CS205 C/C++ Program Design

Final Project Report

SID: 11910718 Name: 陆彦青

Part 0. Introduction & Features

1. This program implements the forward part of a **convolutional neural network (CNN)** using C/C++. The pretrained model provided in `https://github.com/ShiqiYu/SimpleCNNbyCPP` is used to test the code, which can predict whether the input image is a person (upper body only) or not.
2. Theoretically, This network can support **any size of** convolutional kernel with `float` weight and bias, any **reasonable** stride and padding, any layers and any size of input image (the corresponding parameters need to be provided).
3. The project can be compiled in both x86-based MacOS and ARM-based Linux and the compilation process is controlled by CMake.
4. Using the provided model, the total cost of time to complete 3 convolutional layers and output the correct confidence scores can be less than **15 ms** in my computer (i5 CPU, 8 GB RAM) and **650 ms** in the EAIDK-310 board.
5. The essential part of this program - convolutional layer is implemented by **5 different** functions, detailed comparison and analysis are made in this report.

Part 1. Project Information

- Program Running Devices:
  - MacBook Pro 2019 (macOS 11.1, Intel Core i5 2.4GHz, 8 GB RAM)
  - EAIDK-310 (Fedora 28, ARM 1.3GHz, 1GB RAM)
- Development Environment:
  - IDE: CLion 2020.2.5
  - Compiler: Clang 12.0.0
- Source Code Repository:

  https://github.com/Beauchamp-West/2020Fall_CS205/tree/master/Project2
  - test_conv.cpp `//test program of the convolutional layer`
  - test_15samples.cpp `//test program of different input images`
  - cnn.h `//head file of all the functions`
  - cnn.cpp `//implementation of all the CNN functions`
  - CMakeList.txt
  - face_binary_cls.h `//define 2 structs to store the parameters`
  - face_binary_cls.cpp

Part 2. Output Test

Example Output

A demo is provided as an example to output the confidence scores of 2 sample images using the face-predict model. So I take the outputs of `demo.py` as the standard to test my program.

After installing PyTorch in my computer and modifying the codes, I run `demo.py` and get the outputs as follow:

```
PyTorch version: 1.7.1.
face.jpg: bg score: 0.007086, face score: 0.992914.
bg.jpg: bg score: 0.999996, face score: 0.000004.
```

These scores are the same as the outputs shown in README.md at the GitHub repository, which corresponds to the expectation. As we can see, the first score stands for the confidence of background and the second score stands for the face. The sum of them is 1 and more the value approaching 1 means more the possibility that the image is background or face. In my program below, these 2 scores are respectively represented by `score_0` and `score_1`.

Sample Images

Firstly, I use the 2 sample images to test the network. "-o3" is added as a compile options to speed up. In order to test whether the program is **steady** enough to output the same scores everytime, I apply the process of convolution to both `face.jpg` and `bg.jpg` for **10 times**. The screenshots of the result are as follows:

```
face & conv_p, score_0: 0.000000, score_1: 1.000000, duration = 15ms,
face & conv_p, score_0: 0.000000, score_1: 1.000000, duration = 14ms,
face & conv_p, score_0: 0.000000, score_1: 1.000000, duration = 15ms,
face & conv_p, score_0: 0.000000, score_1: 1.000000, duration = 14ms,
face & conv_p, score_0: 0.000000, score_1: 1.000000, duration = 15ms,
face & conv_p, score_0: 0.000000, score_1: 1.000000, duration = 15ms,
face & conv_p, score_0: 0.000000, score_1: 1.000000, duration = 15ms,
face & conv_p, score_0: 0.000000, score_1: 1.000000, duration = 15ms,
face & conv_p, score_0: 0.000000, score_1: 1.000000, duration = 19ms,
face & conv_p, score_0: 0.000000, score_1: 1.000000, duration = 17ms,
bg & conv_p, score_0: 1.000000, score_1: 0.000000, duration = 18ms,
bg & conv_p, score_0: 1.000000, score_1: 0.000000, duration = 16ms,
bg & conv_p, score_0: 1.000000, score_1: 0.000000, duration = 16ms,
bg & conv_p, score_0: 1.000000, score_1: 0.000000, duration = 15ms,
bg & conv_p, score_0: 1.000000, score_1: 0.000000, duration = 15ms,
bg & conv_p, score_0: 1.000000, score_1: 0.000000, duration = 15ms,
bg & conv_p, score_0: 1.000000, score_1: 0.000000, duration = 15ms,
bg & conv_p, score_0: 1.000000, score_1: 0.000000, duration = 15ms,
bg & conv_p, score_0: 1.000000, score_1: 0.000000, duration = 15ms,
bg & conv_p, score_0: 1.000000, score_1: 0.000000, duration = 16ms,
```

