CS205 C/C++ Program Design

Project 2

Implement a CNN using C/C++

Name: 方唯可 Weike Fang **SID**: 22010055

Part 1. Analysis & Description

This final project asks us to implement a convolutional neural network (CNN) using C/C++, guided by https://github.com/ShiqiYu/SimpleCNNbyCPP where a pre-trained model (weights and biases) are provided. The model constitutes 3 convolution layers and 1 fully connected layer, with ReLU and Max Pooling in between. More details can be viewed on the link.

The basis structure of the source code is listed below (detailed source code can be viewed on GitHub):

The convert() method converts the data of the cv:Mat object from BGR to our desired RGB way of storing. convNN() method includes add padding (default to be 0) and do convolution, returning a float pointer as output matrix. max() and maxPooling() methods deal with the max-pooling process, and return a matrix down-sampled by 2. There are many more matmul() methods for matrix multiplication in the source code used for speed comparison. The fullyConnected() methods deal with the last fully connected layer.

```
float * convert(Mat image){}
float* convNN(float* weight, float * bias, float* matData, int stride, int kerSize, int
matSize, int in_channel, int out_channel, int padding=0){}
float max(float a, float b, float c, float d){}
float* maxPooling(float* matData, int row, int col, int channel){}
void matmul(int r1, int c, int c2, float* m1, float* m2, float* result)
float* fullyConnected(float* matData, float* weight, float* bias, int row, int col, int
channel, int out_feature){}
int main(){}
```

Analysis

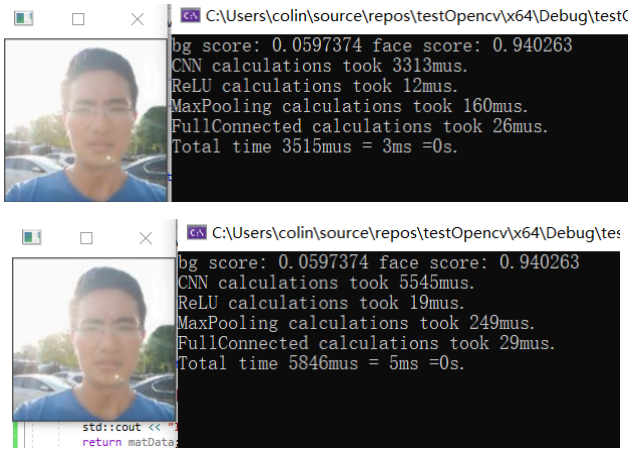
Tradeoff between speed and bug-free for users. Depending on usage scenario, we can favor either one of them. For example, in this project, the designer run the code himself/herself, so there is unlikely to be pathological using scenarios where bugs exist. Therefore, we can favor speed and get rid of many user-friendly qualifications and constraints.

In the example below, we wish the number of rows and columns to be even number when in use so that MaxPooling is executable. Since in a typical CNN model, MaxPooling is used several times, whether or not we add the if..else.. have a great effect on the speed. In this project

```
float max(float a, float b, float c, float d){} //returns the max among the four
float* maxPooling(float* matData, int row, int col, int channel)
{
    if (row % 2 != 0 || col % 2 != 0)
    {
        std::cout << "Invalid input, row, col should be even integers." << std::endl;
        return matData;
    }
    else
    {
        int n_matOut = row / 2 * col / 2 * channel;
        float * matOut = new float[n_matOut];
        for(int i = 0;i<row/2;i++)
```

```
            for(int j = 0;j<col/2;j++)
                for (int c = 0; c < channel; c++)
                {
                    matOut[i * col / 2 * channel + j * channel + c] =
                max(matData[2 * i * col * channel + 2 * j * channel + c],
            matData[2 * i * col * channel + (2 * j + 1) * channel + c],
        matData[(2 * i + 1) * col * channel + 2 * j * channel + c],
               matData[(2 * i + 1) * col * channel + (2 * j + 1) * channel + c]);
                }
        return matOut;
    }
}
```

Time analysis: if the if.. else.. statement is deleted, the runtime for maxpool is around **150 mus**. If it is kept, the runtime is around **250mus**. Such difference may become larger if larger neural network is implemented.



Part 2. Code

I hosted my source codes on GitHub and they can be viewed via https://github.com/ColinFang1009/sustechcpp/tree/master/project2

The finalProject.cpp files include all the needed weights/biases, functions, and implementation of the CNN model. I also put some images as test sets alongside the source code.

Part 3. Result & Verification

Some sample pictures (resized to 128*128) are used for testing the model (weights/bias & calculation correctness). Overall, the calculation appears to be correct, and the model can distinguish face and non-face in general.

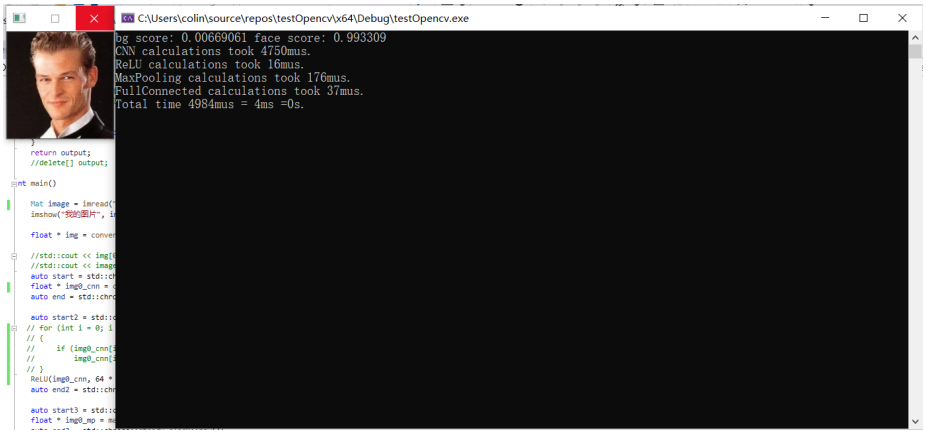
Test case #1: Some face images.

