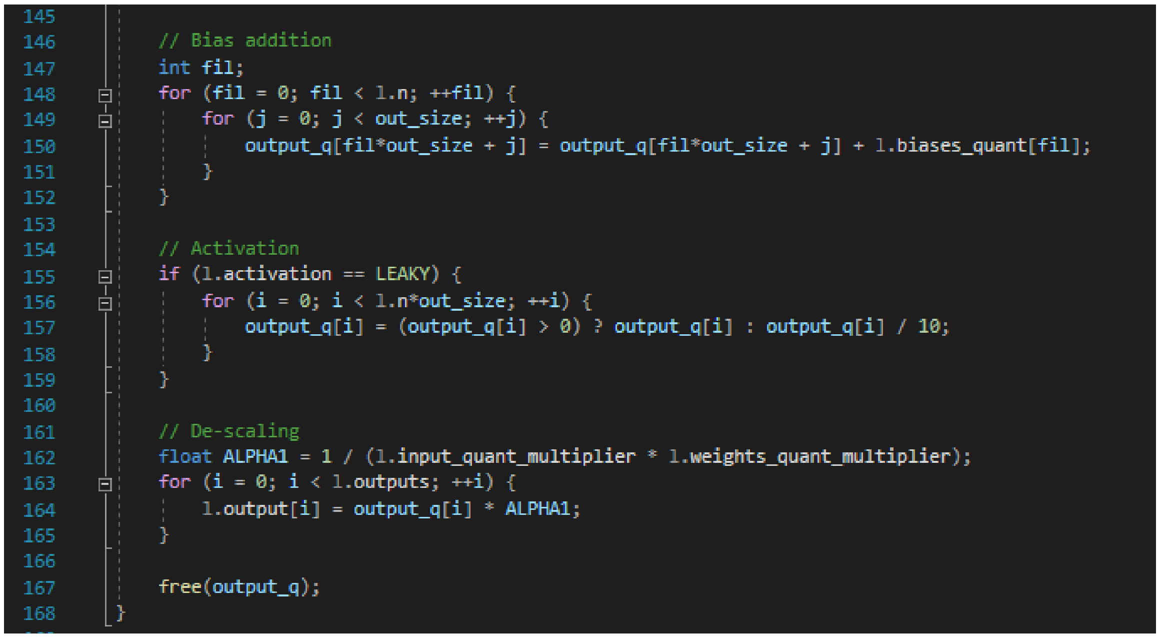
```c
    printf("Multipler    Input    Weight    Bias\n");
    for (j = 0; j < net.n; ++j) {
        layer *l = &net.layers[j];

        /* ... */

        //printf("\n");
        if (l->type == CONVOLUTIONAL) { // Quantize conv layer only
            size_t const weights_size = l->size*l->size*l->c*l->n;
            size_t const filter_size = l->size*l->size*l->c;

            int i, fil;

            // Scaling
            //{{{
                // Input feature map
                l->input_quant_multiplier   = (counter < TOTAL_CALIB_LAYER) ? input_quant_multiplier [counter] : 16;

                // Weight
                l->weights_quant_multiplier = (counter < TOTAL_CALIB_LAYER) ? weight_quant_multiplier[counter] : 16;

                ++counter;
            //}}}
            for (fil = 0; fil < l->n; ++fil) {
                for (i = 0; i < filter_size; ++i) {
                    float w = l->weights[fil*filter_size + i] * l->weights_quant_multiplier; // Scale
                    l->weights_int8[fil*filter_size + i] = max_abs(w, MAX_VAL_8); // Clip
                }
            }

            // Bias Quantization
            float biases_multiplier = (l->weights_quant_multiplier * l->input_quant_multiplier);
            for (fil = 0; fil < l->n; ++fil) {
                float b = l->biases[fil] * biases_multiplier; // Scale
                l->biases_quant[fil] = max_abs(b, MAX_VAL_16); // Clip
            }

            //printf(" CONV%d multipliers: input %g, weights %g, bias %g \n", j, l->input_quant_multiplier, l->weights_quant_multiplier, biases_multiplier);
            printf(" CONV%d: \t%g \t%g \t%g \n", j, l->input_quant_multiplier, l->weights_quant_multiplier, biases_multiplier);
        }
        else {
            //printf(" No quantization for layer %d (layer type: %d) \n", j, l->type);
        }
```

[forward_convolutional_layer_q] □ @ yolov2_forward_network_quantized.c

- Similar to weight, **activation** can be quantized by multiplying it by a factor, i.e., input_quant_multipler, and applying the round-to-nearest operation on the multiplication result.
- Again, activation quantization also aims to find a fixed-point format of a floating-point number. More importantly, **it allows to do a 8-bit x 8-bit multiplication** (gemm_nn_int8_int16 at Line 142).