As shown above, `score_0` is always 0 and `score_1` is 1 for `face.jpg`, which perfectly corresponds to the expectation. The situation for `bg.jpg` is similar. After testing the sample images, the stability of this program seems not bad.

More Images

Besides the 2 sample images, we still want to know whether the program can take effect for other images. So I collect **15 different types of images** and convert their sizes into 128*128, which means the face-predict

model can be applied on them. I test all these images and get the following result:

In order to make the result seems more concise, I set the number of fractional digits as 2 in the following tests.

```
Asian_man_face, score_0: 0.00, score_1: 1.00, duration = 43ms
white_woman_face, score_0: 0.00, score_1: 1.00, duration = 34ms
white_woman_profile, score_0: 0.00, score_1: 1.00, duration = 33ms
Asian_woman_profile(small), score_0: 0.00, score_1: 1.00, duration = 26ms
partly_covered_face, score_0: 0.00, score_1: 1.00, duration = 20ms
black_woman_profile, score_0: 0.00, score_1: 1.00, duration = 22ms
old_man_face, score_0: 0.00, score_1: 1.00, duration = 19ms
Asian_man_profile, score_0: 0.08, score_1: 0.92, duration = 18ms
Asian_woman_face, score_0: 0.00, score_1: 1.00, duration = 18ms
black_man_face, score_0: 0.00, score_1: 1.00, duration = 19ms
cartoon_man_profile, score_0: 0.00, score_1: 1.00, duration = 18ms
mountain&river, score_0: 1.00, score_1: 0.00, duration = 19ms
sky&grassland, score_0: 1.00, score_1: 0.00, duration = 18ms
building, score_0: 1.00, score_1: 0.00, duration = 18ms
Apple_logo&aurora, score_0: 0.07, score_1: 0.93, duration = 18ms
```

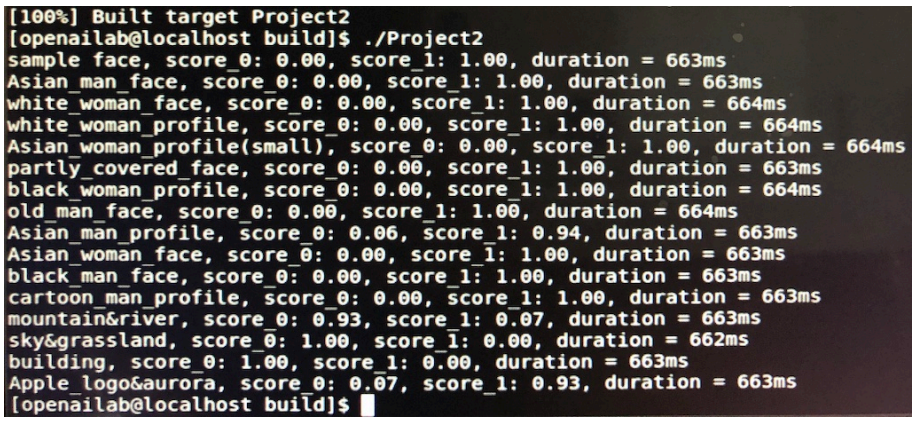
As we can see, the ability to distinguish face and background of this model is very **robust**. No matter the face belongs to white or black, female or male and it is profile or not, the confidence scores always corresponds to the real conditions. Even the **profile of an anime character** can be distinguished. The result just proves the program's precision.

The chart below shows the description and convolutional result of these 15 images. In addition, these images have been uploaded to my GitHub repository. You can check and use them if you want.

