CS205 C/C++ programming - Implementation of Convolutional Layer in Deep Learning using C

Name: Bryan Anthony

SID: 12112230

Part 1 - Analysis

The problem is to Implementing Convolution Operation: You can just design a function (surely some other functions can be called by it) that takes an input image, some kernels and other parameters needed.

One common way to implement the convolutional layer in deep learning is by matrix multiplication. In this approach, the input Matrix and kernels are both reshaped into matrices, and the convolution operation is performed by multiplying the input matrix with a kernel matrix. This results in a new matrix that represents the output feature map.

One way to implement the convolutional layer using matrix multiplication is to convert the kernel into a one-dimensional array and extract certain parts of the input matrix as one-dimensional arrays. The dot product and the vector addition can then be computed using OpenMP and SIMD instructions to speed up the process.

Further explanation, to do the convolutional layer I divided it into two parts. the first part is convolution with n dimension. it means the number of channels could be more than one channel and divided into one channel each. In each channel, we do the convolution layer 1-dimensional which is the dot product and we sum all of it with the other channel result.

For the vector add is easier because we just need to add the output feature map become one matrix. And they all are in the same size.

The formula I used to calcululate the ouput size is
output width = Input matrix width - kernel width + 1
output Heigh = Input matrix height - kernel height + 1
output channel = 1

Part 2 - Code

conv.h

```
#pragma once
#include <stddef.h>
#include <stdbool.h>

typedef struct matrixData
{

    int width;
    int height;
    int channels;
    float *data;

} MatrixData;
MatrixData createMatrix(float, float, float, float *);
void freeMatrix(MatrixData *);
float dotproduct(const float *, const float *, int);
bool relu(MatrixData *);
bool add_padding(MatrixData *m, int padding);
void printMatrix(MatrixData *);
MatrixData convolution1d(MatrixData *inputMatrix, MatrixData *kernel, int start);
MatrixData convolutionNd(MatrixData *inputMatrix, MatrixData *kernel);
bool vecAdd(float *p2, const float *p1, int num);
```

