Project4 Report

前言

本次Project需要设计一个用于CNN的class，可以存储Input、output以及kernel等多维矩阵，并且要方便用户使用，overload一些常用的operator。还要满足系统的泛用性，因此需要支持多种数据类型，还需要管理好内存空间，避免内存泄漏。在通读这些要求之后，我写了一个matrix.h的文件，里面包含了上述功能，并且测试了matrix.h的准确性，还在X86和ARM平台上都进行了实验，完成了本项目。

代码部分（未检查内存版）

我使用了一个matrix.h的文件将所有的定义全部写在里面，然后使用main.cpp文件调用matrix.h里面的方法，来测试结果。

matrix.h里的class声明

我定义了一个模板类matrix，里面包含row、col、channel等基本参数、使用shared_ptr来储存数据，并且包含了关于矩阵运算和图片处理的相关方法。

```
template <typename T>
class matrix{
    public:
    int row;
    int col;
    int channel;
    shared_ptr<T> ptr;

    matrix();
    matrix(const char*);
    matrix(int, int, int, T);
    matrix<T>(matrix<T>&); //softcopy
    //赋值
    template<typename U>
    matrix<T>& operator=(const matrix<U>&); //不同类型hardcopy
    matrix<T>& operator=(const matrix<T>&); //同类型softcopy
    // template<typename U>
    // matrix<T>& operator=(U);
    //相等
    template<typename U>
    bool operator==(const matrix<U>&);
    //加法
    template <typename U>
    matrix<T> operator+(const matrix<U>&);
    template <typename U>
    matrix<T> operator+(U);
    template <typename U>
    matrix<T>& operator+=(matrix<U>&);
    template <typename U>
    matrix<T>& operator+=(U);
    //减法
    template<typename U>
    matrix<T> operator-(const matrix<U>&);
    template<typename U>
    matrix<T> operator-(U);
    template<typename U>
    matrix<T>& operator-=(matrix<U>&);
    template<typename U>
    matrix<T>& operator-=(U);
    //乘法
    template<typename U>
    matrix<T> operator*(const matrix<U>&);  //矩阵点乘
    template<typename U>
    matrix<T> operator*(U);  //常数乘
    template<typename U>
    matrix<T>& operator*=(matrix<U>&);
    template<typename U>
    matrix<T>& operator*=(U);
    //除法
    template<typename U>
    matrix<T> operator/(const matrix<U>&);
    template<typename U>
```

```
    matrix<T> operator/(U);
    template<typename U>
    matrix<T>& operator/=(matrix<U>&);
    template<typename U>
    matrix<T>& operator/=(U);
    //矩阵叉乘
    template<typename U>
    matrix<T> mul(matrix<U>&);
    //矩阵卷积
    matrix<T> conv(matrix<T>&,int ,int);
    //图片处理
    matrix<T> B();
    matrix<T> G();
    matrix<T> R();
    void outputImage(const char*);
    //矩阵输出
    template <typename U>
    friend std::ostream& operator<<(std::ostream&,const matrix<U>&);
};
```

如上代码所示，首先我定义了template class来允许用户输入不同类型的参数，如matrix<int>、matrix<float>等等。然后，我定义了许多template的方法，输入类型与class的可以不同，这样就可以完成不同类型的数据之间的计算（满足课上说的要求）。

在.h文件中，我重载了加减乘除相等赋值等矩阵的基本运算、定义了四种矩阵初始化的方法、定义了矩阵输出的方法、还写了关于图片分RGB层、图片输出的方法。接下来我将具体阐释每一块代码的具体实现方式。