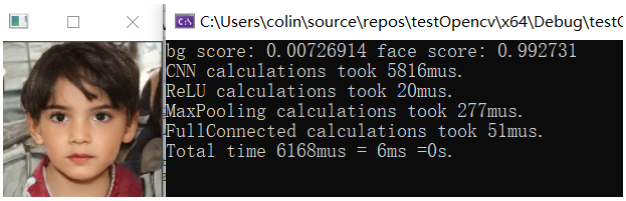
```
Input:
face_1.jpg, face2.jpg, face3.jpg, face4.png, face_lyx.png, face_light.jpg
Output:
face_1.jpg:    face score = 0.993309
face2.jpg:     face score = 0.999719
face3.jpg:     face score = 0.992731
face4.png:     face score = 0.940911                    //blurred screenshot from video
face_lyx.png:  face score = 0.997721
face_light.jpg:face score = 0.940263     // strong light in the background

$ python demo.py --img ./samples/face.jpg
bg score: 0.007086, face score: 0.992914.
```

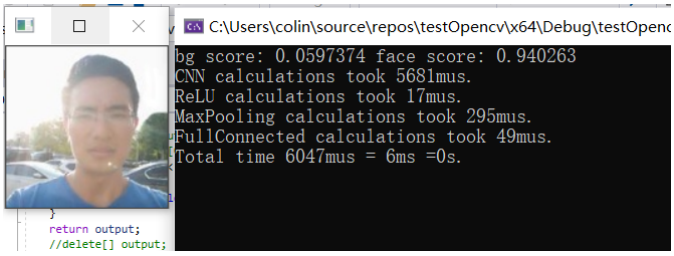
```
difference:0.000395
```

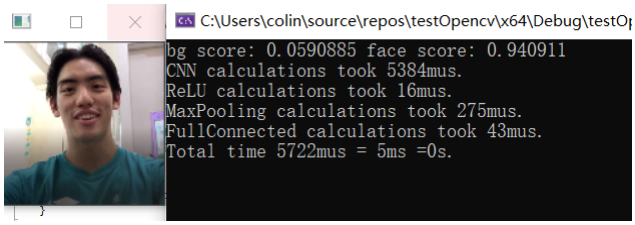
Screen-shot for case #1: As we can see from the above results, the model generally performs well, and the calculation proves to be correct. Of all the human faces, the model prediction for face score all exceeds 0.94, and most reaches 0.99, showing that the pre-trained model does a great job in identifying face images.

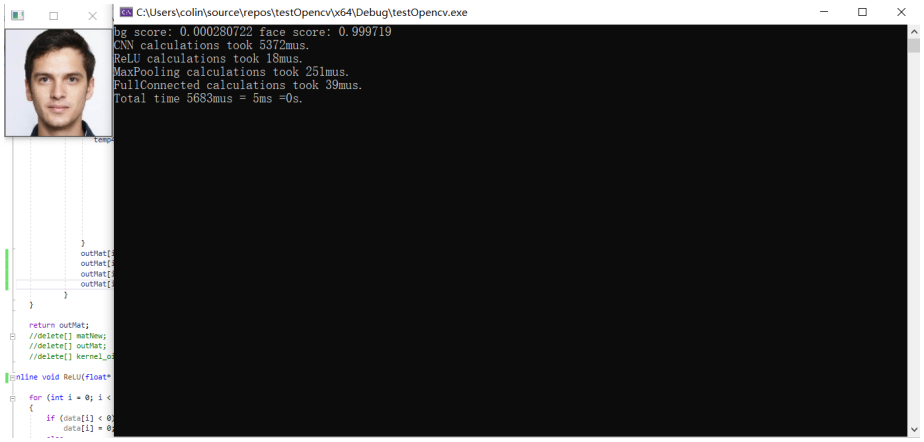
Observation: However, it performs less well in face4.png and face_light, the former being too blurred as screenshotted from a video, the latter being in the strong light.