conv.c

```
#include <stdlib.h>
#include <stdio.h>
```

```
#include <stdbool.h>
#include <string.h>
#include "conv.h"

#ifdef WITH_AVX2
#include <immintrin.h>
#endif

#ifdef WITH_NEON
#include <arm_neon.h>
#endif

#ifdef _OPENMP
#include <omp.h>
#endif

MatrixData createMatrix(float w, float h, float c, float *d)
{
    MatrixData m;
    m.width = w;
    m.height = h;
    m.channels = c;
    m.data = d;
    return m;
}
bool vecAdd(float *p2, const float *p1, int n)
{
    int startIndex = 0;
#if defined(WITH_AVX2)
    __m256 a, c;
#pragma omp parallel for
    for (int i = 0; i < n - (n % 8); i += 8)
    {
        a = _mm256_load_ps(p1 + i);
        c = _mm256_load_ps(p2 + i);
        c = _mm256_add_ps(a, c);
        _mm256_store_ps(p2 + i, c);
    }
    startIndex = n - (n % 8);
#elif defined(WITH_NEON)
    float32x4_t a, b, c;
    for (int i = 0; i < n - (n % 4); i += 4)
    {
        a = vld1q_f32(p1 + i);
        b = vld1q_f32(p2 + i);
        c = vaddq_f32(a, b);
        vst1q_f32(p2 + i, c);
    }
    startIndex = n - (n % 4);

#endif
    for (int i = startIndex; i < n; i++)
    {

        p2[i] += p1[i];
    }
    return true;
}
void freeMatrix(MatrixData *m)
{
    if (m->data != NULL)
        free(m->data);
    m->width = m->height = m->channels = 0;
    m->data = NULL;
}
MatrixData convolution1d(MatrixData *inputMatrix, MatrixData *kernel, int start)
{

    int tempMatrixSize = kernel->width * kernel->height;
    float *tempMatrix;
    int outputWidth = inputMatrix->width - kernel->width + 1;
    int outputHeight = inputMatrix->height - kernel->height + 1;
    int outputSize = outputHeight * outputWidth;
    float *output;
    int ret1 = posix_memalign((void **)&output, 256, outputSize * sizeof(float));
    if (ret1 != 0)
    {
        // Handle error...
        printf("Memory aligned failed\n");
        return createMatrix(0, 0, 0, NULL);
    }
```

```
    int ret2 = posix_memalign((void **)&tempMatrix, 256, tempMatrixSize * sizeof(float));
    if (ret2 != 0)
    {
        // Handle error...
        printf("Memory aligned failed\n");
        return createMatrix(0, 0, 0, NULL);
    }
    int temp_matrix_index = 0;

    int output_matrix_index = 0;
    int firstInd = start;

    for (int k = 1; k <= outputHeight; start++)
    {
        // take the temp matrix

        for (int i = start; i < (inputMatrix->width * inputMatrix->height) + start; i += inputMatrix->width)
        {
            for (int j = i; j < kernel->width + i; j++)
            {

                tempMatrix[temp_matrix_index++] = inputMatrix->data[j];
            }

            if (temp_matrix_index == tempMatrixSize)
            {
                temp_matrix_index = 0;
                break;
            }
        }

        output[output_matrix_index++] = dotproduct(tempMatrix, kernel->data, tempMatrixSize);
        if (start == (inputMatrix->width * k) - kernel->width + firstInd)
        {

            start += kernel->width - 1;
            k++;
        }
    }

    free(tempMatrix);

    return createMatrix(outputWidth, outputHeight, 1, output);
}

MatrixData convolutionNd(MatrixData *inputMatrix, MatrixData *kernel)
{

    int outputWidth = inputMatrix->width - kernel->width + 1;
    int outputHeight = inputMatrix->height - kernel->height + 1;
    int outputSize = outputHeight * outputWidth;
    MatrixData temp;
    MatrixData output;

    int ret1 = posix_memalign((void **)&output.data, 256, outputSize * sizeof(float));
    if (ret1 != 0)
    {
        // Handle error...
        printf("Memory aligned failed\n");
        return createMatrix(0, 0, 0, NULL);
    }
    output.width = outputWidth;
    output.height = outputHeight;
    output.channels = 1;
    int oneMatrixsize = inputMatrix->height * inputMatrix->width;
    memset(output.data, 0, outputSize * sizeof(float));
    for (int i = 0, k = 0; i < inputMatrix->channels; i++, k += oneMatrixsize)
    {

        temp = convolution1d(inputMatrix, kernel, k);

        if (vecAdd(output.data, temp.data, outputSize) == false)
        {
            freeMatrix(&temp);
            return createMatrix(0, 0, 0, NULL);
        }
        freeMatrix(&temp);
    }
    return output;
}
```

```
bool relu(MatrixData *m)
{
    if (m->data == NULL)
    {
        printf("Input is not valid\n");
        return false;
    }
    int len = m->width * m->height * m->channels;
#if defined(WITH_AVX2)

    __m256 a, b;
    b = _mm256_setzero_ps(); // zeros
#pragma omp parallel for
    for (int i = 0; i < len; i += 8)
    {
        a = _mm256_load_ps(m->data + i);
        a = _mm256_max_ps(a, b);
        _mm256_store_ps(m->data + i, a);
    }

#elif defined(WITH_NEON)

    float32x4_t a, b;
    b = vdupq_n_f32(0.0f); // zeros
    for (int i = 0; i < len; i += 4)
    {
        a = vld1q_f32(m->data + i);
        a = vmaxq_f32(a, b);
        vst1q_f32(m->data + i, a);
    }
#else
    // printf("WITH_NORMAL\n");
    for (int i = 0; i < len; i++)
        m->data[i] = m->data[i] > 0 ? m->data[i] : 0;
#endif
    return true;
}

float dotproduct(const float *p1, const float *p2, int n)
{
    float sum = 0.0f;
    int startIndex = 0;
#if defined(WITH_AVX2)
    // printf("WITH_AVX2\n");

    __m256 a, b;
    __m256 c = _mm256_setzero_ps();

#pragma omp parallel for
    for (int i = 0; i < n - (n % 8); i += 8)
    {
        a = _mm256_load_ps(p1 + i);
        b = _mm256_load_ps(p2 + i);
        c = _mm256_add_ps(c, _mm256_mul_ps(a, b));
    }
    startIndex = n - (n % 8);
    c = _mm256_hadd_ps(c, c);
    c = _mm256_hadd_ps(c, c);
    sum = ((float *)&c)[0] + ((float *)&c)[4];
#elif defined(WITH_NEON)
    // printf("WITH_NEON\n");

    float32x4_t a,
        b;
    float32x4_t c = vdupq_n_f32(0);
    startIndex = n - (n % 4);
    for (size_t i = 0; i < n - (n % 4); i += 4)
    {
        a = vld1q_f32(p1 + i);
        b = vld1q_f32(p2 + i);
        c = vaddq_f32(c, vmulq_f32(a, b));
    }
    sum += vgetq_lane_f32(c, 0);
    sum += vgetq_lane_f32(c, 1);
    sum += vgetq_lane_f32(c, 2);
    sum += vgetq_lane_f32(c, 3);

    // printf("WITH_NORMAL\n");

#endif
```

```
    for (int i = startIndex; i < n; i++)
        sum += (p1[i] * p2[i]);
    return sum;
}

bool add_padding(MatrixData *m, int padding)
{
    int padded_width = m->width + 2 * padding;
    int padded_height = m->height + 2 * padding;
    int ch = m->channels;
    int padded_size = padded_width * padded_height * ch;
    float *padded;
    int ret1 = posix_memalign((void **)&padded, 256, padded_size * sizeof(float));

    if (ret1 != 0)
    {
        // Handle error...
        printf("Memory aligned failed\n");
        return 0;
    }
    memset(padded, 0, padded_size * sizeof(float));
    for (int c = 0; c < ch; c++)
    {
        for (int y = 0; y < m->height; y++)
        {
            for (int x = 0; x < m->width; x++)
            {
                int index = (c * m->height + y) * m->width + x;
                int padded_index = ((c * padded_height) + (y + padding)) * padded_width + (x + padding);
                padded[padded_index] = m->data[index];
            }
        }
    }
    freeMatrix(m);
    *m = createMatrix(padded_width, padded_height, ch, padded);

    return true;
}

void printMatrix(MatrixData *m)
{
    int mSize = m->width * m->height * m->channels;
    for (int i = 0; i < mSize; i++)
    {

        printf("%.2f ", m->data[i]);
        if ((i + 1) % m->width == 0)
        {
            printf("\n");
        }
    }
}
```

