

# CS205 Project4

## 1. 思路分析

本次project要求设计一个卷积神经网络中的类，要求包含行数、列数等基本数据，并且需要满足一些其余的要求。

1. 要求类可以支持不同的元素类型：对于此要求，我选择使用**template class**完成，以处理绝大多数情况。
2. 不能使用hard copy，并且需要尽可能避免内存泄漏：对于此要求，我使用了智能指针中的**shared\_ptr**以避免复杂的内存管理，并且传入占据内存的参数均为const。
3. 要求重载运算符：对于此要求，我重载了"**=, ==, (), +, -, \***"的运算符，其中"**=""=="** (**()**) "的重载均在类里面完成，其余运算符的重载则设为友元函数，在类外完成重载。
4. 在X86和ARM平台测试程序并描述差别：对于此要求，我分别在mac和windows系统下运行了程序。

## 2. 代码实现

### 2.1 基础函数

#### 2.1.1 get 和 set

首先我创建了一个类，并且定义了如下变量：

```
private:
    size_t rows;
    size_t cols;
    size_t channels;
    size_t length;
    shared_ptr<T> data;
```

上面三个变量分别为行、列和通道数，length为长度，在data分配内存时较为方便。data是用智能指针定义的一维数组，信息均储存在内。此外，为确保变量不会被轻易修改，故上述变量均设为private，调用时则采用public的get方法。

代码如下：

```
// get函数
    size_t getRows(){
        return rows;
    }
    size_t getCols(){
        return cols;
    }
    size_t getChannels(){
        return channels;
    }
```

```

size_t getLength(){
    return length;
}
shared_ptr<T> getData(){
    if(data == NULL){
        return NULL;
    }else{
        return data;
    }
}

```

除了基础的get函数以外，我还定义了获取具体某一个元素的函数。调取元素有两种方法，一种是将其视作一维数组，直接调用；第二种是按常理认知的行和列及通道数进行调用。需要注意的是需要对传入的参数进行检查，若超出数组范围则返回错误提示，并返回默认值。

```

// 获取矩阵中的元素
T getElement(size_t i) {
    // 进行边界检查
    if(i >= length || data == NULL){
        cerr << "无法调取元素！" << endl;
        return T();
    }else{
        return data.get()[i];
    }
}

T getElement(size_t row, size_t col, size_t channel) {
    if(row > rows || col > cols || channel > channels){
        cerr << "已越界！" << endl;
        return T();
    }else{
        size_t offset = (((row-1) * cols + (col-1)) * channels + (channel-1));
        return data.get()[offset];
    }
}

```

set函数则是以布尔值为返回类型，对参数进行判断之后，更改对应位置的元素。

```

//set函数
bool setElement(size_t i, T value){
    if(i >= length){
        cerr << "传入参数越界！" << endl;
        return false;
    }
    data.get()[i] = value;
    return true;
}

bool setElement(size_t row, size_t col, size_t channel, T value){
    if(row == 0 || col == 0 || channel == 0){
        cerr << "传入参数越界！" << endl;
        return false;
    }
}

```

```

    }
    size_t offset = (((row-1) * cols + (col-1)) * channels + (channel-1));
    data.get()[offset] = value;
    return true;
}

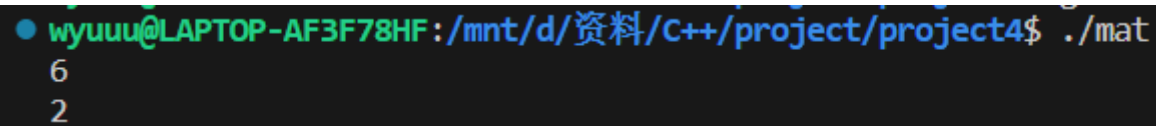
```

用如下代码进行测试：

```

// 测试get和set
cout << A.getElement(2,1,3) << endl;
A.setElement(2,1,3,2);
cout << A.getElement(2,1,3) << endl;