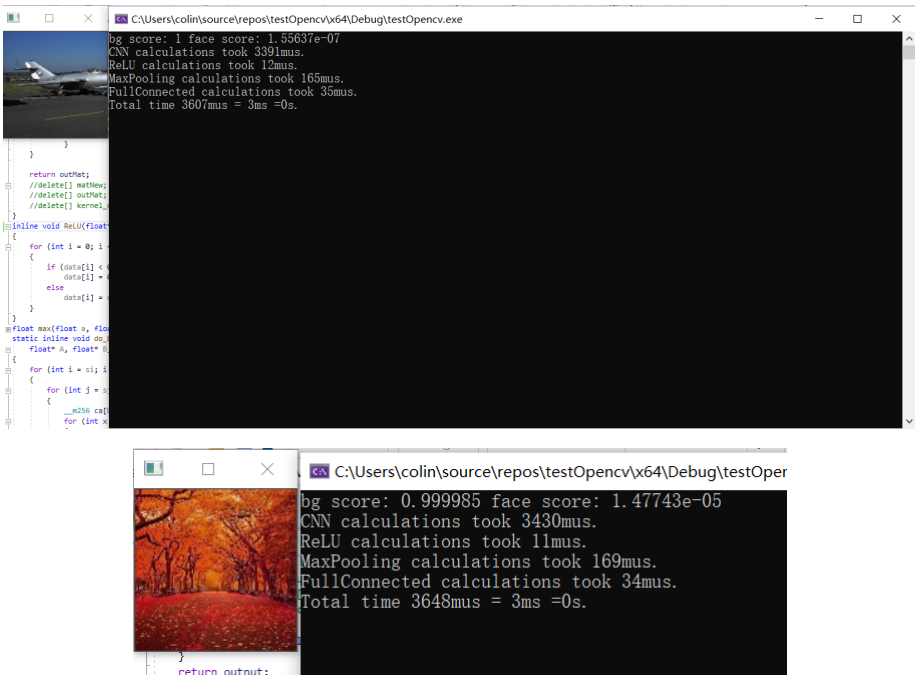




Test case #2: non-face images

```
Input:
bg.jpg, autumn.jpg
Output:
bg.jpg: bg score: 1
autumn.jpg: bg score: 0.999985


$ python demo.py --img ./samples/bg.jpg
bg score: 0.999996, face score: 0.000004.

    Difference: 3.84e^-6
```

Screen-shot for case #2: The two test samples show that the model is still predicting well for non-face background images, with both bg score exceeding 0.99 that matches the real case. Since the test set is relatively small, we cannot conclude that the model does a good job identifying non-face images. Indeed, as pointed out by my classmates, the model performs poorly when it is inputted with bright and colorful background images, probably because they are underrepresented in the training set/



Difference in results

Compared with Python, the results from C/C++ has minimal difference (0.000395 and 0.00000384)

Possible reasons for the difference:

1. Python uses PyTorch to implement the CNN, whereas we use float * as matrix to calculate. During the process, some precision is lost when doing arithmetics with floating-point numbers
2. When the model is outputted as .cpp file and floating-point numbers, some of the weights/biases might be truncated or rounded, thus leading to inaccuracy in calculation.
3. In my project implementation, sometimes copying and floating-point number comparison may lead to uncertainties and errors. For example, if the float has too many digit, comparison may not work and thus MaxPooling leads to fluctuations.

Test case #3: Example of using function template to calculate other types of data (for example, int, double, ...)

For future use to calculate CNN model, there might be other data type such as integers, double, etc. Therefore, a way to improve the code is to use template, and when called, users can specify the data type for calculation. An example is given below.

```
//Function Templates
```

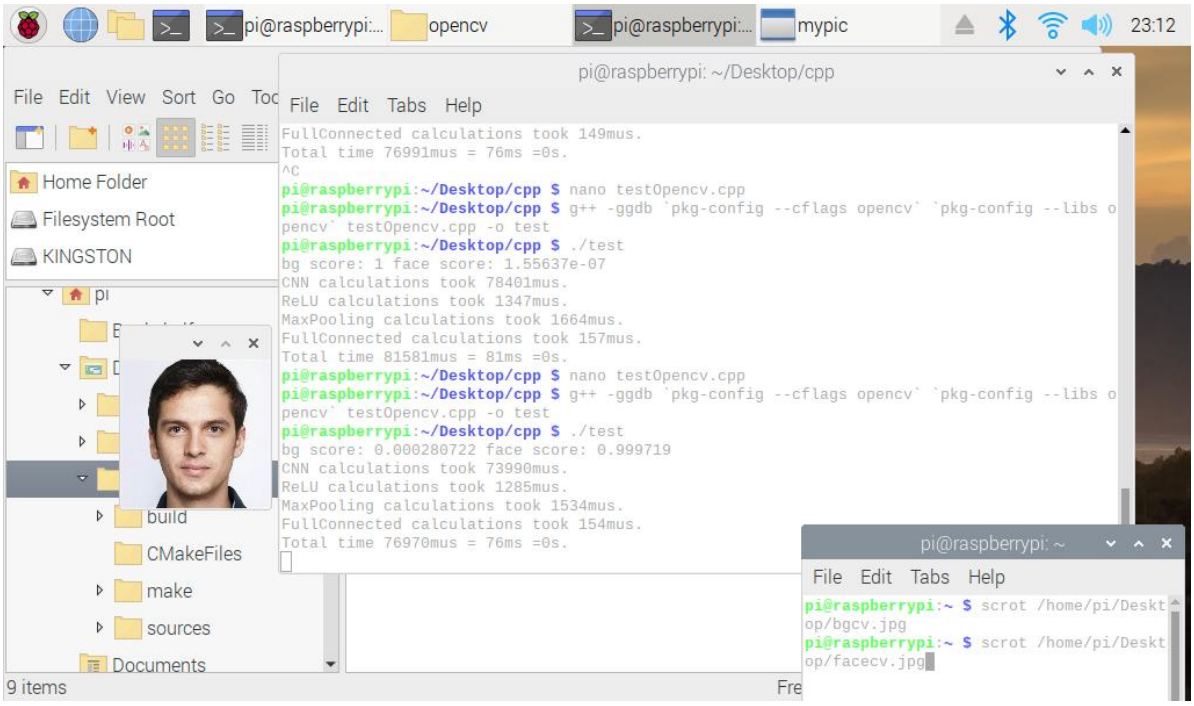
```
template <typename T>
T* convNN_tmplt(T* weight, T* bias, T* matData, int stride, int kerSize, int matSize,
int in_channel, int out_channel, int padding = 0)
{
    int sizeNew = matSize + padding * 2;
    int n = sizeNew * sizeNew * in_channel;
    T* matNew = new T[n];
    for (int i = 0; i < n; i++)
        matNew[i] = (T)0;
    int outSize = (sizeNew + 2 * padding - kerSize) / stride + 1; // take the floor
value if there is remainder
    if (padding > 0)
    {
        for (int i = 0; i < matSize; i++)
            for (int j = 0; j < matSize; j++)
                    for (int c = 0; c < in_channel; c++)
                     matNew[(i+1) * sizeNew * in_channel + (j+1) * in_channel + c] =
matData[i * matSize * in_channel + j * in_channel + c];

    }
    else
        matNew = matData;
    int n_outMat = outSize * outSize * out_channel;
    T* outMat = new T[n_outMat];
    for (int i = 0; i < n_outMat; i++)
        outMat[i] = (T)0;
    int n_kerneloi = kerSize * kerSize * in_channel;
    T* kernel_oi = new T[n_kerneloi];
    for (int o = 0; o < out_channel; ++o)
    {
        for (int i = 0; i < in_channel; ++i)
        {
            // weights
            kernel_oi[0 + i * 9] = weight[o * (in_channel * 9) + i * (9) + 0];
            kernel_oi[1 + i * 9] = weight[o * (in_channel * 9) + i * (9) + 1];
            kernel_oi[2 + i * 9] = weight[o * (in_channel * 9) + i * (9) + 2];
            kernel_oi[3 + i * 9] = weight[o * (in_channel * 9) + i * (9) + 3];
            kernel_oi[4 + i * 9] = weight[o * (in_channel * 9) + i * (9) + 4];
            kernel_oi[5 + i * 9] = weight[o * (in_channel * 9) + i * (9) + 5];
            kernel_oi[6 + i * 9] = weight[o * (in_channel * 9) + i * (9) + 6];
            kernel_oi[7 + i * 9] = weight[o * (in_channel * 9) + i * (9) + 7];
            kernel_oi[8 + i * 9] = weight[o * (in_channel * 9) + i * (9) + 8];
        }

        for (int i = 0; i < outSize; i++)
            for (int j = 0; j < outSize; j++)
            {

                for (int c = 0; c < in_channel; c++)
                {
                    outMat[i * outSize * out_channel + j * out_channel + o] +=
                        kernel_oi[0 + c * 9] * matNew[stride * i * sizeNew * in_channel
+ (stride * j) * in_channel + c] +
                       //...//
                        kernel_oi[8 + c * 9] * matNew[(stride * i + 2) * sizeNew *
in_channel + (stride * j + 2) * in_channel + c];
                }
                outMat[i * outSize * out_channel + j * out_channel + o] += bias[o];
            }
    }
    return outMat;
```