main.c

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include "conv.h"

#define TIME_START clock_t start = clock();
#define TIME_END                                                \
    clock_t end = clock();                                      \
    double time_taken = (double)(end - start) / CLOCKS_PER_SEC; \
    printf("Program took %f seconds to execute\n", time_taken);

int main(int argc, char const *argv[])
{

    int width, height, channels;
    // printf("input width, height, and the number channels of Matrix: \n");
    scanf("%d %d %d", &width, &height, &channels);
    float *matrix;
    int mSize = width * height * channels;

    int ret1 = posix_memalign((void **)&matrix, 256, mSize * sizeof(float));
```

```
    if (ret1 != 0)
    {
        // Handle error...
        printf("Memory aligned failed\n");
        return 0;
    }
    // printf("input the Matrix: \n");
    for (int i = 0; i < mSize; i++)
    {
        scanf("%f", &matrix[i]);
    }
    MatrixData originMatrix = createMatrix(width, height, channels, matrix);
    int kwidth, kheight, numofkernel;
    // printf("please input number of kernel \n");
    scanf("%d", &numofkernel);
    MatrixData *filterMatrix = (MatrixData *)malloc(sizeof(MatrixData) * numofkernel);
    // printf("input width and height of kernel with channels %d: \n", channels);
    scanf("%d %d", &kwidth, &kheight);
    int kSize = kwidth * kheight * channels;

    for (int i = 0; i < numofkernel; i++)
    {
        float *kernel;

        ret1 = posix_memalign((void **)&kernel, 256, kSize * sizeof(float));
        if (ret1 != 0)
        {
            // Handle error...
            printf("Memory aligned failed\n");
            return 0;
        }
        // printf("kernel-%d\n", i + 1);