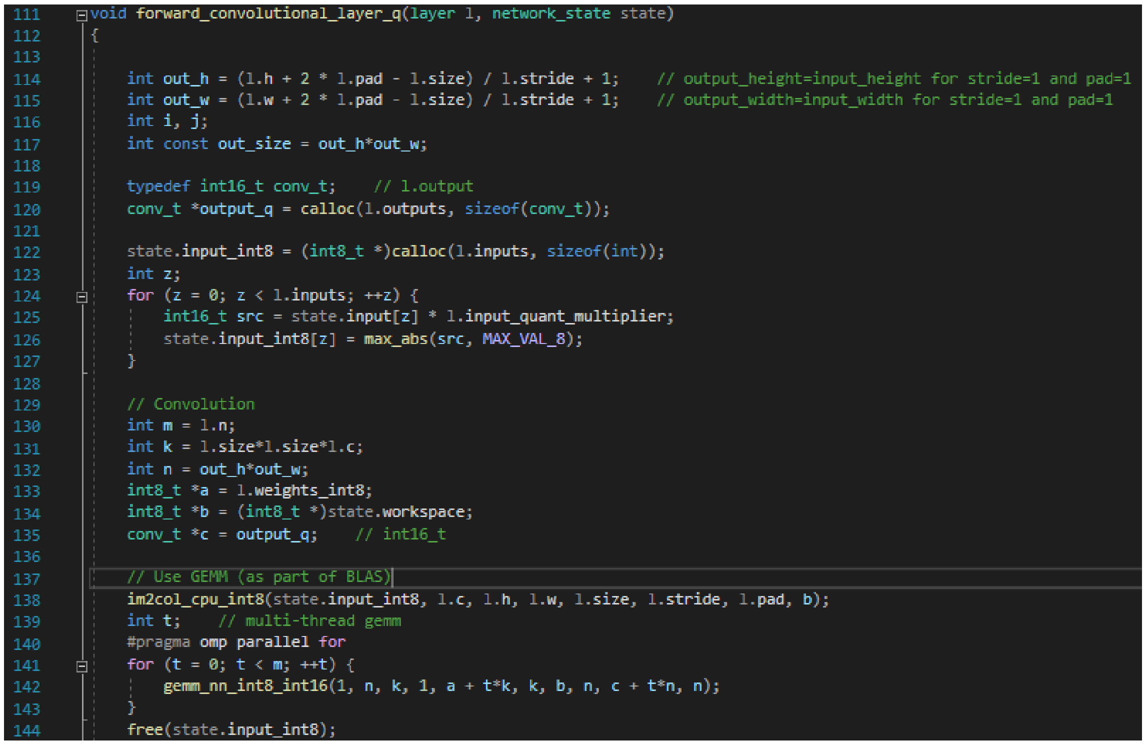
3. Descaling

After convolution, it adds a bias, performs activation, and then do **descaling**. Descaling can be considered as conversion from a fixed-point number to a floating-point one. For the model's **outputs to be interpretable**, they must be **converted back to their original scale.**



```

    // Bias addition
    int fil;
    for (fil = 0; fil < l.n; ++fil) {
        for (j = 0; j < out_size; ++j) {
            output_q[fil*out_size + j] = output_q[fil*out_size + j] + l.biases_quant[fil];
        }
    }

    // Activation
    if (l.activation == LEAKY) {
        for (i = 0; i < l.n*out_size; ++i) {
            output_q[i] = (output_q[i] > 0) ? output_q[i] : output_q[i] / 10;
        }
    }

    // De-scaling
    float ALPHA1 = 1 / (l.input_quant_multiplier * l.weights_quant_multiplier);
    for (i = 0; i < l.outputs; ++i) {
        l.output[i] = output_q[i] * ALPHA1;
    }

    free(output_q);
}
```