代码具体实现

构造器Constructor

在构造器部分我定义了四种构造方法，分别是：

- 无输入，定义空矩阵。
- 输入文件名，将图像传入。
- 输入矩阵channel、row、col、初始值，初始化矩阵。
- 输入一个同类型矩阵，复制该矩阵规模，并soft copy该矩阵数据。

```
//Constructor（无输入）
template <typename T>
matrix<T>::matrix()
{
    this->row = 0;
    this->col = 0;
    this->channel = 0;
    this->ptr = nullptr;
}

// Constructor（图片输入，结果按通道分开）
template <typename T>
matrix<T>::matrix(const char *filename)
{                                   // 这里也许可以用内存一致连续优化
    FILE *fp = fopen(filename, "rb"); // 打开文件。
    if (fp == NULL)
    { // 打开文件失败
        cout << "Error: open file failed!" << endl;
        this->row = 0; // 若打开失败，则令图像的行数为0，在主程序中判断行数是否为0来判断image初始化是否失败
    }
    // 得到图片长和宽，这里图片是一个像素3byte，分别是BGR
    int row, col;
    fseek(fp, 18, SEEK_CUR);
    fread(&col, sizeof(int), 1, fp);
    fread(&row, sizeof(int), 1, fp);
    this->row = row;
    this->col = col;
    this->channel = 3;
    // 得到图片大小 = 54(定义文件) + 1920x1080x3 = 6220854
    fseek(fp, 0, SEEK_END); // 到文档末尾
    int len = ftell(fp);
    T *data = new T[len - 54];                    // data中包含填充的0数据
    int gap = ((len - col * row * 3 - 54) / row); // 计算图片自动填充0的个数
```

```
    fseek(fp, 0, SEEK_SET);                         // 先把指针指回0
    fseek(fp, 54, SEEK_CUR);                        // 再加上54偏移，到达数据位
    fread(data, 1, (len - 54), fp);                 // 把图片里的数据全部导入data,len-54的数据量包含补0,3*row*col不包含补0
    // 将填充0前的数据导出
    T *real_data = new T[row * col * this->channel];
    int move = 0;
    for (int i = 0; i < len - 54; i++)
    {
        if ((i - move) % (3 * col) == (3 * col - 1))
        {
            real_data[i - move] = data[i];
            move = move + gap;
            i = i + gap;
            continue;
        }
        real_data[i - move] = data[i];
    }
    shared_ptr<T> pointer(new T[row * col * this->channel]);
    int size = this->row * this->col;
    // 将图层分开
    for (int k = 0; k < this->channel; k++)
    {
        for (int i = 0; i < size; i++)
        {
            pointer.get()[i + k * size] = real_data[i * this->channel + k];
        }
    }
    // memcpy(pointer.get(), real_data, row * col * this->channel); // 将数据拷贝到image中
    this->ptr = pointer;
    fclose(fp); // 清除文件指针
    delete[] data;
    delete[] real_data;
}

// Constructor矩阵初始化为常数
template <typename T>
matrix<T>::matrix(int channel, int row, int col, T val)
{
    if (row <= 0 || col <= 0 || channel <= 0)
    {
        cout << "Error: matrix size should be positive!" << endl;
    }
    this->row = row;
    this->col = col;
    this->channel = channel;
    shared_ptr<T> pointer(new T[row * col * channel]);
    for (int i = 0; i < row * col * channel; i++)
    {
        pointer.get()[i] = val;
    }
    this->ptr = pointer;
}

// Constructor矩阵初始化
template <typename T>
matrix<T>::matrix(matrix<T> &m)
{
    this->row = m.row;
    this->col = m.col;
    this->channel = m.channel;
    this->ptr = m.ptr; // soft copy
}
```

矩阵赋值

接下来，我定义了三种矩阵赋值的方式，分别是：

- 传入同类型矩阵，使用softcopy。
- 传入不同类型矩阵，使用hardcopy。
- 传入常数，然后将矩阵中的值全部设为该常数。

```
//矩阵赋值,softcopy
template<typename T>
matrix<T> &matrix<T>::operator=(const matrix<T> &m)
{
```

```
    this->channel = m.channel;
    this->row = m.row;
    this->col = m.col;
    this->ptr = m.ptr;
    return *this;
}

// 矩阵赋值,hardcopy
template <typename T>
template <typename U>
matrix<T> &matrix<T>::operator=(const matrix<U> &m)
{
    this->channel = m.channel;
    this->row = m.row;
    this->col = m.col;
    shared_ptr<T> pointer(new T[m.row * m.col * m.channel]);
        for(int i = 0; i < m.row * m.col * m.channel; i++){
            pointer.get()[i] = m.ptr.get()[i];
        }
    this->ptr = pointer;
    return *this;
}

// 矩阵赋值常数
template <typename T>
template <typename U>
matrix<T> &matrix<T>::operator=(U val)
{
    for (int i = 0; i < this->row * this->col * this->channel; i++)
    {
        this->ptr.get()[i] = val;
    }
    return *this;
}
```

在写hardcopy的时候我曾想过课上说的尽量使用memcpy（copy in c++）来做，然后我去网上找了各种资料。但是那些方法对于shared_ptr共享指针大多都很难使用。我还尝试了直接转换shared_ptr的类型的方法，但是按照官方的文档（如何：创建和使用 shared_ptr 实例 | Microsoft Learn）写出来的代码却怎么调也出错。尝试将memcpy和转换结合还是出错。在做了很多尝试无果之后只好作罢。

```
//错误代码
// this->ptr = dynamic_pointer_cast<T>(m.ptr);
```

判断相等的方法重载

重载相等判断的方法：如果矩阵的类型、行列通道数、数据有所不同则返回false，否则判断相同，返回true。

```
//矩阵相等判断
template<typename T>
template<typename U>
bool matrix<T>::operator==(const matrix<U> &m){
//类型不一致
    if(typeid(T).name() != typeid(U).name()) {
        return false;
    }
//行列通道数不一致
    else if(this->row != m.row || this->col != m.col || this->channel != m.channel){
        return false;
    }
//数值不一致
    for(int i = 0; i < this->row * this->col * this->channel; i++){
        if(this->ptr.get()[i] != m.ptr.get()[i]){
            return false;
        }
    }
    return true;
}
```