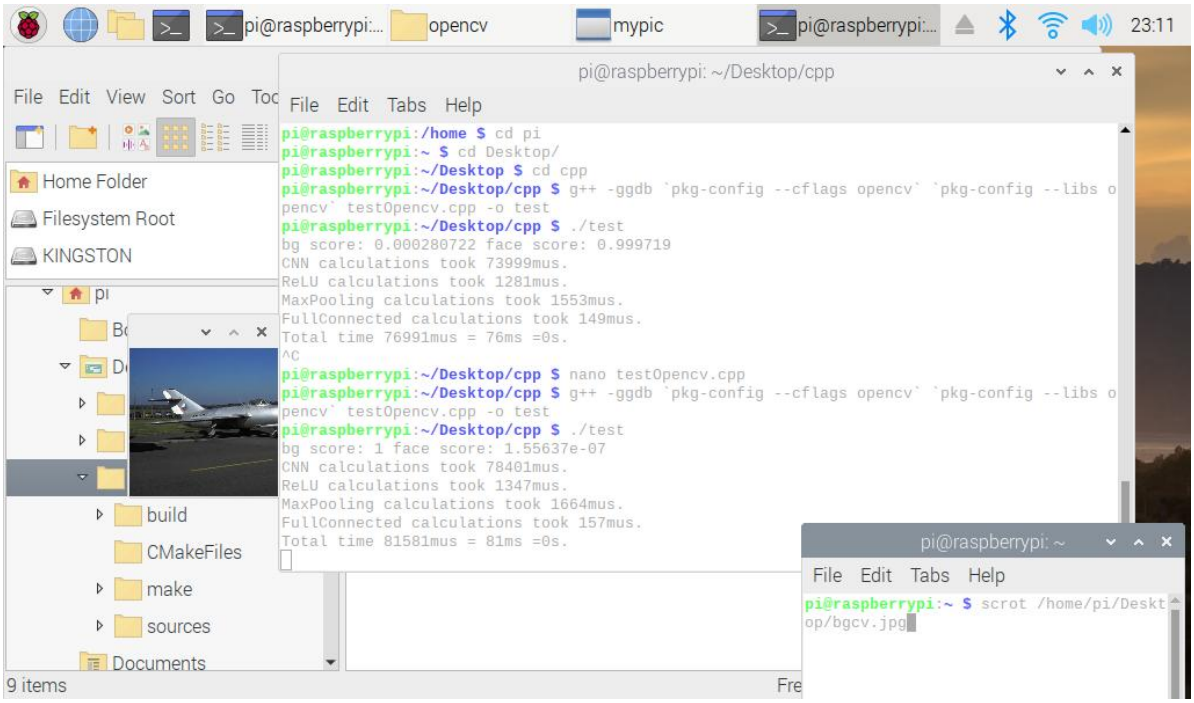
```
}

//Class template
template <typename T>
    class Matrix3D
    {
        public:
        T * data;
        int rows;
        int cols;
        int channels;
        float scale;
        Matrix3D();
        Matrix3D(int r, int c, int channels, T * d)
    }
```

Test case #4: Run on ARM Board (Raspberry Pi 4 Computer Model B 8GB RAM)

```
Compile: g++ -ggdb `pkg-config --cflags opencv` `pkg-config --libs opencv`
testOpencv.cpp -o test
Run: ./test
```

Screen-shot for case #4: Two pictures are run on ARM dev board, one face and one non-face. The resulting face/bg score matches the results from X64 PC. Time taken is a bit longer (before speed enhancement, ~36ms on PC).