```



```

wyuuu@LAPTOP-AF3F78HF:/mnt/d/资料/C++/project/project4$ ./mat
6
2

```

得到结果符合预期。

## 2.1.2 构造函数

构造函数分为三种，第一种是传入行数、列数和通道数，数据默认为0；第二种是可以将所有数据变为同一个常量，如1，2.1f等；第三种是直接传入智能指针与数据长度。

由于有智能指针的存在，析构造函数的作用仅为特殊情况下手动释放资源。

```

// 构造函数和析构造函数
Mat(size_t rows, size_t cols, size_t channels) : rows(rows), cols(cols),
channels(channels){
    length = rows * cols * channels;
    T* ptr = new T[length];
    data.reset(ptr);
}
Mat(size_t rows, size_t cols, size_t channels, T a) : rows(rows), cols(cols),
channels(channels)
{
    length = rows * cols * channels;
    T* ptr = new T[length];
    data.reset(ptr);
    for (int i = 0; i < length; i++)
    {
        data.get()[i] = a;
    }
}
Mat(size_t row, size_t col, size_t channel, const shared_ptr<T>& array, size_t
n) : rows(row), cols(col), channels(channel) {
    if (n != row * col * channel) {
        cerr << "传入数组长度不符!" << endl;
    } else {
        length = row * col * channel;
        data = array;
    }
}

```

```

    }
}

~Mat(){
    data.reset();
}

```

测试代码如下：

```

// 测试构造函数
float* array3 = new float[6]{1.0f,1.1f,1.2f,1.3f,1.4f,1.5f};
shared_ptr<float> sp3(array3);
Mat <float> C(2,1,3,sp3,6);
Mat <int> D(2,1,3);
Mat <int> E(2,1,3,1);
cout << "方法一: ";
for(int i = 0; i < 6; i++){
    cout << D.getElement(i) << " ";
}
cout << endl;
cout << "方法二: ";
for(int i = 0; i < 6; i++){
    cout << E.getElement(i) << " ";
}
cout << endl;
cout << "方法三: ";
for(int i = 0; i < 6; i++){
    cout << C.getElement(i) << " ";
}
cout << endl;

```

```

● wyuuu@LAPTOP-AF3F78HF:/mnt/d/资料/C++/project/project4$ ./mat
方法一: 0 0 0 0 0 0
方法二: 1 1 1 1 1 1
方法三: 1 1.1 1.2 1.3 1.4 1.5

```

结果符合预期。

## 2.2 运算符重载

### 2.2.1 () 和 = 的重载

() 和 = 的功能类似，都是复制，且均在类中实现，由于操作由智能指针完成，所以都是浅拷贝。

```

// 操作符的重载
Mat &operator()(const Mat& other)
{
    if (other.data == NULL)
    {

```

```

        cerr << "传入空指针!" << endl;
        return *this;
    }
    else
    {
        rows = other.rows;
        cols = other.cols;
        channels = other.channels;
        data = other.data;
        return *this;
    }
}
Mat &operator=(const Mat& other)
{
    if (other.data == NULL)
    {
        cerr << "传入空指针!" << endl;
        return *this;
    }
    else
    {
        rows = other.rows;
        cols = other.cols;
        channels = other.channels;
        data = other.data;
        return *this;
    }
}
}

```

测试代码如下：

```

// 测试 () 和 =
Mat <float> C1 = C;
Mat <float> C2(C);
cout << "C: ";
for(int i = 0; i < 6; i++){
    cout << C.getElement(i) << " ";
}
cout << endl;
cout << "C1: ";
for(int i = 0; i < 6; i++){
    cout << C1.getElement(i) << " ";
}
cout << endl;
cout << "C2: ";
for(int i = 0; i < 6; i++){
    cout << C2.getElement(i) << " ";
}
cout << endl;

```

结果如下：

```
C: 1 1.1 1.2 1.3 1.4 1.5
C1: 1 1.1 1.2 1.3 1.4 1.5
C2: 1 1.1 1.2 1.3 1.4 1.5
```

符合预期。

## 2.2.2 == 的重载

==的返回结果为布尔类型，判断两个Mat类是否相等以返回，相等返回1，否则返回0。具体实现为先判断行数、列数、通道数是否相等，若不等则直接返回false，若相等则继续比较每一个值是否相等，若不等则返回false，若最后每一个值均相等则返回true。

```
bool operator==(const Mat& other)
{
    // 对输入矩阵进行检查
    if (rows != other.rows || cols != other.cols || channels != other.channels)
    {
        return false;
    }
    else
    {
        T *raw_ptr1 = data.get(); // 获取指向数组的原始指针
        const T *raw_ptr2 = other.data.get();
        for (int i = 0; i < length; i++)
        {
            if (raw_ptr1[i] != raw_ptr2[i])
            {
                return false;
            }
        }
        return true;
    }
}
```

测试代码如下：

```
int a = C1 == C2;
cout << "C1,C2是否相等：" << a << endl;
```

```
C1,C2是否相等: 1
```