加减乘除的方法重载

在接下来的代码中，我重载了加减乘除以及自加自减自乘自除共八种operator。

我对于每种运算都定义了对矩阵的和对常数的操作方法，由于代码大量重复，展示仅展示加法部分。

```
// 矩阵加矩阵
template <typename T>
template <typename U>
matrix<T> matrix<T>::operator+(const matrix<U> &m)
{
    matrix<T> result(this->channel, this->row, this->col, 0);
    if (this->row != m.row || this->col != m.col != this->channel != m.channel)
    {
        cout << "Error: matrix size not match!" << endl;
        return result;
    }
    for (int i = 0; i < this->row * this->col * this->channel; i++)
    {
        result.ptr.get()[i] = m.ptr.get()[i] + this->ptr.get()[i];
    }
    return result;
}

// 矩阵加常数
template <typename T>
template <typename U>
matrix<T> matrix<T>::operator+(U val)
{
    matrix<T> result(this->channel, this->row, this->col, 0);
    for (int i = 0; i < this->row * this->col * this->channel; i++)
    {
        result.ptr.get()[i] = val + this->ptr.get()[i];
    }
    return result;
}

// 矩阵加等于矩阵
template <typename T>
template <typename U>
matrix<T>& matrix<T>::operator+=(matrix<U> &m)
{
    if (this->row != m.row || this->col != m.col != this->channel != m.channel)
    {
        cout << "Error: matrix size not match!" << endl;
        return *this;
    }
    for (int i = 0; i < this->row * this->col * this->channel; i++)
    {
        this->ptr.get()[i] += m.ptr.get()[i];
    }
    return *this;
}

//矩阵加等于常数
template <typename T>
template <typename U>
matrix<T>& matrix<T>::operator+=(U val)
{
    for (int i = 0; i < this->row * this->col * this->channel; i++)
    {
        this->ptr.get()[i] += val;
    }
    return *this;
}
```

矩阵叉乘

我定义了矩阵叉乘的方法，由于矩阵通道数可以不同，因此返回全部通道两两相乘的结果，因此结果通道数目为两个相乘矩阵的通道数目之积，运算的基本思路是前一个矩阵的每一行乘以后一个矩阵的每一列再加和。当前一个矩阵的行数和后一个矩阵的列数不一致时报错。

```
// 矩阵叉乘，结果channel数等于两个矩阵channel之积
template <typename T>
template <typename U>
matrix<T> matrix<T>::mul(matrix<U> &m)
{
    matrix<T> mat(this->channel * m.channel, this->row, m.col, 0);
    if (this->col != m.row)
```

```
    {
        cout << "mul Error: matrix col and row not match!" << endl;
        return mat;
    }
    int mul_size = this->col;
    mat.row = this->row;
    mat.col = m.col;
    mat.channel = this->channel * m.channel;
    cout << "mul size: " << mat.row << " " << mat.col << " " << mat.channel << endl;
    T sum = static_cast<T>(0);
    shared_ptr<T> pointer(new T[mat.row * mat.col * mat.channel]);
    for (int i = 0; i < this->channel; i++)//选择第一个矩阵的第i个通道
    {
        for (int j = 0; j < m.channel; j++)//选择第二个矩阵的第j个通道
        {
            for (int ii = 0; ii < mat.row; ii++)//选择第一个矩阵的第i行
            {
                for (int jj = 0; jj < mat.col; jj++)//选择第二个矩阵的第j列
                {
                    for (int s = 0; s < mul_size; s++)//第一个矩阵的第i行和第二个矩阵的第j列进行叉乘
                    {
                        sum += this->ptr.get()[i * this->col * this->row + ii * this->col + s] * m.ptr.get()[j * m.col * m.row + s * m.col
                        // cout << i * this->col * this->row + ii * this->col + s << " " << j * m.col * m.row + s * m.col + jj << endl;
                    }
                    pointer.get()[i * mat.col * mat.row * m.channel + j * mat.col * mat.row + ii * mat.col + jj] = sum;
                }
                sum = static_cast<T>(0);
            }
        }
    }
    mat.ptr = pointer;
    return mat;
}
```

图像（矩阵）卷积

本Project背景为适用CNN的matrix，因此我定义了矩阵卷积的方法。该方法需要传入kernel、stride、padding三个参数，要求kernel的channel数需要与待卷积矩阵的kernel数保持一致，而且kernel的size不能太大。卷积结果包含三个通道中的分别卷积的结果（即RGB数据分别保存，在这里没有求和）。

```
template <typename T>
matrix<T> matrix<T>::conv(matrix<T> &kernel, int stride, int padding)
{
    matrix<T> mat(this->channel, 1, 1, 0);
    if (this->row < kernel.row || this->col < kernel.col)
    {
        cout << "conv Error: kernel size is too small!" << endl;
        return mat;
    }
    if (this->channel != kernel.channel)
    {
        cout << "conv Error: matrix channel not match!" << endl;
        return mat;
    }
    // 卷积操作
    mat.row = (this->row + 2 * padding - kernel.row + 1) / stride;
    mat.col = (this->col + 2 * padding - kernel.col + 1) / stride;
    shared_ptr<T> pointer(new T[mat.row * mat.col * mat.channel]);
    mat.ptr = pointer;
    T sum = 0;
    int x, y;
    // 遍历output里的每一位数据
    for (int k = 0; k < kernel.channel; k++)
    {
        for (int i = 0; i < mat.row; i++)
        {
            for (int j = 0; j < mat.col; j++)
            {
                // 计算coved_data中每一位的数据，具体方式为对应位置相乘求和
                for (int ii = 0; ii < kernel.row; ii++)
                {
                    x = k * this->row * this->col - padding + stride * i + ii;
                    for (int jj = 0; jj < kernel.col; jj++)
                    {
                        y = k * this->row * this->col - padding + stride * j + jj;
                        if (x < 0 || x >= this->row || y < 0 || y >= this->col)
```

```
                        continue; // 判断数字是否取padding值，若是padding则0
                    sum += this->ptr.get()[y + x * this->col + k * this->col * this->row] * kernel.ptr.get()[jj + ii * kernel.col];
                }
            }
            mat.ptr.get()[k * mat.col * mat.row + i * mat.col + j] = sum;
            sum = 0;
        }
    }
}
return mat;
}
```