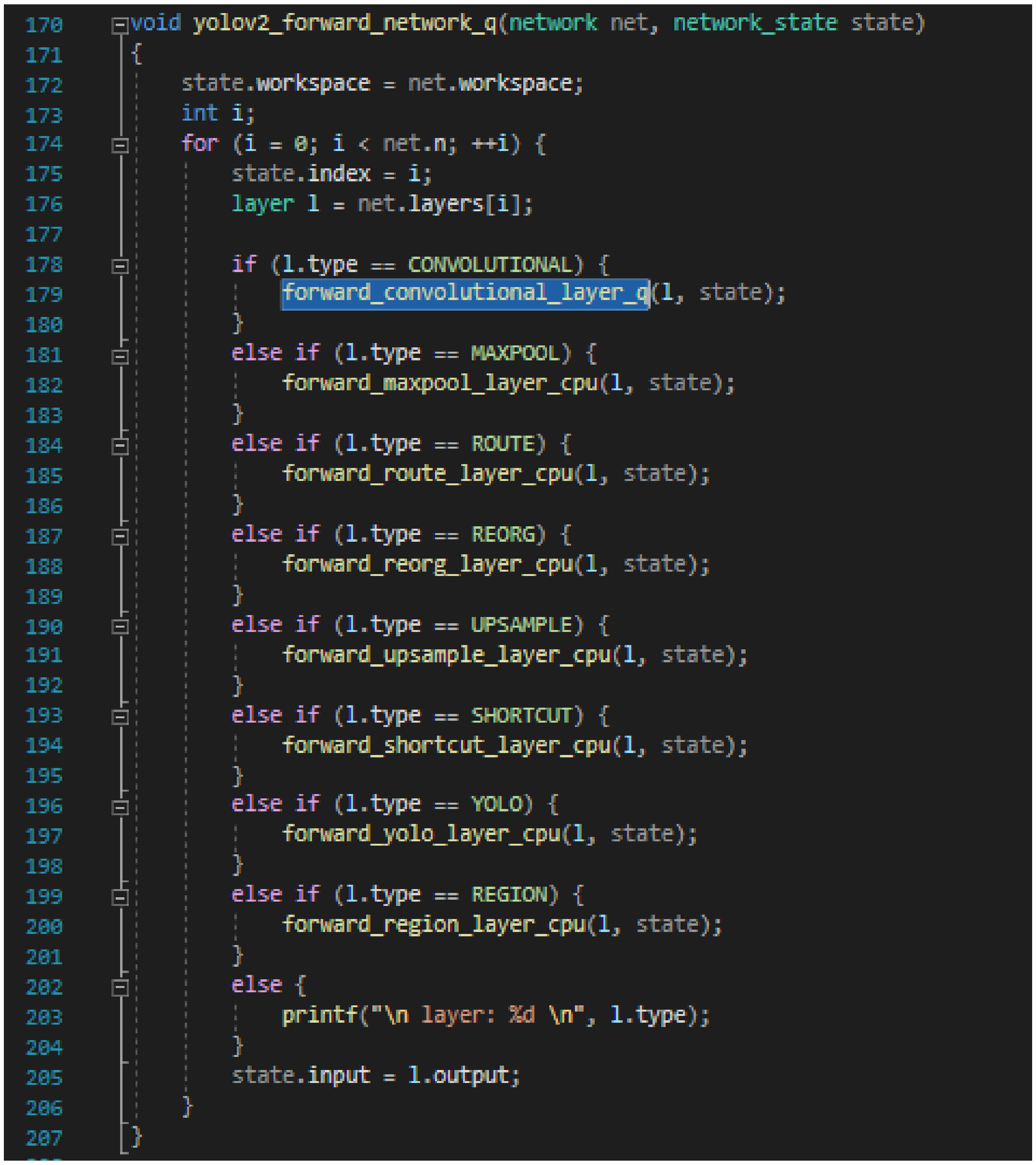
We can **merge a descaling step to an input quantization** (Lines 125-126). In other words, we want to store input_int8.



```c
void forward_convolutional_layer_q(layer l, network_state state)
{

    int out_h = (l.h + 2 * l.pad - l.size) / l.stride + 1;    // output_height=input_height for stride=1 and pad=1
    int out_w = (l.w + 2 * l.pad - l.size) / l.stride + 1;    // output_width=input_width for stride=1 and pad=1
    int i, j;
    int const out_size = out_h*out_w;

    typedef int16_t conv_t;    // l.output
    conv_t *output_q = calloc(l.outputs, sizeof(conv_t));

    state.input_int8 = (int8_t *)calloc(l.inputs, sizeof(int));
    int z;
    for (z = 0; z < l.inputs; ++z) {
        int16_t src = state.input[z] * l.input_quant_multiplier;
        state.input_int8[z] = max_abs(src, MAX_VAL_8);
    }

    // Convolution
    int m = l.n;
    int k = l.size*l.size*l.c;
    int n = out_h*out_w;
    int8_t *a = l.weights_int8;
    int8_t *b = (int8_t *)state.workspace;
    conv_t *c = output_q;    // int16_t

    // Use GEMM (as part of BLAS)
    im2col_cpu_int8(state.input_int8, l.c, l.h, l.w, l.size, l.stride, l.pad, b);
    int t;    // multi-thread gemm
    #pragma omp parallel for
    for (t = 0; t < m; ++t) {
        gemm_nn_int8_int16(1, n, k, 1, a + t*k, k, b, n, c + t*n, n);
    }
    free(state.input_int8);
```