Item	Image	score_0	score_1	Success or not?
0	Asian man face	0.00	1.00	Y
1	white woman face	0.00	1.00	Y
2	white woman profile	0.00	1.00	Y
3	Asian woman profile	0.00	1.00	Y
4	partly covered face	0.00	1.00	Y
5	black woman profile	0.00	1.00	Y
6	old man face	0.00	1.00	Y
7	Asian man profile	0.08	0.92	Y
8	Asian woman face	0.00	1.00	Y
9	black man face	0.00	1.00	Y
10	anime man profile	0.00	1.00	Y
11	mountain and river	1.00	0.00	Y
12	sky and grassland	1.00	0.00	Y
13	building	1.00	0.00	Y
14	aurora	0.07	0.93	N

Test on ARM Development Board

The performance of this program on the ARM development board is an important point that we focus on. So I use all the images above as input samples to test whether the program can output the same scores running on the board.



Compared with the screenshot above, the scores are almost the same for all the 16 images. Thus we can conclude that **the platform do not affect the output of this program**. But due to the limit of hardware performance, the duration of running this program on the board increases averagely **30 times** compared with my computer.

Part 3. Main Functions Analysis & Improvement

Before we start the analysis of each function, we can see how much time is cost in **each part**:

```
face, conversion costs 1 ms,
convolution costs 14 ms, reLU costs 0 ms, max pooling costs 0 ms,
convolution costs 23 ms, reLU costs 0 ms, max pooling costs 0 ms,
convolution costs 3 ms, reLU costs 0 ms, max pooling costs 0 ms,
full connection costs 0 ms, softmax costs 0 ms, total duration = 43ms
```

As we can see, **convolution** occupies most of the cost time while other parts nearly have no effect on it. So the optimization of convolutional layer is the key point.

1. conv()

This function implements the **convolutional layer**, which convolve the input **tensor** (a 1 dimensional float array in this program) and pass the result (also a float array) to the next layer. Generally, this precess can be seen as **the addition of a series of dot products plus a factor (bias)**.

The objects of dot product are two **3 dimensional** (height,width,depth) vectors with `float` elements. The number of dot products depends on the size of convolutional kernel, stride, padding, the size of input array and the depth of the output array. Here is the complete code of the original version:

```
1  void Image::conv(int out_dep, int in_dep, int ker_size, int pad, int stride, const float * weight,
       const float * bias) {
2    auto * kernel = new float[ker_size*ker_size]; //卷积核
3    int out_h = (height+2*pad-ker_size)/stride+1;
4    int out_w = (width+2*pad-ker_size)/stride+1;
5    auto * out = new float[out_dep*out_h*out_w](); //输出的数组
6    int in_kr, in_kc, out_kr, out_kc, in_k;
7    int kernel_r, kernel_c, p;
8
9    for (int i = 0; i < out_dep; ++i) {
10       for (int j = 0; j < in_dep; ++j) {
11           for (int k = 0; k < ker_size*ker_size; ++k)
```

```
            kernel[k] = weight[i * (in_dep * ker_size * ker_size) + j * (ker_size * ker_size) + k
                ];
        for (int k = 0; k < out_h*out_w; ++k) {
            out_kr = k/out_w; //输出数组第i层第kr行
            out_kc = k%out_w;
            in_kr = out_kr*stride-pad;
            in_kc = out_kc*stride-pad;
            in_k = in_kr*width+in_kc; //输入数组第j层对应卷积部分首地址
            for (int l = 0; l < ker_size*ker_size; ++l) {
                kernel_r = l/ker_size; //卷积核第r层
                kernel_c = l%ker_size;
                p = kernel_r*width+kernel_c; //输入数组对应坐标(从卷积部分首地址算起)
                bool out_of_range = in_kr+kernel_r < 0 || in_kr+kernel_r >= width || in_kc+
                    kernel_c < 0 || in_kc+kernel_c >= width; //判断坐标是否在非填充区域外
                if (!out_of_range) //填充部分为0故不参与计算
                    out[i * out_w * out_h + k] += kernel[l] * data[j * height * width + in_k + p
                        ];
            }
        }
    }
    for (int k = 0; k < out_w*out_h; ++k) {
        out[i * out_w * out_h + k] += bias[i];
    }
  }

  delete [] kernel;
  delete [] data;
  data = out;
  height = out_h, width = out_w, depth = out_dep;
  length = height*width*depth;
}
```

The average duration of the **whole process** when involving this function is **20ms** in my computer (-o3 has been added). This number may be acceptable, but we still want to find some place that can be optimized.