图像分层处理

在处理图像的时候难免会希望对RGB三种颜色分开进行处理，于是我写了方法能够提取出图像的某一层(hardcopy)。在内存中我定义图像数据按照channel,row,col顺序来储存的，因此提取图像某一层的方法比较简单，直接跳过几个channel读取row*col个数据即可。

```
// 提取图片图层,BGR
template <typename T>
matrix<T> matrix<T>::B()
{
    if(this->channel != 3)
    {
        cout << "B Error: channel not equal to 3!" << endl;
        return *this;
    }
    matrix<T> mat(1, this->row, this->col, 0);
    shared_ptr<T> pointer(new T[mat.row * mat.col]);
    memcpy(pointer.get(), this->ptr.get(), mat.row * mat.col * sizeof(T));
    mat.ptr = pointer;
    return mat;
}

template <typename T>
matrix<T> matrix<T>::G()
{
    if(this->channel != 3)
    {
        cout << "G Error: channel not equal to 3!" << endl;
        return *this;
    }
    matrix<T> mat(1, this->row, this->col, 0);
    shared_ptr<T> pointer(new T[mat.row * mat.col]);
    memcpy(pointer.get(), this->ptr.get() + mat.row * mat.col, mat.row * mat.col * sizeof(T));
    mat.ptr = pointer;
    return mat;
}

template <typename T>
matrix<T> matrix<T>::R()
{
    if(this->channel != 3)
    {
        cout << "R Error: channel not equal to 3!" << endl;
        return *this;
    }
    matrix<T> mat(1, this->row, this->col, 0);
    shared_ptr<T> pointer(new T[mat.row * mat.col]);
    memcpy(pointer.get(), this->ptr.get() + 2 * mat.row * mat.col, mat.row * mat.col * sizeof(T));
    mat.ptr = pointer;
    return mat;
}
```

输出图片的方法

如Project3一致，我自定义了输出图片的方法，该函数需要传入一个filename，即可把矩阵中的数据按照8位灰度图的方式输出。

```
//输出8位图片
template <typename T>
void matrix<T>::outputImage(const char *filename)
{
    //把图片再转成输出的格式
    shared_ptr<T> pointer(new T[this->row * this->col]);
    for(int i = 0; i < this->row * this->col; i++)
```

```
    {
        for(int j = 0; j < this->channel; j++)
        pointer.get()[i] += this->ptr.get()[i + j * this->row * this->col];
    }
    int row = this->row;
    // 每一行需要考虑补0，让每一行是四字节的整数倍
    cout << this->col << endl;
    int gap = (this->col % 4 == 0) ? 0 : 4 - (this->col % 4);
    int col = this->col + gap;
    int bmi[] = {col * row + 54 + 1024, 0, 54, 40, this->col, row, 1 | 8 << 16, 0, col * row, 0, 0, 256, 0};
    cout << "gap="<<gap <<endl;
    // 定义颜色表
    typedef struct _RGBQUAD
    {
        // public:
        byte rgbBlue;     // 蓝色分量
        byte rgbGreen;    // 绿色分量
        byte rgbRed;      // 红色分量
        byte rgbReserved; // 保留值,必须为0
    } RGBQUAD;
    RGBQUAD rgbquad[256];
    for (int p = 0; p < 256; p++)
    {
        rgbquad[p].rgbBlue = (byte)p;
        rgbquad[p].rgbGreen = (byte)p;
        rgbquad[p].rgbRed = (byte)p;
        rgbquad[p].rgbReserved = (byte)0;
    }
    FILE *fp = fopen(filename, "wb");
    if( fp == NULL){
        cout << "File open error!" << endl;
    }
    fprintf(fp, "BM");
    fwrite(&bmi, 52, 1, fp);
    fwrite(&rgbquad, 4, 256, fp);
    unsigned char *array = new unsigned char[col * row];
    for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < col; j++)
        {
            if (j >= col - gap)
            {
                array[i * col + j] = (byte)0;
                continue;
            }
            else
                array[i * col + j] = (byte)pointer.get()[j + this->col * i];
        }
    }
    fwrite(array, 1, row * col, fp);
    fclose(fp);
    delete[] array;
}
```