        // printf("input the Kernel: \n");
        for (int i = 0; i < kSize; i++)
        {
            scanf("%f", &kernel[i]);
        }
        filterMatrix[i] = createMatrix(kwidth, kheight, channels, kernel);
        // free(kernel);
    }
    // after input matrix and kernel
    float *ouputData;

    ret1 = posix_memalign((void **)&ouputData, 256, (originMatrix.width - kwidth + 1) * (originMatrix.height - kheight + 1) * sizeof(float)
    if (ret1 != 0)
    {
        // Handle error...
        printf("Memory aligned failed\n");
        return 0;
    }
    MatrixData featureMap = createMatrix(originMatrix.width - kwidth + 1, originMatrix.height - kheight + 1, 1, ouputData);
    memset(ouputData, 0, (originMatrix.width - kwidth + 1) * (originMatrix.height - kheight + 1) * sizeof(float));
    MatrixData temp;

    // add_padding(&originMatrix, 10); //uncomment this part if you want to use padding the second argument is the number of padding you wa

    TIME_START // uncomment this part will give you the time execution
        for (int i = 0; i < numofkernel; i++)
    {
        temp = convolutionNd(&originMatrix, &filterMatrix[i]);

        vecAdd(featureMap.data, temp.data, temp.height * temp.width * temp.channels);
        freeMatrix(&temp);
    }
    TIME_END // uncomment this part will give you the time execution

        // relu(&featureMap); //uncomment this part if you want to use relu

        printMatrix(&featureMap);

    // REALEASE THE MEMORY
    for (int i = 0; i < numofkernel; i++)
    {
        free(filterMatrix[i].data);
    }
    freeMatrix(&originMatrix);
    free(filterMatrix);
    freeMatrix(&featureMap);
    return 0;
}
```

Part 3 - Result & Verification

Input format:

The first line contains three integers: the input matrix's **width, height, and channels**.

The next **width * height * channels** numbers represent the input data of the matrix.

After that, the input specifies the number of kernels, denoted by **X**.

On the next line, two integers specify the **kernel's width** and **height.**

For the following **X** lines, input the matrix of each kernel with a size of **kernel width * kernel height * channel**.

(**Note:** The channel of the kernel is following the channel of the input matrix.)

Test case #1:

input

```
5 5 1
3 3 2 1 0
0 0 1 3 1
3 1 2 2 3
2 0 0 2 2
2 0 0 0 1
1
3 3
0 1 2
2 2 0
0 1 2
```

output

```
12 12 17
10 17 19
9 6 14
```

Test case #2:

input

```
3 3 3
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
3
3 3
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3
```

output