[yolov2_forward_network_q] @ yolov2_forward_network_quantized.c

This function is similar to yolov2_forward_network, which **takes a model and an image and generates the detected tensors**. It is also executed layer by layer, where a layer can be convolution, max-pooling, route, or upsample.



```
void yolov2_forward_network_q(network net, network_state state)
{
    state.workspace = net.workspace;
    int i;
    for (i = 0; i < net.n; ++i) {
        state.index = i;
        layer l = net.layers[i];

        if (l.type == CONVOLUTIONAL) {
            forward_convolutional_layer_q(l, state);
        }
        else if (l.type == MAXPOOL) {
            forward_maxpool_layer_cpu(l, state);
        }
        else if (l.type == ROUTE) {
            forward_route_layer_cpu(l, state);
        }
        else if (l.type == REORG) {
            forward_reorg_layer_cpu(l, state);
        }
        else if (l.type == UPSAMPLE) {
            forward_upsample_layer_cpu(l, state);
        }
        else if (l.type == SHORTCUT) {
            forward_shortcut_layer_cpu(l, state);
        }
        else if (l.type == YOLO) {
            forward_yolo_layer_cpu(l, state);
        }
        else if (l.type == REGION) {
            forward_region_layer_cpu(l, state);
        }
        else {
            printf("\n layer: %d \n", l.type);
        }
        state.input = l.output;
    }
}
```

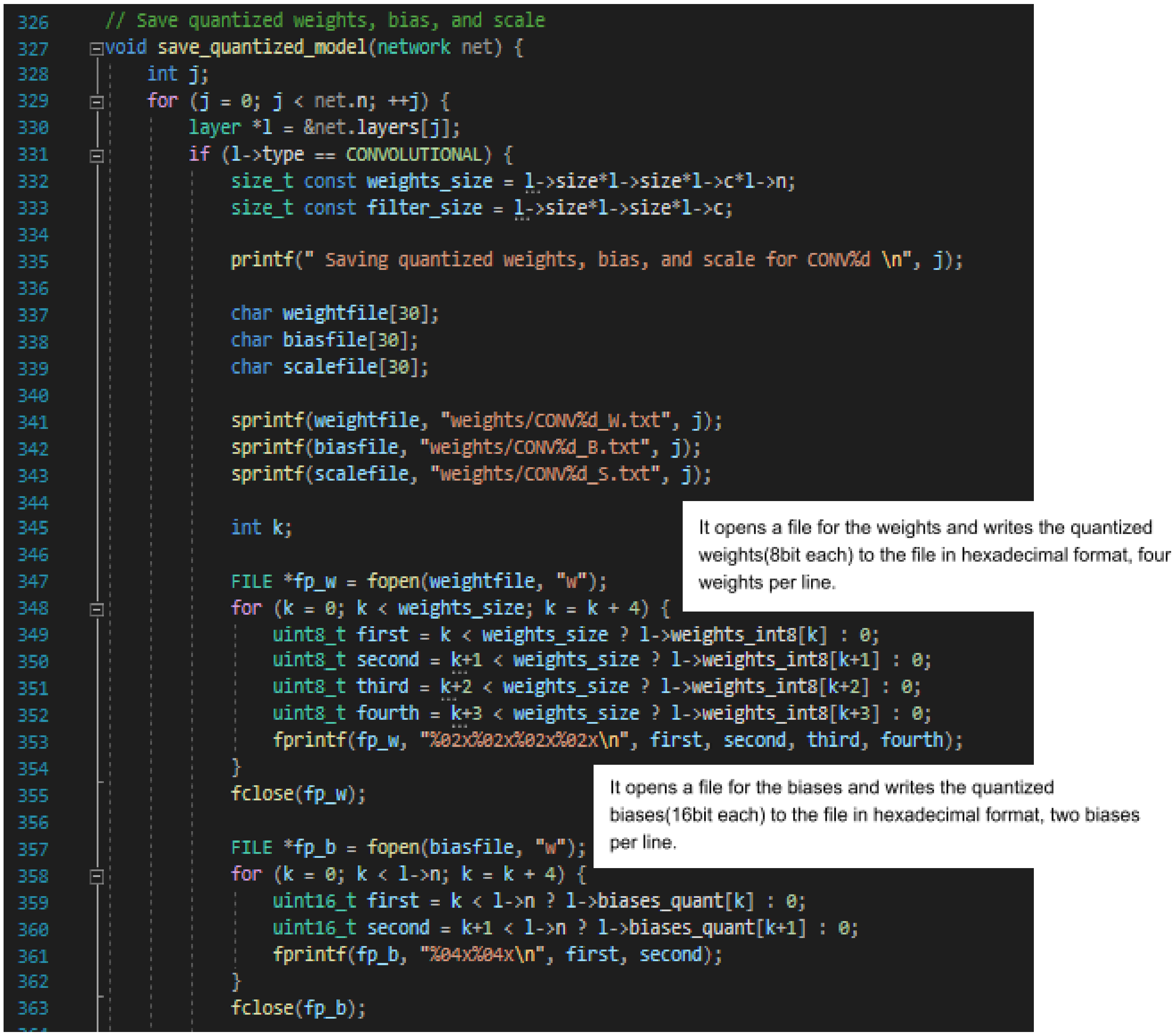
Note that the **final output** consists of the two tensors **8x8x195 and 16x16x195 from Layers 27 and 33,** respectively, which are **post-processed by the YOLO layers to generate bounding boxes.**