矩阵输出

定义矩阵输出：输出类型、行列数，然后按顺序输出每个channel的矩阵

```
template <typename T>
std::ostream &operator<<(std::ostream &out, const matrix<T> &a)
{
    out << typeid(T).name() << "; row = " << a.row << "col = " << a.col << endl;
    for (int s = 1; s <= a.channel; s++)
    {
        out << "channel: " << s << endl;
        for (int i = 0; i < a.row; i++)
        {
            for (int j = 0; j < a.col; j++)
            {
                // out << std::dec << a.ptr.get()[i * a.col + j] << " ";
                out << a.ptr.get()[i * a.col + j] << " ";
            }
            out << endl;
        }
    }
    return out;
}
```

```
//示例代码
matrix<float> m3(2,2,3,1);
cout << m3 << endl;
//输出结果
//f; row = 2 col = 3
//channel: 1
//1 1 1
//1 1 1
//channel: 2
//1 1 1
//1 1 1
```

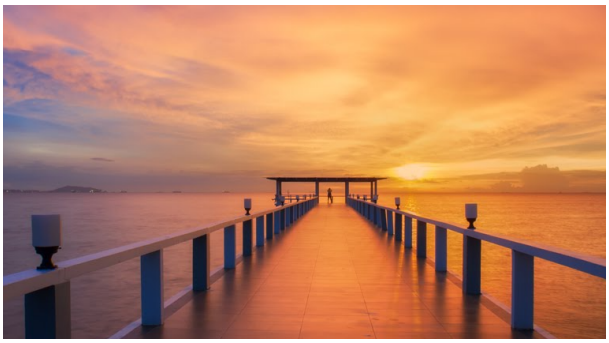
方法正确性检验

在完成代码并debug完毕后，我在main函数里检验上述方法的正确性。

```
#include "matrix.h"
#include <iostream>
#include <memory>

using namespace std;

int main(){
    //矩阵运算正确性检验
    matrix<double> m1(1,2,3,2.0);
    matrix m2(m1);
    matrix<float> m3(2,2,3,1);
    cout << "m1: "<< m1 << endl;
    cout << "m2: "<< m2 << endl;
    cout << "m3: "<< m3 << endl;
    cout << "m1+m2:" << (m1+m2) << endl;
    string str;
    (m1 == m2) ? str = "equal" : str = "not equal";
    cout << "m1 == m2 " << str << endl;
    (m1 == m3) ? str = "equal" : str = "not equal";
    cout << "m1 == m3 " << str << endl;
    (m1 == (m1 + m2)) ? str = "equal" : str = "not equal";
    cout << "m1 == m1 + m2 " << str << endl;
    matrix<int> m4;
    m4 = (m1 + 1)*3 - 1 + m2/m3 + 1 ;
    cout << "m4" << m4 << endl;
    m1 = m2;  //检验soft copy
    m1 = m4;  //检验hard copy
    cout << "operator check!" << endl;

    // 卷积方法正确性实验
    matrix<unsigned char> m5(3,3,2,1);
    matrix<unsigned char> m7(3,2,2,1);
    matrix<unsigned char> result = m5.conv(m7,1,0);
    matrix<int> m6;
    m6 = result;
    cout << "output"<< m6 << endl;
    // 图像处理正确性实验
    matrix<unsigned char> image("test_picture.bmp");
    matrix<unsigned char> kernel(3,1,1,1);
    matrix<unsigned char> output = image.conv(kernel,1,0);
    output.outputImage("output_image.bmp");
    return 0;
}
```

```
//实验结果
./a.out
//m1: d; row = 2 col = 3
//channel: 1
//2 2 2
//2 2 2
//
//m2: d; row = 2 col = 3
//channel: 1
//2 2 2
//2 2 2
//
```

```
//m3: f; row = 2 col = 3
//channel: 1
//1 1 1
//1 1 1
//channel: 2
//1 1 1
//1 1 1
//
//m1+m2:d; row = 2 col = 3
//channel: 1
//4 4 4
//4 4 4
//
//m1 == m2 equal
//m1 == m3 not equal
//m1 == m1 + m2 not equal
//hardcopy
//m4i; row = 2 col = 3
//channel: 1
//9 9 9
//9 9 9
//
//softcopy
//hardcopy
//operator check!
//hardcopy
//outputi; row = 2 col = 1
//channel: 1
//4
//4
//channel: 2
//4
//4
//channel: 3
//4
//4
```



输入图像

输出图像

可以看到，各种方法都可以正常使用。

问题的发现与解决

在完成本次Project的过程中，我遇到过很多的问题，并逐一解决了。

1. 首先遇到的一个问题就是，在刚接触template时，我以为它也可以用.h和.cpp的方法来写。直到十三周大课上课时老师说不能用.cpp文件写时，我才终于明白了之前一直报错的原因。template的方法在编译时需要找到对应的实现方法，但是编译器并不能去.cpp文件中找对应的方法，因此不可以用.h和.cpp的方法来实现。最终我把.cpp中的内容转移到了.h文件内。
2. 在本次项目中，题目要求重点管理好内存，因此，我使用课上教的CPP特有的shared_ptr共享指针来储存数据，站在巨人的肩膀上实现内存的正确管理。然而shared_ptr在使用时与传统指针略有不同，在不同类型指针变换时出现过很多问题，于是我去查阅了很多关于智能指针的相关信息，修复了很多之前没想到的错误。
3. 在kernel、image和output统一格式之后需要更为规范的格式来处理数据，因此，我统一使用channel、row、col的顺序来储存数据。但是图像读取时得到的数据是按照row、col、channel来排列的，因此在project中图像输入和输出添加了对应的转换格式的代

