MAS WES – 268

Project5 BNN

Rajan Verma

Github Path : https://github.com/rverma999/wes237C/tree/main/project5_bnn

I tried to code the BNN but couldn't get a pass on the testbench. I think the TB is very sensitive to how the coding happens and only works with a preset coding style any deviation from that doesn't work.

This is a hypothesis , I still dint get time to debug it further or try alternative approaches.

Also I plan to try and updating the weighs but that again I couldn't achieve as yet.

I tried reversing the binarizing logic, updating ignoring 16 MSB and 16 LSBs but I always got the TB failing absolutely. I think a deeper understanding is required for me to truly be able to debug this. May be once I try to update the weights I will understand depths of this NN.

Each Layer explained :

First layer : 784->128 neurons

- Takes binary input (packed in 25 32-bit words)
- Performs XNOR operation between inputs and weights
- Uses popcount to count matching bits
- Applies threshold activation function

Second layer : 128->64

- Packs previous layer's outputs into bits
- Performs XNOR with weights
- Uses popcount for bit counting
- Applies same threshold activation

Third Layer 64->10

- Similar bit packing and XNOR operations as perevious layers.
- No activation function in final layer
- Outputs raw scores for each class

Code:

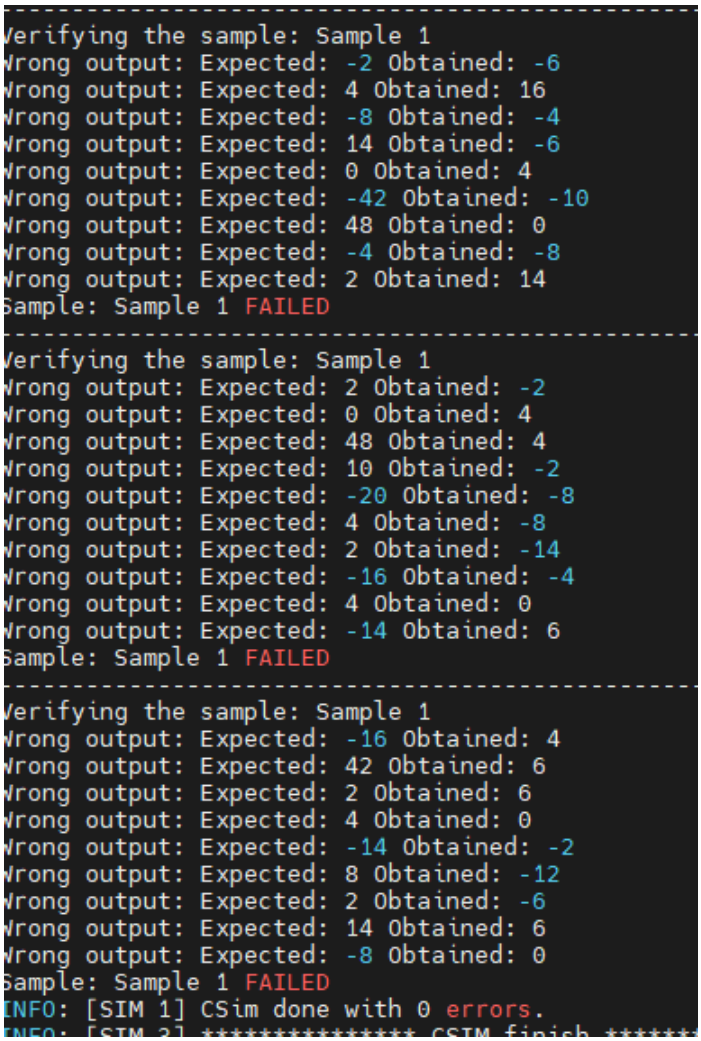
```
// First Layer
        printf("\n\n ******************\nSample weights:\n");
        for(int i = 0; i < 5; i++) {
            printf("w1[%d] = %u\n", i, w1[i]);
            printf("w2[%d] = %u\n", i, w2[i]);
            printf("w3[%d] = %u\n", i, w3[i]);
        }

memset(sum, 0, sizeof(sum));
for(int in = 0; in < 25; in++) { //each sample
    for(int out_n = 0; out_n < 128; out_n++) { //neuron   128X25 = 3200
        unsigned int input_word = IN[in];
        unsigned int weight_word = w1[out_n * 25 + in];
        if (in == 24) {
            // Mask to keep only the upper 16 bits (valid data)
            //input_word &= 0xFFFF0000;
            //weight_word &= 0xFFFF0000;
            input_word &= 0x0000FFFF;
            weight_word &= 0x0000FFFF;
        }
        xnor_result = ~(input_word ^ weight_word);
        sum[out_n] += __builtin_popcount(xnor_result);
        if(in==0) printf("first layer : xnor_result=%x sum=",xnor_result,sum[out_n]);       }
}
for(int out_n = 0; out_n < 128; out_n++) {
        if(out_n<2)printf("Layer1_out[%d]  before processing sum_out = %d\n", out_n, sum[out_n]);
    sum[out_n] = sum[out_n] * 2 - 784;
    if(out_n<2)printf("Layer1_out[%d]  after processing sum_out = %d\n", out_n, sum[out_n]);
    layer1_out[out_n] = sum[out_n] > 0 ? 1 : -1; //binaring
}
```

```
// Second Layer
memset(sum, 0, sizeof(sum));
for(int out_n = 0; out_n < 64; out_n++) {
    sum[out_n] = 0;
    for(int word = 0; word < 4; word++) {
        unsigned int input_bits = 0;
        for(int bit = 0; bit < 32; bit++) {
            int input_idx = (word * 32) + bit;
            if(input_idx < 128) {
                if(layer1_out[input_idx] == -1) {  // Map +1 to bit 1
                    input_bits |= (1u << bit);
                }
                // No need to set bit if layer1_out[input_idx] == -1 (bit remains 0)
            }
        }
        xnor_result = ~(input_bits ^ w2[out_n * 4 + word]); // XNOR
        sum[out_n] += __builtin_popcount(xnor_result);
    }
    sum[out_n] = sum[out_n] * 2 - 128;
    layer2_out[out_n] = sum[out_n] > 0 ? 1 : -1;
}

for(int i = 0; i < 64; i++) {
    //if(i < 5) printf("Layer2_out[%d] = %d\n", i, layer2_out[i]);
    printf("Layer2_out[%d] = %d\n", i, layer2_out[i]);
}

// Final Layer
memset(sum, 0, sizeof(sum));
for(int out_n = 0; out_n < 10; out_n++) {
    sum[out_n] = 0;
    for(int word = 0; word < 2; word++) {
        unsigned int input_bits = 0;
        for(int bit = 0; bit < 32; bit++) {
            int input_idx = word * 32 + bit;
            if(input_idx < 64) {
                if(layer2_out[input_idx] == -1) {  // Map +1 to bit 1
                    input_bits |= (1u << bit);
                }
                // No need to set bit if layer2_out[input_idx] == -1 (bit remains 0)
            }
        }
        xnor_result = ~(input_bits ^ w3[out_n * 2 + word]); // XNOR
        sum[out_n] += __builtin_popcount(xnor_result);
    }
    sum[out_n] = sum[out_n] * 2 - 64;
    // No activation function in the output layer
}
```