Multithreading

Multithread is a common architecture used in programs which involves complex calculation. So here I try to use **OpenMP** but the result is NOT satisfying. The cost of time is over **200ms** which increases tenfold compared with the original version. It is because the **relativity** of each loop is strong and the **number of loops** is not so large. For example, many variables with the same name are used in different loop, the "+=" operator is used in the inner loop. Therefore, in order to avoid the potential conflicts, many blocks of code have to be "protected". It means that many threads have to wait for others while the switch of threads cost much more time than single-thread.

Cyclic Order

Since multithread is not very useful in this function, we should pay attention to the code itself. Firstly, we can notice that the loop of `ker_size*ker_size` appears **twice** and in the inner loop, `out[i*out_w*out_h + k]` has **no relation** with the loop itself. These 2 issues can be solved by **changing the order of cyclic layers** like this:

```
for (int i = 0; i < out_dep; ++i) {
    out_i = &out[i * out_w * out_h]; //输出矩阵第i层首地址
    bias_i = bias[i];
    for (int j = 0; j < in_dep; ++j) {
        data_j = &data[j * height * width]; // 输入矩阵第j层首地址
        for (int l = 0; l < ker_size*ker_size; ++l) {
            kernel_r = l/ker_size; //卷积核第r层
            kernel_c = l%ker_size;
            p = kernel_r*width+kernel_c; //输入矩阵对应坐标偏移量(从卷积部分首地址算起)
            kernel_l = weight[i * (in_dep * ker_size * ker_size) + j * (ker_size * ker_size) + l];
            for (int k = 0; k < out_h*out_w; ++k) {
                out_kr = k/out_w; //输出矩阵第i层第kr行
                out_kc = k%out_w;
                in_kr = out_kr*stride-pad;
                in_kc = out_kc*stride-pad;
                in_k = in_kr*width+in_kc; //输入矩阵第j层对应卷积部分首地址
                bool out_of_range = in_kr+kernel_r < 0 || in_kr+kernel_r >= width || in_kc+kernel_c
                     < 0 || in_kc+kernel_c >= width; //判断坐标是否在非填充区域外
                if (!out_of_range) //填充部分为0故不参与计算
                    out_i[k] += kernel_l * data_j[in_k + p];
            }
        }
    }
    ...
}
```

This new function is named `conv1()` and the order becomes `ijlk`. The loops look more **concise** and we expect it has some improvement in efficiency. Considering that the change is relatively **small**, I repeat the whole process **100 times** to compare the difference of duration.

```
(face & conv) * 100, duration = 2068ms,
(face & conv1) * 100, duration = 2072ms,
(face & conv) * 100, duration = 2085ms,
```

Although repeated 100 times, the difference is NOT obvious. So we can conclude that **changing the order will not improve the efficiency**.

Ternary Operator

Besides the order of loops, we can see that an **if** is used in the inner loop, which can determine whether the location of input array calculated by the location of output array is **out of range** (<0). Usually, the **ternary operator** has a better performance in efficiency compared with **if** in the same condition. Because the former will directly calculate **both of the expressions** divided by ':' but the latter will first verify the condition and then execute the expression.

After changing `if` with ternary operator, we get 2 new functions `conv2_1` and `cinv2_2`. Then repeating the whole process for 100 times using the 4 convolution functions respectively, we get:

```
(face & conv) * 100, duration = 2257ms,
(face & conv1) * 100, duration = 2503ms,
(face & conv2_1) * 100, duration = 2393ms,
(face & conv2_2) * 100, duration = 2419ms,
```

Apparently, the result cannot make us satisfied. In order to get **significant** improvement, we need to look at other directions.