码。在Project3时我使用C语言自定义了读取图片和输出图片的方法，我将这些方法修改得更符合cpp规范后直接移入了本项目。

4. 更加理解const的作用和意义。以前我不是很理解为什么传入参数需要加const，直到我发现下面代码出错，并且上网搜索了原因之后，我才算理解了const的作用。

```
//matrix.h
    template <typename U>
    matrix<T> operator+(matrix<U>&);
//main.cpp
    matrix<int> m1(1,2,3,1);
    matrix<int> m2(1,2,3,2);
    matrix<int> m3(1,2,3,3);
    matrix<int> m = m1 + m2 + m3;
```

在这里，由于+方法传入的是一个非const的参数，因此该参数必须是一个明确储存的参数，比如m1,m2,m3，但是这里m1的+的传入参数是m2+m3之后的结果，这是一个临时变量，在这一行代码运行完就会释放掉，因此是一个const变量，在方法上必须要声明const，否则就会出错。

```
//修改后
template <typename U>
matrix<T> operator+(const matrix<U>&);
```

在修改后，代码就正确了。

5. 更加理解了传入参数加&的作用和意义。在上课时我记得老师说&在cpp中是引用，但我以为传入引用与不传入引用仅仅是速度上有所区别。所以之前我像背公式一样记住返回*this的方法前面需要加&号，可是也不知道为什么。对于以下四个返回值为*this的方法，之前在我使用普通指针存储数据时出现了问题。

```
matrix<T>& operator+=(matrix<U>&); //正确
matrix<T> operator+=(matrix<U>&);  ///直接将数据传入，但是return的值是临时变量，在函数结束后会销毁
matrix<T>& operator+=(matrix<U>);  //将U用另一个变量存储传入方法中，函数输出值直接等于方法中的变量
matrix<T> operator+=(matrix<U>);  //传入参数被复制，传出参数也复制。
```

这样，如果我用普通指针作为class的一个member，那么在第二种和第四种函数方法里，都会出现由于函数结束临时变量被清除，调用delete清除指针，从而得到的matrix的指针指向一块被释放的区域，即随机值，这在内存分配中很危险。而第三种在本情况下会影响方法的速度，占用更多资源，但对结果无太大影响。经过这些错误，我算是彻底明白了这些&的作用。

内存泄漏的检查

在这些事情做完之后，本来project应该算是完成了，但我总觉得还是缺点什么没做。我再次审视Project要求，发现本篇重点在内存管理，而我的内存管理我全都交给伟大的shared_ptr来解决。这时我突然意识到：**我该怎么确定我的内存真的被shared_ptr自动释放了呢？**想到这，我赶紧上网搜索检查内存泄漏的方法。