Speed Optimization

After time analysis of different operations (graph below), I find that in this project, convolution layers (including adding padding) take the greatest amount of time (~95%). Therefore, wishing to optimize the speed of this particular model calculation, I put the majority of effort in improving the efficiency of the conNN() function. Still, I try to enhance the efficiency of other functions. Although they have minimal effect of improvement in this project, it's helpful for future applications when the model becomes greater.

The initial total time taken ranges from **35ms - 50 ms**. It's fast enough for small pictures (128*128) but may becomes much slower when the size of the picture increases.

Test case #5: Improved function compared with convNN_v1(). I get rid of the trivial details in the function below for better viewing. Here I break the outmost loop and calculate four out_channel at a time (so o += 4 for every loop). Such a trick works the best when accompanied with C/C++ optimization flags, as inspired from Project1.

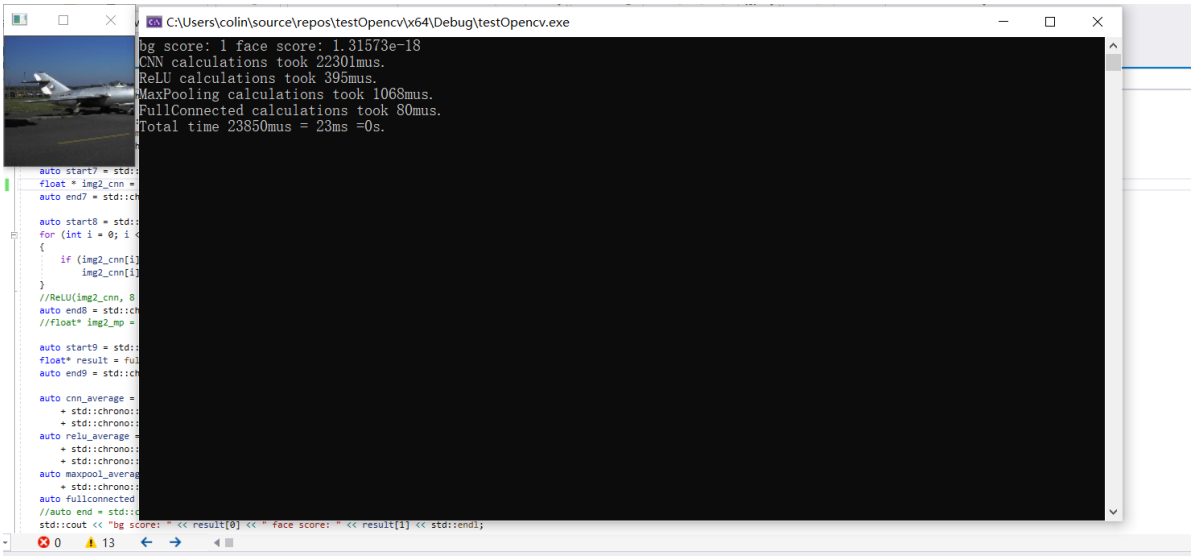
Also, I use float temp to store the intermediate result, avoiding accessing the matrix every time of the operation.

```
float* convNN(float* weight, float * bias, float* matData, int stride, int kerSize, int
matSize, int in_channel, int out_channel, int padding=0)
{
    int sizeNew = matSize + padding*2;
    int n = sizeNew * sizeNew * in_channel;
    float * matNew = new float[n];
    int outSize = (sizeNew + 2 * padding - kerSize) / stride + 1; // take the floor
value if there is remainder
    int n_outMat = outSize * outSize * out_channel;
```

```
    float * outMat = new float[n_outMat];
    //Padding
    for (int i = 0; i < n; i++)
        matNew[i] = 0.0f;
    if (padding > 0)
    {
        for(int i = 0;i < matSize;i++)
            for (int j = 0; j < matSize; j++)
                for (int c = 0; c < in_channel; c++)
                    matNew[(i+1) * sizeNew * in_channel + (j+1) * in_channel + c] =
matData[i * matSize * in_channel + j * in_channel + c];

    }
    else
        matNew = matData;


    for (int o = 0; o < out_channel; o+=4)
    {
        float temp1 = 0.0f;
        float temp2 = 0.0f;
        float temp3 = 0.0f;
        float temp4 = 0.0f;
        for (int i = 0; i < outSize; i++)
            for (int j = 0; j < outSize; j++)
            {
                temp1 = 0.0f;
                temp2 = 0.0f;
                temp4 = 0.0f;
                temp3 = 0.0f;
                for (int c = 0; c < in_channel; c++)
                {
                    temp1 +=
                 weight[o * (in_channel * 9) + c * 9 + 0] *  matNew[stride * i *
sizeNew * in_channel + stride * j * in_channel + c] +
                 weight[o * in_channel * 9 + c * 9 + 1] * matNew[stride * i * sizeNew *
in_channel + (stride * j + 1) * in_channel + c] +
                 weight[o * (in_channel * 9) + c * 9 + 2] * matNew[stride * i * sizeNew
* in_channel + (stride * j + 2) * in_channel + c] +
                 weight[o * (in_channel * 9) + c * 9+ 3] * matNew[(stride * i + 1) *
sizeNew * in_channel + stride * j * in_channel + c] +
                 weight[o * (in_channel * 9) + c * 9+ 4] * matNew[(stride * i + 1) *
sizeNew * in_channel + (stride * j + 1) * in_channel + c] +
                 weight[o * (in_channel * 9) + c * 9 + 5] * matNew[(stride * i + 1) *
sizeNew * in_channel + (stride * j + 2) * in_channel + c]+
                 weight[o * (in_channel * 9) + c * 9 + 6] * matNew[(stride * i + 2) *
sizeNew * in_channel + stride * j * in_channel + c] +
                 weight[o * (in_channel * 9) + c * 9 + 7] * matNew[(stride * i + 2) *
sizeNew * in_channel + (stride * j + 1) * in_channel + c]+
                 weight[o * (in_channel * 9) + c * 9) + 8] * matNew[(stride * i + 2) *
sizeNew * in_channel + (stride * j + 2) * in_channel +c];
                    temp2 +=;//save space
                    temp3 +=;
                   temp4 +=;
                }
                outMat[i * outSize * out_channel + j * out_channel + o] =  temp1 +
bias[o];
                outMat[i * outSize * out_channel + j * out_channel + o+1] = temp2 +
bias[o+1];
                outMat[i * outSize * out_channel + j * out_channel + o+2] = temp3 +
bias[o+2];
```

```
                outMat[i * outSize * out_channel + j * out_channel + o+3] = temp4 +
bias[o+3];
            }
    }
    return outMat;
}
```