Point-wise Storage

We know that the image with **more than 1 channel** can be stored in **2 forms** as a 1 dimensional array — **channel-wise** and **point-wise**. Since the given parameters are stored channel-wisely, we naturally use this form in our program. However, will these two forms have some difference in efficiency? The answer is **yes** and the difference is relatively **remarkable**. The **fifth** function of convolution which is based on the storage form of point-wise is as follows:

```
void Image::conv_p(int out_dep, int in_dep, int ker_size, int pad, int stride, float *weight, const
     float *bias) {
    float * kernel; //卷积核
    int out_h = (height+2*pad-ker_size)/stride+1;
    int out_w = (width+2*pad-ker_size)/stride+1;
    auto * out = new float[out_dep*out_h*out_w](); //输出的数组
    int in_r, in_c, out_r, out_c, in_l;
    int kernel_r, kernel_c, p;

    for (int i = 0; i < out_h*out_w; ++i) {
        out_r = i/out_w;
        out_c = i%out_w;
        in_r = out_r*stride-pad;
        in_c = out_c*stride-pad;
        for (int j = 0; j < out_dep; ++j) {
            for (int k = 0; k < ker_size*ker_size; ++k) {
                kernel = &weight[k*in_dep*out_dep + j*in_dep];
                kernel_r = k/ker_size; //卷积核第r层
                kernel_c = k%ker_size;
                p = kernel_r*width+kernel_c; //平面上坐标偏移量
                for (int l = 0; l < in_dep; ++l) {
                    in_l = (in_r*width+in_c+p) * in_dep + l; //原矩阵对应坐标
                    bool out_of_range = in_r+kernel_r < 0 || in_r+kernel_r >= width
                                        || in_c+kernel_c < 0 || in_c+kernel_c >= width; //判断坐标
                                            是否在非填充区域外
                    out[i*out_dep+j] += out_of_range ? 0 : kernel[l]*data[in_l];
                }
            }
            out[i*out_dep+j] += bias[j];
        }
    }
    ... //the rest part is the same as the original one
}
```

After the test, using this function can **improve** the efficiency **over 150%**. Actually, this time includes the conversion of channel-wise weights so if the weights are given point-wise, it can be faster.

```
(face & conv) * 100, duration = 2256ms,
(face & conv1) * 100, duration = 2347ms,
(face & conv2_1) * 100, duration = 2347ms,
(face & conv2_2) * 100, duration = 2343ms,
(face & conv_p) * 100, duration = 857ms,
```

In the end, I use a chart to summary the different functions and the corresponding efficiency:

Name	conv	conv1	conv2_1	conv2_2	conv_p
Duration (100 times) / ms	2256	2347	2347	2343	857

2. reLU()

This is a common **nonlinear activation function** used in CNN which can prevent the "over-fitting" of the data. It is a very simple function so I do not show it here.

3. max_pool()

This function chooses the **maximum** of a 2*2 block and passes this value to the next layer. It loses part of the information but reserves the basic features of an image while reducing the dimension.

There are 2 max_pooling functions for channel-wise and point-wise respectively.

4. fc()

This function is essentially a **matrix multiplication** of an n*length matrix (weights) and a length*1 matrix (input) plus some bias. It connects each individual layers together and then output some scores, each of which represents the **possibility** that the image belongs to a specific type.

Similar to the max pooling, it has 2 version corresponding to different forms of storage.

5. softmax()

This function aims to **normalize** the output scores in the range [0,1] and guarantee their **sum is 1**. The ordinary softmax function looks like this:

Suppose there are n scores $x_1, x_2, \cdots, x_n$ and the corresponding confidence scores are $p_1, p_2, \cdots, p_n$. Then

$$p_i = f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}, i = 1, 2, \cdots, n$$

This formula has the risk of **overflow** or **underflow**. On the one hand, the exponential function increases **very fast** so if x_i is big enough, e^{x_i} will overflow. On the other hand, if every x_i is small enough, the denominator may be rounded to **0** and lead to the underflow. The solution used in this program is to modify the function like this:

Put $M = max(x_1, x_2, \cdots, x_n)$, then

$$p_i = g(x_i) = f(x_i - M) = \frac{e^{(x_i - M)}}{\sum_{j=1}^{n} e^{(x_j - M)}}, i = 1, 2, \cdots, n$$

Using this formula, the result will not change but both overflow and underflow can be **avoided** because the max value of e^{x_i} becomes 1 and the denominator is **at least** 1.

The implementation of this function is also not complex:

```cpp
void Image::softmax() {
    float sum = 0, max = 0;
    for (int i = 0; i < length; ++i) {
        max = max > data[i] ? max : data[i];
    }
    for (int i = 0; i < length; ++i) {
        sum += exp(data[i]-max);
    }
    for (int i = 0; i < length; ++i) {
        data[i] = exp(data[i]-max)/sum; //防止上下溢出
    }
}
```

6. forward()

This function **packages** the forward part of CNN, which includes loading the parameters, convolution, reLU, max pooling, fully-connection and softmax. There are 5 `forward()` functions corresponding to 5 different convolutional functions.