使用Valgrind memcheck进行内存检查

在网上搜索一段时间后，我发现了一个开源软件常用于内存泄漏的检查，Valgrind。

在下载安装完成后，我用其中memcheck工具进行了内存检查，以下是输出结果

```
root@DESKTOP-7LSGQN0:/mnt/g/cpp_program/Project4_test# valgrind --tool=memcheck --leak-check=full ./main
==18401== Memcheck, a memory error detector
==18401== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==18401== Using Valgrind-3.21.0 and LibVEX; rerun with -h for copyright info
==18401== Command: ./main
==18401==
==18401== Mismatched free() / delete / delete []
==18401==    at 0x484BAB6: operator delete(void*, unsigned long) (vg_replace_malloc.c:1072)
==18401==    by 0x113B87: std::_Sp_counted_ptr<unsigned char*, (__gnu_cxx::_Lock_policy)2>::_M_dispose() (shared_ptr_base.h:348)
==18401==    by 0x1126A4: std::_Sp_counted_base<(__gnu_cxx::_Lock_policy)2>::_M_release() (shared_ptr_base.h:168)
==18401==    by 0x112351: std::__shared_count<(__gnu_cxx::_Lock_policy)2>::operator=(std::__shared_count<(__gnu_cxx::_Lock_policy)2> const&
==18401==    by 0x111F40: std::__shared_ptr<unsigned char, (__gnu_cxx::_Lock_policy)2>::operator=(std::__shared_ptr<unsigned char, (__gnu_c
==18401==    by 0x111F6E: std::shared_ptr<unsigned char>::operator=(std::shared_ptr<unsigned char> const&) (shared_ptr.h:359)
==18401==    by 0x10F983: matrix<unsigned char>::conv(matrix<unsigned char>&, int, int) (matrix.h:538)
==18401==    by 0x10A50B: main (main.cpp:43)
```

```
==18401==  Address 0x5005cc0 is 0 bytes inside a block of size 3 alloc'd
==18401==    at 0x484A2D3: operator new[](unsigned long) (vg_replace_malloc.c:714)
==18401==    by 0x10F68B: matrix<unsigned char>::matrix(int, int, int, unsigned char) (matrix.h:161)
==18401==    by 0x10F853: matrix<unsigned char>::conv(matrix<unsigned char>&, int, int) (matrix.h:523)
==18401==    by 0x10A50B: main (main.cpp:43)
==18401==
==18401== Mismatched free() / delete / delete []
==18401==    at 0x484BAB6: operator delete(void*, unsigned long) (vg_replace_malloc.c:1072)
==18401==    by 0x113CCB: std::_Sp_counted_ptr<int*, (__gnu_cxx::_Lock_policy)2>::_M_dispose() (shared_ptr_base.h:348)
==18401==    by 0x1126A4: std::_Sp_counted_base<(__gnu_cxx::_Lock_policy)2>::_M_release() (shared_ptr_base.h:168)
==18401==    by 0x11207A: std::__shared_count<(__gnu_cxx::_Lock_policy)2>::~__shared_count() (shared_ptr_base.h:705)
==18401==    by 0x111C95: std::__shared_ptr<int, (__gnu_cxx::_Lock_policy)2>::~__shared_ptr() (shared_ptr_base.h:1154)
==18401==    by 0x111CB5: std::shared_ptr<int>::~shared_ptr() (shared_ptr.h:122)
==18401==    by 0x111CD9: matrix<int>::~matrix() (matrix.h:11)
==18401==    by 0x10A53B: main (main.cpp:50)
==18401==  Address 0x5056660 is 0 bytes inside a block of size 1,319,220 alloc'd
==18401==    at 0x484A2D3: operator new[](unsigned long) (vg_replace_malloc.c:714)
==18401==    by 0x11211C: matrix<int>& matrix<int>::operator=<unsigned char>(matrix<unsigned char> const&) (matrix.h:198)
==18401==    by 0x10A52A: main (main.cpp:45)
==18401==
==18401== Mismatched free() / delete / delete []
==18401==    at 0x484BAB6: operator delete(void*, unsigned long) (vg_replace_malloc.c:1072)
==18401==    by 0x113B87: std::_Sp_counted_ptr<unsigned char*, (__gnu_cxx::_Lock_policy)2>::_M_dispose() (shared_ptr_base.h:348)
==18401==    by 0x1126A4: std::_Sp_counted_base<(__gnu_cxx::_Lock_policy)2>::_M_release() (shared_ptr_base.h:168)
==18401==    by 0x11207A: std::__shared_count<(__gnu_cxx::_Lock_policy)2>::~__shared_count() (shared_ptr_base.h:705)
==18401==    by 0x111C2D: std::__shared_ptr<unsigned char, (__gnu_cxx::_Lock_policy)2>::~__shared_ptr() (shared_ptr_base.h:1154)
==18401==    by 0x111C4D: std::shared_ptr<unsigned char>::~shared_ptr() (shared_ptr.h:122)
==18401==    by 0x111C71: matrix<unsigned char>::~matrix() (matrix.h:11)
==18401==    by 0x10A547: main (main.cpp:50)
==18401==  Address 0x5005d70 is 0 bytes inside a block of size 329,805 alloc'd
==18401==    at 0x484A2D3: operator new[](unsigned long) (vg_replace_malloc.c:714)
==18401==    by 0x10F95A: matrix<unsigned char>::conv(matrix<unsigned char>&, int, int) (matrix.h:537)
==18401==    by 0x10A50B: main (main.cpp:43)
==18401==
==18401==
==18401== HEAP SUMMARY:
==18401==     in use at exit: 0 bytes in 0 blocks
==18401==   total heap usage: 17 allocs, 17 frees, 4,035,397 bytes allocated
==18401==
==18401== All heap blocks were freed -- no leaks are possible
==18401==
==18401== For lists of detected and suppressed errors, rerun with: -s
==18401== ERROR SUMMARY: 6 errors from 3 contexts (suppressed: 0 from 0)
```

valgrind指出我的代码中有Mismatched free() / delete / delete []的问题，在网上搜索这一问题原因后（C/C++的内存泄漏检测工具Valgrind memcheck的使用经历 - 知乎 (zhihu.com)），我发现这是由于混乱调用malloc/new/delete/free的结果，或者是错用了delete和delete[]。在网上又搜索了许多关于shared_ptr指针的相关问题后，我才发现在声明时我需要声明指针数组，才能调用delete[]方法，否则指针会默认调用delete方法，从而导致内存泄漏。

```
shared_ptr<T> pointer(new T[row*col]);//错误，调用delete.
shared_ptr<T[]> pointer(new T[row*col]);//正确,调用delete[]
```