Verifying the sample: Sample 1
Wrong output: Expected: -2 Obtained: -6
Wrong output: Expected: 4 Obtained: 16
Wrong output: Expected: -8 Obtained: -4
Wrong output: Expected: 14 Obtained: -6
Wrong output: Expected: 0 Obtained: 4
Wrong output: Expected: -42 Obtained: -10
Wrong output: Expected: 48 Obtained: 0
Wrong output: Expected: -4 Obtained: -8
Wrong output: Expected: 2 Obtained: 14
Sample: Sample 1 FAILED
Verifying the sample: Sample 1
Wrong output: Expected: 2 Obtained: -2
Wrong output: Expected: 0 Obtained: 4
Wrong output: Expected: 48 Obtained: 4
Wrong output: Expected: 10 Obtained: -2
Wrong output: Expected: -20 Obtained: -8
Wrong output: Expected: 4 Obtained: -8
Wrong output: Expected: 2 Obtained: -14
Wrong output: Expected: -16 Obtained: -4
Wrong output: Expected: 4 Obtained: 0
Wrong output: Expected: -14 Obtained: 6
Sample: Sample 1 FAILED
Verifying the sample: Sample 1
Wrong output: Expected: -16 Obtained: 4
Wrong output: Expected: 42 Obtained: 6
Wrong output: Expected: 2 Obtained: 6
Wrong output: Expected: 4 Obtained: 0
Wrong output: Expected: -14 Obtained: -2
Wrong output: Expected: 8 Obtained: -12
Wrong output: Expected: 2 Obtained: -6
Wrong output: Expected: 14 Obtained: 6
Wrong output: Expected: -8 Obtained: 0
Sample: Sample 1 FAILED
INFO: [SIM 1] CSim done with 0 errors.