Part 4. Other Functions & CMake

1. convert2float()

This function is used before the main part of convolution, which aims to convert the Mat of input image to a 1-dimension `float` array. The complete code is shown below.

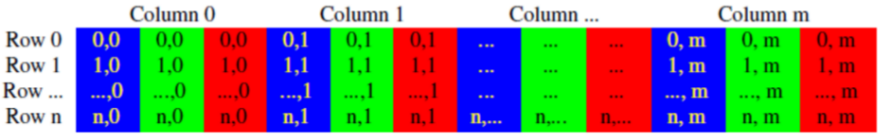
* Due to the limit of space, I only show the function that stores the data **channel-wisely**. The **point-wise** verison `convert2float_p()` can be checked out in my GitHub repository. Actually, that function is more simple.

```cpp
void convert2float(const Mat & image, float * data) {
    Mat out = image.clone(); //复制输入图像
    cvtColor(out, out, COLOR_BGR2RGB, 0); //颜色顺序转换
    out.convertTo(out, CV_32F, 1.0 / 255); //数据类型转换

    if (out.isContinuous()) { //判断图像元素是否在内存上连续
        int height = out.rows, width = out.cols, dep = out.channels();

        for (int i = 0; i < out.channels(); ++i) {
            for (int j = 0; j < height; ++j) {
                const auto *row = out.ptr<float>(j); //获取第j行数据的指针
                for (int k = 0; k < width; ++k)
                    data[i*height*width+j*width+k] = row[i+k*dep]; //channel-wise方式存储
            }
        }
    }
}
```

We know that in OpenCV, image is stored as `Mat` **point-wisely**. I find a figure from https://blog.csdn.net/koibiki/article/details/85954121 which can show the storage form clearly:



Firstly, `Mat` stores colors as `unchar` ([0,255]) in the order **BGR**. So we can use `cvtColor` and `convertTo` in OpenCV to convert the order into **RGB**, convert the type into `float` and normalize the value in the range [0,1].

Then, we need to iterate through the `Mat` to assign the values to the `float` array. `ptr<typename>()` and `at<typename>()` are 2 main functions to get the data of `Mat`. We use the **former** because the data here is **continous** and directly getting the pointer of a row is more efficient.

Actually, in the original version of this function, there was a **bug** that I did not notice for a long period of time. When the bug exists, the output of the total program is like this:

```
channels*rows*cols = 3*128*128
face & o3, score_0: 0.000000, score_1: 1.000000, duration = 21ms,
face & o3, score_0: 0.011270, score_1: 0.988730, duration = 20ms,
face & o3, score_0: 0.000000, score_1: 1.000000, duration = 20ms,
face & o3, score_0: 0.012208, score_1: 0.987792, duration = 19ms,
face & o3, score_0: 0.000000, score_1: 1.000000, duration = 19ms,
face & o3, score_0: 0.011270, score_1: 0.988730, duration = 18ms,
face & o3, score_0: 0.000000, score_1: 1.000000, duration = 18ms,
face & o3, score_0: 0.012548, score_1: 0.987452, duration = 18ms,
face & o3, score_0: 0.000000, score_1: 1.000000, duration = 18ms,
face & o3, score_0: 0.012548, score_1: 0.987452, duration = 19ms,
channels*rows*cols = 3*128*128
bg & o3, score_0: 1.000000, score_1: 0.000000, duration = 20ms,
bg & o3, score_0: 0.999998, score_1: 0.000002, duration = 22ms,
bg & o3, score_0: 1.000000, score_1: 0.000000, duration = 22ms,
bg & o3, score_0: 0.999998, score_1: 0.000002, duration = 19ms,
bg & o3, score_0: 1.000000, score_1: 0.000000, duration = 18ms,
bg & o3, score_0: 0.999998, score_1: 0.000002, duration = 18ms,
bg & o3, score_0: 1.000000, score_1: 0.000000, duration = 18ms,
bg & o3, score_0: 0.999998, score_1: 0.000002, duration = 18ms,
bg & o3, score_0: 1.000000, score_1: 0.000000, duration = 18ms,
bg & o3, score_0: 0.999998, score_1: 0.000002, duration = 18ms,
```