于是我将代码中的所有声明shared_ptr的位置全部修改，这次再检查内存泄漏：

```
valgrind --tool=memcheck --leak-check=full ./main
==770== Memcheck, a memory error detector
==770== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==770== Using Valgrind-3.21.0 and LibVEX; rerun with -h for copyright info
==770== Command: ./main
...
==770==
==770== HEAP SUMMARY:
==770==     in use at exit: 0 bytes in 0 blocks
==770==   total heap usage: 55 allocs, 55 frees, 1,624,435 bytes allocated
==770==
==770== All heap blocks were freed -- no leaks are possible
==770==
==770== For lists of detected and suppressed errors, rerun with: -s
==770== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

这样就确定没有内存泄漏了！

因此，完整版的代码是把所有的shared_ptr<T>改为shared_ptr<T[]>，详见matrix.h文件。

```
class matrix{
    public:
    int row;
    int col;
    int channel;
    shared_ptr<T[]> ptr;
...
```

在ARM平台的测试

在ARM平台运行同样的代码时出现了错误。搜索原因发现不同平台对g++的支持和默认程度不一致，所以会导致出现问题。

```
g++ main.cpp
main.cpp: In function ‘int main()’:
main.cpp:11:12: error: missing template arguments before ‘m2’
     matrix m2(m1);
            ^~
main.cpp:15:22: error: ‘m2’ was not declared in this scope
     cout << "m2: "<< m2 << endl;
                     ^~
main.cpp:15:22: note: suggested alternative: ‘m3’
     cout << "m2: "<< m2 << endl;
                     ^~
                     m3
main.cpp:35:43: error: cannot bind non-const lvalue reference of type ‘matrix<un
     matrix<unsigned char> result = m5.conv(m7,1,0);
                                    ~~~~~~~^~~~~~~~
In file included from main.cpp:1:
matrix.h:171:1: note:   initializing argument 1 of ‘matrix<T>::matrix(matrix<T>&
 matrix<T>::matrix(matrix<T> &m)
 ^~~~~~~~
main.cpp:42:46: error: cannot bind non-const lvalue reference of type ‘matrix<un
     matrix<unsigned char> output = image.conv(kernel,1,0);
                                    ~~~~~~~~~~^~~~~~~~~~~~
In file included from main.cpp:1:
matrix.h:171:1: note:   initializing argument 1 of ‘matrix<T>::matrix(matrix<T>&
 matrix<T>::matrix(matrix<T> &m)
 ^~~~~~~~
```

1. 第 11行的代码缺少对tempalte具体类型的声明，在网上查找原因后得知可能是ARM平台没有自动推断template类型的功能，因此需要显示的写明matrix的类型<double>。

```
matrix m2(m1);
```

2. 编译器说35行和42行在矩阵初始化时左值和右值不匹配，检查代码后发现确实存在这个问题。

```
//出错代码
matrix<T>(matrix<T>&); //未加const

matrix<unsigned char> result = m5.conv(m7,1,0);
//m5.conv(m7,1,0)是右值，而初始化方法中未加const，存在将右值转化成左值的问题。
```

在初始化方法中加入const之后，问题解决。但为什么在x86平台上面的问题没有报错呢？可能这就是不同平台g++的容忍范围的区别吧。

修复以上问题后，代码正常运行。

```
[openailab@localhost project4]$ g++ main.cpp
[openailab@localhost project4]$ ./a.out
m1: d; row = 2 col = 3
channel: 1
2 2 2
2 2 2
```

```
m2: d; row = 2 col = 3
channel: 1
2 2 2
2 2 2

m3: f; row = 2 col = 3
channel: 1
1 1 1
1 1 1
channel: 2
1 1 1
1 1 1

m1+m2:d; row = 2 col = 3
channel: 1
4 4 4
4 4 4

m1 == m2 equal
m1 == m3 not equal
m1 == m1 + m2 not equal
hardcopy
m4i; row = 2 col = 3
channel: 1
9 9 9
9 9 9

softcopy
hardcopy
operator check!
hardcopy
outputi; row = 2 col = 1
channel: 1
4
4
channel: 2
4
4
channel: 3
4
4
```

总结

Highlights

- 完成Project基本要求。
- 可以支持不同类型矩阵的操作。
- 自定义的图片输入输出方法，支持CNN的卷积操作。
- 进行了内存泄漏检查。
- 进行了x86和ARM不同平台的测试。

在本次Project中，我顺利完成了题目中的要求，并且专门检查了内存泄漏，也在新的平台尝试了同样的代码并发现了问题。经过这么多次project，我发现总是project开始的时候看起来都很简单，但是在实践尝试的时候就会出现各种问题，而在解决这些预料之外的问题的同时就是自己的提升。比如我在这次project中更加理解了模板template的使用方法、巩固了const和&的基本知识；又比如在project3中我曾嫌弃自己把代码跨平台的方法太low了（发到网盘，再下载），当时就想尝试用SSH发送文件但是失败了。这次我就成功使用SSH进行了跨平台的连接，大大增大了测试效率（误）。这件事可能本身没有太大意义，但对我自己来说还是挺有成就感的，也算是实现了自己的想法。

在完成了project4基本代码后，我一直在思考如何改进让我的代码更加正确、更加方便，这可能就是我在前三次project中学到的东西。总之，又一个project落下帷幕了，希望最后一次project自己也能继续努力。