结果符合预期。

## 2.2.3 + 和 - 的重载

加法和减法重载类似，都是以友元函数的形式实现，在类的内部进行声明，外部具体实现。具体实现考虑了两种情况，一种是两者都是Mat类时，对应元素相加；另一种是常数a与Mat类相加，实现方法为构造与传入Mat尺寸相同的矩阵，并令其所有元素为a，再进行加减法。

具体代码如下:

```
// 加法
template <typename T2>
Mat<T2> operator+(const Mat<T2> &lhs, const Mat<T2> &rhs)
{
    Mat<T2> result(lhs.rows, lhs.cols, lhs.channels);
    // 对输入矩阵进行检查
    if (lhs.rows != rhs.rows || lhs.cols != rhs.cols || lhs.channels !=
rhs.channels)
    {
        cerr << "矩阵大小不一致! " << endl;
        return result;
    }
    else
    {
        T2 *ptr = result.data.get();
        const T2 *raw_ptr1 = lhs.data.get(); // 获取指向数组的原始指针
        const T2 *raw_ptr2 = rhs.data.get();
        for (int i = 0; i < lhs.length; i++)
        {
            ptr[i] = raw_ptr1[i] + raw_ptr2[i]; // 修改元素值
        }
        return result;
    }
}

template <typename T2>
Mat<T2> operator+(T2 a, const Mat<T2> &lhs)
{
    Mat<T2> result(lhs.rows, lhs.cols, lhs.channels);
    Mat<T2> rhs(lhs.rows, lhs.cols, lhs.channels, a);
    // 对输入矩阵进行检查
    T2 *ptr = result.data.get();
    const T2 *raw_ptr1 = lhs.data.get(); // 获取指向数组的原始指针
    const T2 *raw_ptr2 = rhs.data.get();
    for (int i = 0; i < lhs.length; i++)
    {
        ptr[i] = raw_ptr1[i] + raw_ptr2[i]; // 修改元素值
    }
    return result;
}

template <typename T2>

// 减法
Mat<T2> operator-(const Mat<T2> &lhs, const Mat<T2> &rhs)
{
    Mat<T2> result(lhs.rows, lhs.cols, lhs.channels);
    // 对输入矩阵进行检查
    if (lhs.rows != rhs.rows || lhs.cols != rhs.cols || lhs.channels !=
rhs.channels)
    {
        cerr << "矩阵大小不一致! " << endl;
```

```

        return result;
    }
    else
    {
        T2 *ptr = result.data.get();
        const T2 *raw_ptr1 = lhs.data.get(); // 获取指向数组的原始指针
        const T2 *raw_ptr2 = rhs.data.get();
        for (int i = 0; i < lhs.length; i++)
        {
            ptr[i] = raw_ptr1[i] - raw_ptr2[i]; // 修改元素值
        }
        return result;
    }
}

template <typename T2>
Mat<T2> operator-(T2 a, const Mat<T2> &lhs)
{
    Mat<T2> result(lhs.rows, lhs.cols, lhs.channels);
    Mat<T2> rhs(lhs.rows, lhs.cols, lhs.channels, a);
    // 对输入矩阵进行检查
    T2 *ptr = result.data.get();
    const T2 *raw_ptr1 = lhs.data.get(); // 获取指向数组的原始指针
    const T2 *raw_ptr2 = rhs.data.get();
    for (int i = 0; i < lhs.length; i++)
    {
        ptr[i] = raw_ptr1[i] - raw_ptr2[i]; // 修改元素值
    }
    return result;
}

```

测试代码如下：

```

// 测试 + 和 -
C = C1 + C2;
cout << "C1 + C2 = : ";
for(int i = 0; i < 6; i++){
    cout << C.getElement(i) << " ";
}
cout << endl;
C = C1 - C2;
cout << "C1 - C2 = : ";
for(int i = 0; i < 6; i++){
    cout << C.getElement(i) << " ";
}
cout << endl;