Screen-shot for case #5: From this, the time taken reduces to **23 ms** (ranges from 20-30 ms per testing). The improvement is not significant in this case (since the calculation is small), but may turns out to be significant in larger scale. Moreover, the CNN calculation time reduces from 34935 mus to 22301 mus, roughly to its 2/3 time.



Test case #6: Using Compiling (optimization) flag, favor speed.

In Visual Studio, there are three types of optimization flags during compiling to maximize speed or size of the code. Specifically, -O1 optimizes the size of the code (smallest), -O2 optimizes the speed, and -Ox (-O3 in VS Code) takes care of both size and speed, but may loss some compatibility, as indicated in the Visual Studio Documentation.

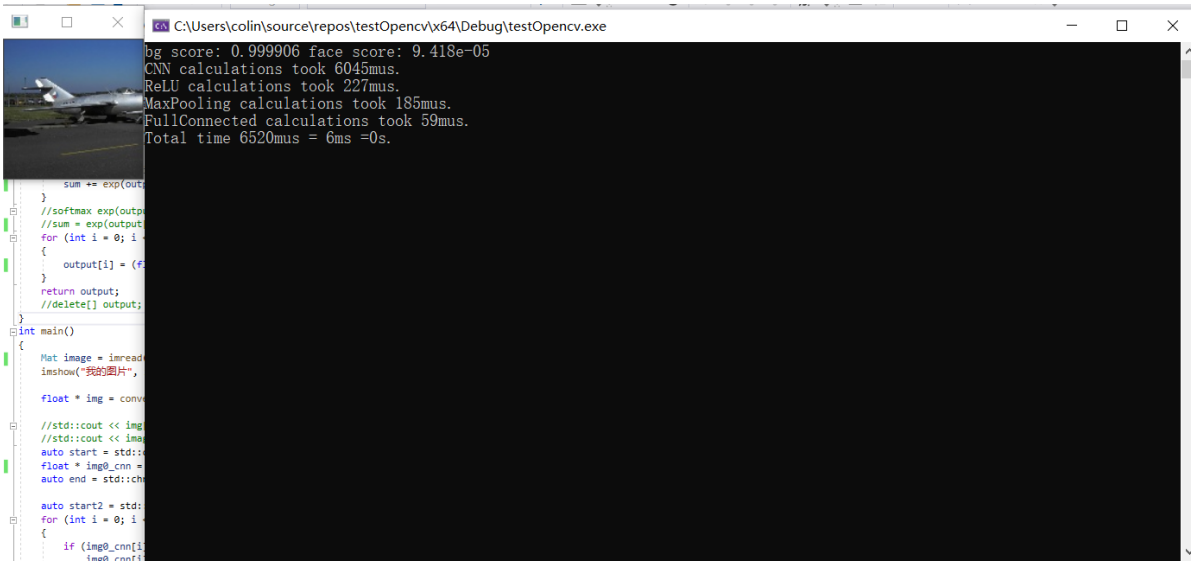
In this project, I choose /O2 to favor fast code because the size of the model is small.

```
In Visual Studio: Properties - C/C++ - Optimization, choose /O2 Maximum Optimization
(favor speed)
Total time for the convNN version 1: 6ms
Total time for the convNN version 2: 3ms
```

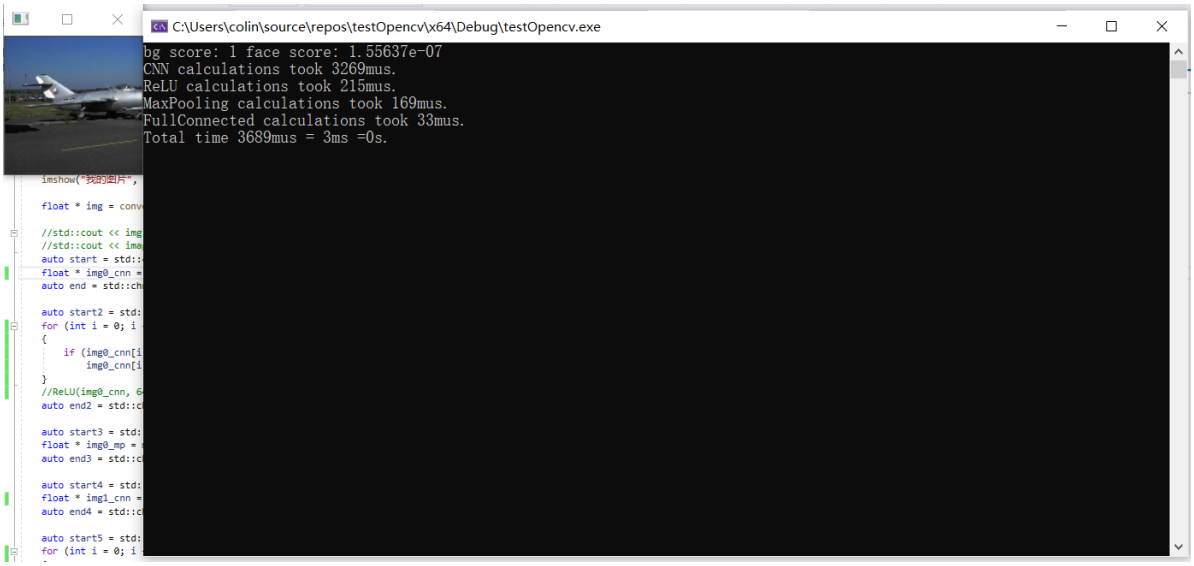
Screen-shot for case #6:

As we can see from the screenshot below, with the optimization flag, the total calculation time reduces to its **1/10**. Moreover, the version 2 convNN function performs better, taking only 3ms in total for the whole model.

Using /O2 optimization flag for the initial version.



Using /O2 optimization flag for the 2nd version of convNN() function.



Test case #7: Improving Fully Connected Layer by accelerating matrix multiplication

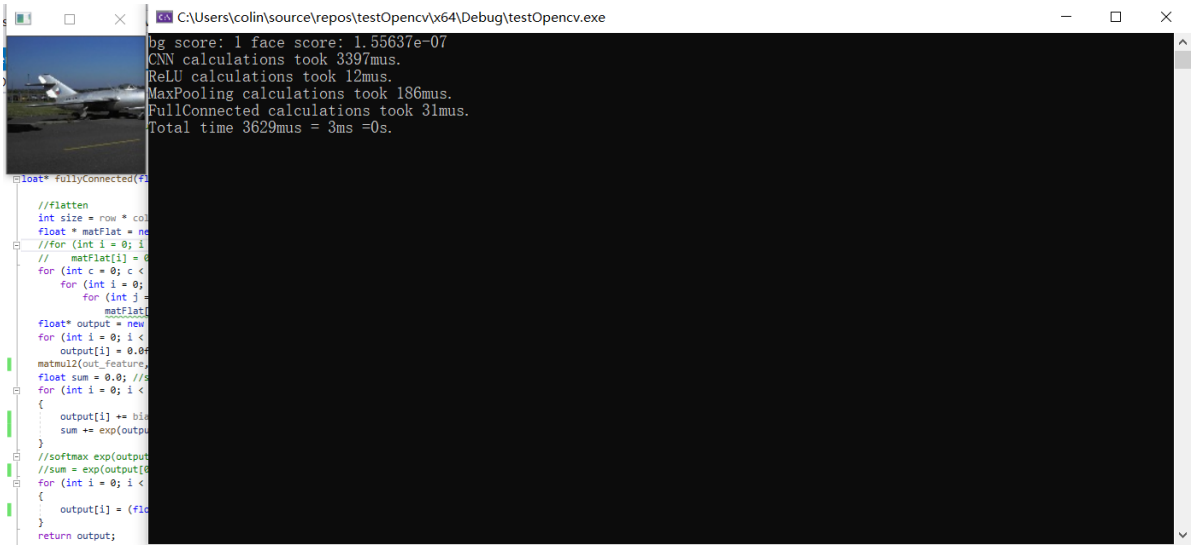
Inspired from Project1, I try using matrix multiplication to improve the Fully Connected Layer speed. Besides the brute force method, three imporved matmul is listed below. matmul2 simply switch the k-loop and j-loop to enable continuous access to memory. matmul3 function uses AVX Intrinsics to vectorize the calculation, and matmul6, in addition to using intrinsics, enable openmp parallel for multi-thread calculation.

However, in this case, the matrix multiplication size in the Fully Connected Layer is too small to show the effect. I cannot maximize the efficiency of the matmul3 and matmul6 functions since the size of the matrix is even smaller than the BLOCKSIZE / __m256 intrinsics. Therefore, for testing, I only test matmul2 in comparison with matmul1 (brute force); the others are left for matrix multiplication of larger scale in the future.

```
#define UNROLL 4     //4
#define BLOCKSIZE 64 //32
void matmul2(int r1, int c, int c2, float* m1, float* m2, float* result)
{
    //m1: r1*c, m2: c*c2, result: r1*c2
    float s;
    for (int i = 0; i < r1; ++i)
        for (int k = 0; k < c; ++k)
        {
            s = m1[i * c + k];
            for (int j = 0; j < c2; ++j)
```

```
                result[i * (c2)+j] += s * m2[k * (c2)+j];
        }
}
//Use AVX intrinsics
void matmul3(size_t r1, size_t c, size_t c2, float* m1, float* m2, float* result)
{
    //m1: r1*c, m2: c*c2, result: r1*c2
    //each _mm256 can store 8 float,(=8*4 byte = 256bit)
    for (size_t i = 0; i < r1; i++)
        for (size_t j = 0; j < c2; j += 8)
        {
            __m256 c0 = _mm256_load_ps(result + i * c2 + j); // c0 = Cij
            for (size_t k = 0; k < c; k++)
            {
                c0 = _mm256_add_ps(c0,
                    _mm256_mul_ps(_mm256_load_ps(m1 + i * c + k),
                        _mm256_broadcast_ss(m2 + k * c2 + j)));
            }
            _mm256_store_ps(result + i * c2 + j, c0); // Cij = c0
            ;
        }
}
static inline void do_block_f(size_t r1, size_t c, size_t c2, int si, int sj, int sk,
float* A, float* B, float* C);
//Moreover, use openmp parallel to enable multi-thread calculation
void matmul6(size_t r1, size_t c, size_t c2, float* m1, float* m2, float* result)
{
#pragma omp parallel for
    for (int sj = 0; sj < c2; sj += BLOCKSIZE)
        for (int si = 0; si < r1; si += BLOCKSIZE)
            for (int sk = 0; sk < c; sk += BLOCKSIZE)
                do_block_f(r1, c, c2, si, sj, sk, m1, m2, result);
}
```

Screen-shot for case #7: Under /O2 flag, the two matmul functions perform almost the same. While matmul1 takes (30-50 mus) to complete matrix calculation, matmul2 takes around **~30 mus**, quick enough for the layer.