```
162
```

Test case #3:

input

7 12 1
174.61136 -171.30317 -13.69410 192.83274 210.87194 -166.49262 29.56846
234.98650 175.06305 42.55915 -22.74195 18.35557 -158.76088 113.59924
-64.31511 -148.61469 -59.16994 137.95201 -123.37056 -191.38510 -73.87436
248.91074 -99.77266 -127.95663 76.24206 164.40580 147.22900 -92.06523
-102.65283 238.20587 -115.31792 172.26555 239.84912 -56.81348 -3.90170
-98.72710 -187.69873 19.88529 231.99691 50.00961 9.98607 -187.31774
33.97372 -73.22712 -250.56467 -236.63036 -241.01910 84.93744 199.23885
191.70495 -19.44642 44.58766 120.05381 -37.92156 -90.75112 78.53444
223.29366 1.21825 46.67164 211.39595 -214.75542 -130.69391 254.55048
-1.16159 -183.71336 -51.57398 244.19721 -243.57829 -90.69351 -191.69700
57.99808 132.60660 181.70882 119.77036 -228.23715 173.72104 174.44084
-162.19071 -118.96106 -210.85550 -182.99310 249.12274 -205.06105 -118.84408
1
5 5
3 2 1 4 32
34 45 223 54 23
234 55 23 5 23 6
234 4 2 34 5 3
4 2 5 4 3

output

-9832.29 -67744.34 3301.55
101013.05 -42976.98 -15187.18
-72802.45 69096.92 72551.24
-53813.21 -36777.17 15788.12
2544.17 20910.34 -22080.71
-36478.02 -73116.86 4032.07
11268.71 14747.92 74160.66
31887.17 35020.78 -11738.94

Test case #4:

input

3 3 1
4 4 4
4 4 4
4 4 4
1
1 1
5

output

20 20 20
20 20 20
20 20 20

Explanation for test case #2

3 3 3 → matrix **width, height and channels**
1 1 1 1 1 1 1 1 1 → 1st channel
1 1 1 1 1 1 1 1 1 → 2nd channel (matrix data with the size of **matrix width*_height_*channels)**
1 1 1 1 1 1 1 1 1 → 3rd channel
3 → number of kernels

3 3 → **kernel width and height**
1 1 1 1 1 1 1 1 1 → 1st channel
1 1 1 1 1 1 1 1 1 → 2nd channel **(first kernel)**
1 1 1 1 1 1 1 1 1 → 3rd channel
2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 **(second kernel)**
2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 **(third kernel)**
3 3 3 3 3 3 3 3 3

Result in terminal

```
3 3 3
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
3
3 3
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3
Program took 0.000149 seconds to execute
162.00
```

I added a time execution to compare between the normal way to compute convolutional layer versus using SIMD and OMP for the dot product and vec add.

I also added the **Relu function** you can use if you inside main.c (just uncomment it).

```
TIME_END // uncomment this part will give you the time execution

    // relu(&featureMap); //uncomment this part if you want to use relu

    printMatrix(&featureMap);

// REALEASE THE MEMORY
```

And there also addpadding if you need to add padding before the computation. (inside main.c)

```
// add_padding(&originMatrix, 10); //uncomment this part if you want to use padding the second argument is the number of padding you want to add

TIME_START // uncomment this part will give you the time execution
    for (int i = 0; i < numofkernel; i++)
```

To compile using AVX2 we can do this **gcc *.c -mavx2 -DWITH_AVX2**, if you use macos you can compile using **gcc *.c -mfpu=neon -DWITH_NEON** (I cannot confirm if this is the correct way to compile using NEON since I haven't tried it by my self).

But to be honest, because my operating system is using **windows,** I can't try compile using **WITH_NEON.**

for the sake of comparison, I generated a test case of a matrix with 54*32*1 and one kernel 16*16
using **SIMD and OMP** it took time **0.002208** seconds**.** Next the same test case but without optimization, it took time **0.002561** seconds.

So using **SIMD and OMP** is **1.16** times faster than the normal one.

for the small number size of kernel like 1x1,3x3,5x5 the result is most likely the same or even without optimization is faster. Because when using SIMD and OMP we need to make the size of the array we want to compute is multiple of 8 for AVX2 and a multiple of 4 for NEON so if it is less than those numbers then the computation will be done using normal computation. for a small size of array doing dot product is even faster or likely the same as using optimization. But when the matrix size let say 2000000, SIMD and OMP will be much faster than the normal way.

The last thing, I've tried to compile using **-fsanitize=address** flag and AddressSanitizer didn't detect any memory errors. So I think all allocated memory has been freed.

Part 4 - Difficulties & Solutions

▼ Using SIMD when the array size is not multiple of 8. I encounter this using modulo of 8. So the result of the modulo will be using for loop.

▼ Take certain element in 1D array and form it to another 1D array to do the dot product with the kernel. I solve this by finding the pattern for jump to the certain array element that we want, since we know the size of 1 channel matrix.

▼ Free the memory when it's not used anymore. I've done this with the help of -**fsanitize=address** that give the report of memory leak.