4. Prepare Quantized Data

1) Quantized Model

[save_quantized_model] @ yolov2_forward_network_quantized.c

You can use a template provided to store weights and biases layer by layer. The quantized model is stored in your computing order. Our quantized weights are generally stored in DRAM or block ram (BRAM). From DRAM or BRAM, a CNN engine must load weights into registers, when computing each layer.



```c
    // Save quantized weights, bias, and scale
void save_quantized_model(network net) {
    int j;
    for (j = 0; j < net.n; ++j) {
        layer *l = &net.layers[j];
        if (l->type == CONVOLUTIONAL) {
            size_t const weights_size = l->size*l->size*l->c*l->n;
            size_t const filter_size = l->size*l->size*l->c;

            printf(" Saving quantized weights, bias, and scale for CONV%d \n", j);

            char weightfile[30];
            char biasfile[30];
            char scalefile[30];

            sprintf(weightfile, "weights/CONV%d_W.txt", j);
            sprintf(biasfile, "weights/CONV%d_B.txt", j);
            sprintf(scalefile, "weights/CONV%d_S.txt", j);

            int k;

            FILE *fp_w = fopen(weightfile, "w");
            for (k = 0; k < weights_size; k = k + 4) {
                uint8_t first = k < weights_size ? l->weights_int8[k] : 0;
                uint8_t second = k+1 < weights_size ? l->weights_int8[k+1] : 0;
                uint8_t third = k+2 < weights_size ? l->weights_int8[k+2] : 0;
                uint8_t fourth = k+3 < weights_size ? l->weights_int8[k+3] : 0;
                fprintf(fp_w, "%02x%02x%02x%02x\n", first, second, third, fourth);
            }
            fclose(fp_w);

            FILE *fp_b = fopen(biasfile, "w");
            for (k = 0; k < l->n; k = k + 4) {
                uint16_t first = k < l->n ? l->biases_quant[k] : 0;
                uint16_t second = k+1 < l->n ? l->biases_quant[k+1] : 0;
                fprintf(fp_b, "%04x%04x\n", first, second);
            }
            fclose(fp_b);
```

It opens a file for the weights and writes the quantized weights(8bit each) to the file in hexadecimal format, four weights per line.

It opens a file for the biases and writes the quantized biases(16bit each) to the file in hexadecimal format, two biases per line.

2) Quantized Input image

- **Input image:** The input image has a size of 256x256x3, where pixels are ordered in a 1D array. Note that the **input image is normalized to [0, 1].** When executing the first layer, the input image is **multiplied by an input quantized multiplier** to generate fixed-point pixels.

⇒ Consider saving input in int8 format and sending it to DRAM.

- **Input feature maps:** Each convolution layer may take the output of its previous layer as an input. Therefore, it is necessary to **store input_int8 in a specific format as a test vector.**

3) Output feature maps

- To verify the functionality of a CNN engine, it requires **generating a test vector for output.** When designing an engine, we must compare the output from the HW simulation and the output from the AIX2024 SDK. Make sure that the results are identical. Again, you could use the sample template to write a file.

References

[4]. Jacob, “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference,” CVPR 2018.