Test case #8: Improving the speed of the ReLU part (tradeoff between modularization and speed)

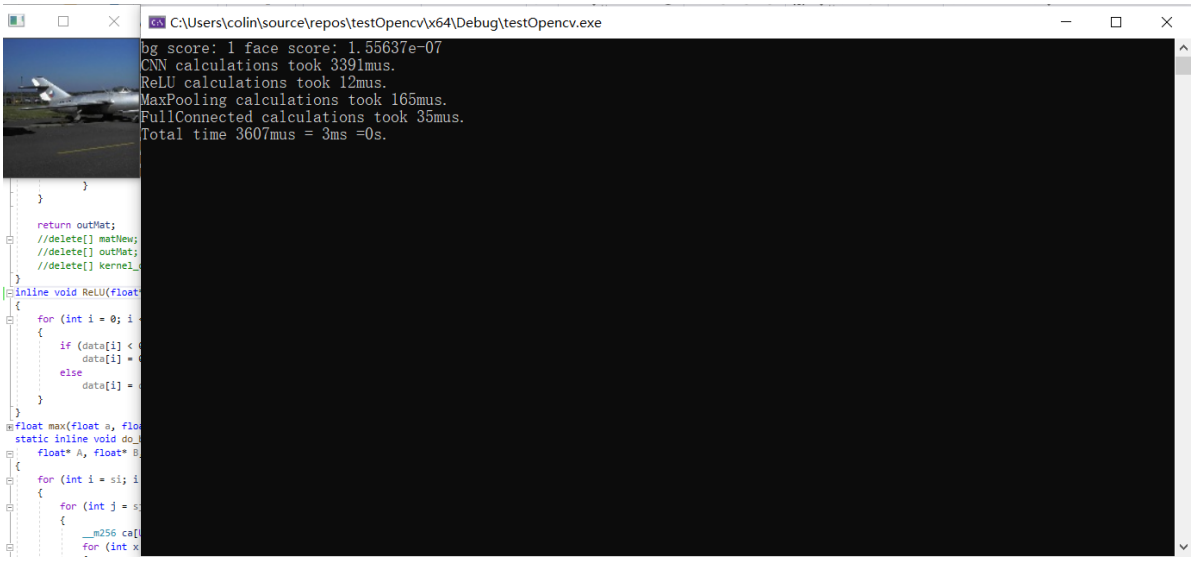
Since the function ReLU is very simple (only takes ~3 lines), removing the call of the function may enhance the speed of calculation.

```
inline void ReLU(float* data, int size)
```

```
{
    for (int i = 0; i < size; i++)
    {
        if (data[i] < 0)
            data[i] = 0;
        else
            data[i] = data[i];
    }
}
int main()
{
    //...Instead of calling the ReLU function, we can directly write the function
inline (or make the function an inline function)
    ReLU(img2_cnn, 8 * 8 * 32);
    //xxx
}
Time previously for ReLU specifically: 400-499 mus;
After using inline: 12 mus;
```

Screen-shot for case #8: Use inline for ReLU function

In C/C++, in order to deal with the problem of frequently calling small function that takes up memory in the stack, we use inline to describe / qualify the function. Then the compiler will work its magic to optimize the memory and speed. In this case, since ReLU function is small, I use inline and reduces the speed of ReLU calculation by a lot (though small effect on the total time).



Part 4. Difficulties & Solutions

Difficulty 1: Image cannot be seen since it's hidden at the corner and cannot be moved.

Solution: 1: Change the cv::imshow() function parameter to set its position.

Difficulty 2: Using compiling optimization flag in Visual Studio. In previous project and assignment, I use VS Code and type command with optimization flags like -O3 in the terminal. In this project, I use Visual Studio since it's more convenient for OpenCV.

Solution: Following the online tutorial on how to set optimization in Visual Studio (source: https://blog.csdn.net/weixin_42929607/article/details/106248798).

Difficulty 3: When adding /O2 optimization flag, the code went wrong as /O2 is incompatible with /RTC1.

Solution: In property page - C/C++ - Code generation page, set the "Basic Runtime checks" to default, and the problem is solved.

Difficulty 4: Setting up Cmake and OpenCV on ARM Dev board (Raspberrt Pi). I encountered a lot of difficulties during the process, but gladly they are all solved.

Solution: I followed several online tutorials to install and solve problems encountered during the installation process. Sources:https://blog.csdn.net/luotuo44/article/details/8909258 | https://blog.csdn.net/PecoHe/article/details/97476135 | https://blog.csdn.net/yzk2356911358/article/details/109119265

Difficulty 5: While the other functions are straightforward, I was stuck at the ConvNN() function at first, the one that multiplies entry-wise using 3*3 kernel.

Solution: In my first attempt, I let the out_channel be the outmost for loop. Then, I extract the kernel data as instructed by the SimpleCNN README. Then, I calculate for each entry of the output matrix by multiplying the matrix with the kernel entrywisely. As for speed enhancement, I tried the methods discussed in test case # 5 and 6.