```

```

C1 + C2 = : 2 2.2 2.4 2.6 2.8 3
C1 - C2 = : 0 0 0 0 0 0

```

结果符合预期。



## 2.2.4 \*的重载

这里实现的是点乘的重载，即矩阵对应位置元素相乘，同样有常数与矩阵相乘和矩阵与矩阵相乘两种形式，具体代码如下：

```
template <typename T2>
Mat<T2> operator*(const Mat<T2> &lhs, const Mat<T2> &rhs)
{
    Mat<T2> result(lhs.rows, lhs.cols, lhs.channels);
    // 对输入矩阵进行检查
    if (lhs.rows != rhs.rows || lhs.cols != rhs.cols || lhs.channels !=
rhs.channels)
    {
        cerr << "矩阵大小不一致！" << endl;
        return result;
    }
    else
    {
        T2 *ptr = result.data.get();
        const T2 *raw_ptr1 = lhs.data.get(); // 获取指向数组的原始指针
        const T2 *raw_ptr2 = rhs.data.get();
        for (int i = 0; i < lhs.length; i++)
        {
            ptr[i] = raw_ptr1[i] * raw_ptr2[i]; // 修改元素值
        }
        return result;
    }
}

template <typename T2>
Mat<T2> operator*(T2 a, const Mat<T2> &lhs)
{
    Mat<T2> result(lhs.rows, lhs.cols, lhs.channels);
    Mat<T2> rhs(lhs.rows, lhs.cols, lhs.channels, a);
    // 对输入矩阵进行检查
    T2 *ptr = result.data.get();
    const T2 *raw_ptr1 = lhs.data.get(); // 获取指向数组的原始指针
    const T2 *raw_ptr2 = rhs.data.get();
    for (int i = 0; i < lhs.length; i++)
    {
        ptr[i] = raw_ptr1[i] * raw_ptr2[i]; // 修改元素值
    }
    return result;
}
```

测试代码如下：

```
// 测试 *
C = C1 * C2;
cout << "C1 * C2 = : ";
for(int i = 0; i < 6; i++){
    cout << C.getElement(i) << " ";
}
cout << endl;
C = 2.0f * C1;
cout << "2 * C1 = : ";
for(int i = 0; i < 6; i++){
    cout << C.getElement(i) << " ";
}
cout << endl;
```

测试结果如下：

```
C1 * C2 = : 1 1.21 1.44 1.69 1.96 2.25
2 * C1 = : 2 2.2 2.4 2.6 2.8 3
```

结果符合预期。

## 3. X86与ARM平台的对比

### 3.1 结果上对比

```
before set: 6
after set: 2
方法一： 0 0 0 0 0 0
方法二： 1 1 1 1 1 1
方法三： 1 1.1 1.2 1.3 1.4 1.5
C: 1 1.1 1.2 1.3 1.4 1.5
C1: 1 1.1 1.2 1.3 1.4 1.5
C2: 1 1.1 1.2 1.3 1.4 1.5
C1,C2是否相等： 1
C1 + C2 = : 2 2.2 2.4 2.6 2.8 3
C1 - C2 = : 0 0 0 0 0 0
C1 * C2 = : 1 1.21 1.44 1.69 1.96 2.25
2 * C1 = : 2 2.2 2.4 2.6 2.8 3
```

```

● wyuuu@LAPTOP-AF3F78HF:/mnt/d/资料/C++/project/project4$ ./mat
before set: 6
after set: 2
方法一: 0 0 0 0 0 0
方法二: 1 1 1 1 1 1
方法三: 1 1.1 1.2 1.3 1.4 1.5
C: 1 1.1 1.2 1.3 1.4 1.5
C1: 1 1.1 1.2 1.3 1.4 1.5
C2: 1 1.1 1.2 1.3 1.4 1.5
C1,C2是否相等: 1
C1 + C2 = : 2 2.2 2.4 2.6 2.8 3
C1 - C2 = : 0 0 0 0 0 0
C1 * C2 = : 1 1.21 1.44 1.69 1.96 2.25
2 * C1 = : 2 2.2 2.4 2.6 2.8 3

```

可以看到，在结果上并没有区别。

## 3.2 性能上对比

令加法、减法、乘法各做1000次，ARM系统跑出来的结果为56ms。

```
Execution time: 56 milliseconds
```

X86跑出来的结果为63ms。

```
Execution time: 63 milliseconds
```

同样没有太大区别。

## 4.总结

在正式开始这次项目之前，我以为这次project的工作量不会太大，但当我正式开始做的时候才发现这次project的繁杂程度。首先在智能指针上我就遇到了很大的阻碍，经常莫名其妙的出bug，并且还不能直接用智能指针初始化数组（据说C++20标准可以），不过经过不断的满屏报错，对内存管理也有了更深的理解。

其次就是操作符的重载，有些操作符适合在类内重载，而有些操作符适合在类外重载（如常数与矩阵的计算等）。在这次project中，由于时间所限，仍然有许多操作没有完成，例如 << 的重载，叉乘的实现等。

其次就是对传入参数的检查，一个成熟的程序需要能够应对各种情况，但有时候bug还是会从不知名的地方冒出来，参数检查可以起到很好的排查作用，在我程序的调试过程中，就遇到了很多的bug，好在有各个报错提示，让我得以修改bug。