Among all these scores, the output values have some **slight difference**. Intuitively, I consider it as the accumulation of **round-off error** caused by repeated calculation of `float`. But later I recognize that this guess is virtually wrong because the codes that are repeatedly runned are always the same and in the same compile environment. Even though there exists some error, the error should be identical for each cycle. In order to see the difference more clearly, I temporarily delete the `softmax()` function to see the real output value. As shown below, the difference of real output value of each cycle is **far more larger** than that could be caused by round-off error.

```
face & o3, score_0: -9.558433, score_1: 9.861198, duration = 19ms,
face & o3, score_0: -2.161172, score_1: 2.313119, duration = 19ms,
face & o3, score_0: -9.558433, score_1: 9.861198, duration = 19ms,
face & o3, score_0: -2.101691, score_1: 2.263842, duration = 19ms,
face & o3, score_0: -9.558433, score_1: 9.861198, duration = 19ms,
face & o3, score_0: -2.161172, score_1: 2.313119, duration = 19ms,
face & o3, score_0: -9.563560, score_1: 9.870146, duration = 21ms,
face & o3, score_0: -2.161172, score_1: 2.313119, duration = 20ms,
face & o3, score_0: -9.558433, score_1: 9.861198, duration = 20ms,
face & o3, score_0: -2.161172, score_1: 2.313119, duration = 20ms,
```

After seeing these scores, the appearance of **alternate output values** according to different cycle gives me a **hint**. The function `cvtColor()` in OpenCV is used here to convert the default order of pixels color BGR to RGB which we need. But in the original version, I directly operate the input image as its **address** is passed into the function :

```
void convert2float(const Mat & image, float * data) {
    Mat out;
    cvtColor(image,image, COLOR_BGR2RGB,0); //image is directly modified
    image.convertTo(out, CV_32F, 1.0/255);
    ...
}
```

Even though the type is `const`, the compiler does not give me any error information. As a result, in the first loop, the image is converted to RGB so the output scores are correct. But in the next loop, the image is converted back to BGR so the scores deviate from the correct calue. And in the third loop the order becomes proper again. After knowing the cause, it is easy to fix this bug and the correct version has be presented at the begining of this part. However, it is relatively secret to find.

2. convert_order()

This function is used to convert the order of **convolutional weights** from channel-wise to point-wise.

3. max_4()

This function returns the maximum of 4 `float` values, which is used in max pooling.

4. Image::show()

This function prints the **confidence scores** of an image. The number of scores can be **arbitrary**.

5. CMakeList.txt

CMake is used to controll the compilation and add the support for **OpenCV** and **OpenMP** on both macOS and Linux, the complete code is as follows:

```
cmake_minimum_required(VERSION 3.14)
project(Project2)

set(CMAKE_CXX_STANDARD 11)

find_package(OpenCV REQUIRED)
find_package(OpenMP REQUIRED)

IF(APPLE)
    set(OPENMP_LIBRARIES "/usr/local/Cellar/libomp/11.0.0/lib")
    set(OPENMP_INCLUDES "/usr/local/Cellar/libomp/11.0.0/include")

    include_directories("${OPENMP_INCLUDES}")
    link_directories("${OPENMP_LIBRARIES}")

    set(CMAKE_CXX_FLAGS "-Xpreprocessor -fopenmp -lomp -Wno-unused-command-line-argument -O3")
ELSEIF(LINUX)
```

```
    set(CMAKE_CXX_FLAGS "-O3 ${OpenMP_CXX_FLAGS}")
ENDIF()

add_executable(Project2 main.cpp cnn.cpp)
target_link_libraries(Project2 ${OpenCV_LIBS})
```

The configuration on Linux is relatively simple since these required flags can automatically generate. But on macOS, the compiler **clang** do not support OpenMP very well so we have to add them manually.

Part 5. Key Point Summary

1. This program implements the **complete forward part of CNN**, the **stability** and **generality** of it is verified.
2. The result proves the **point-wise** storage form of data has a better performance than channel-wise in **efficiency**.
3. The overflow and underflow in `softmax()` can be prevented by calculating $f(x_i - M)$.