CS205C/C++ Programming-Project5
A Simple CNN Model

Name: 王宇航　SID：12012208

Content

Part1 - Analysis

首先我们需要明白这次Project的目的，手动实现一个简单的CNN模型。由于训练的数据已经给出，所以最困难的一步，学习如何识别人脸其实可以跳过了。根本目的是让我们对CNN的有一个较为全面的了解，清楚每一层进行了什么操作。同时，也要将前几次Project积累的经验用到这上面来：包括但不限于小心内存管理，认真检查函数参数，如何优雅的完成任务等。需要像对待一个工程一样对待这次Project，而不是仅仅完成任务，计算出正确答案。

Part2 - Code(部分代码展示)

Mat.hpp

1)Mat类

```
class Mat_conv
{
public:
    size_t row;
    size_t coloumn;
    size_t channel;
    int *counter;
    float *Matrix;

public:
    Mat_conv();
    Mat_conv(int r, int c, int channel);
    Mat_conv(const Mat_conv &Mat);
    ~Mat_conv();
    bool conv(const Mat_conv &mat1, const conv_param &mat2, Mat_conv &Mat);
    bool dot(const Mat_conv &mat1, const conv_param &mat2, Mat_conv &matMul, int i);
    bool setZero(int i, int j, size_t in_size, float (&t)[9], const Mat_conv &mat1,
                 int flag1, size_t flag2, size_t flag3, size_t Ksize1);
    bool maxPool(const Mat_conv &mat, const Mat_conv &maxPool);
    bool ReLu(const Mat_conv &mat);
    bool setBias1(const Mat_conv &mat, float *(&bias));
    bool fullConnect(float *(&result), const Mat_conv &mat, fc_param &fullC);
    bool softMax(float *(&result), int size);
    bool CNN(const Mat_conv &mat, conv_param *(conv_params), fc_param &fc_params);
};
```

如上，这是Mat类的定义，本次Project，我将row，coloumn和channel全部改为了size_t类型，同时，沿用上次Project的做法，将矩阵数据存在一维指针中，用一个指针记录有几个对象共享这份矩阵数据，采用写时拷贝的方式进行内存管理。再者，听了老师的代码规范后，本程序严格检查传入的参数。这个类我将成员函数的返回值全部置为bool，这样一但检测出错误马上返回，避免程序崩溃。

2)conv_param类

这个类沿用了Github上示例文档的类，没有做改变。

3)fc_param类

同理，沿用了Github上示例文档的类。

weight.hpp(训练数据头文件)

这个头文件定义了训练得出的参数，在需要使用的cpp文件中include即可。

func.cpp

1)宏define

```
#define coutError                                   \
    cout << "----------------------" << endl;    \
    cerr << "error! nullptr argument" << endl; \
    cerr << "file = " << __FILE__ << "   fun =  " << __func__ << "  line =  " << __LINE__ - 4 << endl;

#define flagCheck                       \
    if (!flag)                           \
    {                                   \
        cout << "----------------------" << endl;  \
        cerr << "error! return value is false" << endl;  \
        cerr << "file = " << __FILE__ << "   fun =  " << __func__ << "  line =  " << __LINE__ - 4 << endl; \
        return false;                   \
    }
```

参考了于老师的检查参数信息的方法后，我对于每次传入的参数都进行了严格检查，coutError这个宏适用于传入的参数中存在空指针的情况，发现此类错误在后面加一个coutError即可，十分优雅，并且可以打印出现错误的文件名，函数名以及出错的行号。还有一点，由于每层函数的调用返回值均为bool，一旦出错，可以从出错位置不断回溯，将该方法栈中所有函数调用的位置打印出来，错误信息一目了然，方便调试。而flagCheck这个宏适用于每次操作后检查是否出错，如果出错则返回false，避免程序崩溃。（补充，在上了于老师C/C++异常处理课时后，我了解到了有更简便的写法，如assert断言言，但考虑到修改工程量较大，并未修改）

2)构造函数（无参，有参，拷贝构造）

有参与无参构造较为简单，且传入的row，col，channel为int类型，我对其做了是否为正数的检查，若出现小于0的数则不予构造。这里仅对拷贝构造函数进行说明。

```
Mat_conv::Mat_conv(const Mat_conv &Mat)
{
    if (Mat.Matrix == NULL)
    {
```

```
        coutError;
    }
    else{
    this->row = Mat.row;
    this->coloumn = Mat.coloumn;
    this->channel = Mat.channel;
    this->counter = Mat.counter;
    (*this->counter)++;
    this->Matrix = Mat.Matrix;
    }
};
```

可见，在进行检查后，构造过程中，将Mat的所有成员变量直接赋值给this，对于指针类型的Matrix，两者共享一份内存，同时将counter++，既避免了硬拷贝浪费空间，又可以解决软拷贝的内存释放问题

3)析构函数

```
Mat_conv::~Mat_conv()
{

    if ((*this->counter) <= 1)
    { //若当前仅一个Mat类享有这份数据，则直接delete
        cout << "class is release" << endl;
        delete this->counter;
        //cout<<"free------"<<hex<<(void*)this->Matrix<<endl;
        this->counter == NULL;
        if (this->Matrix != NULL)
        {
            delete[](this->Matrix);
            this->Matrix == NULL;
        }
    }
    else
    { //若当前有多个Mat类共享这份数据，则将共享的类的个数减一，表示其中一个类被删除了
        (*this->counter)--;
    }
}
```

在析构函数中，每次调用时，若调用类的counter为1，则直接释放内存，若不为1，则将counter减一，避免double free。释放时判断Matrix是否为空指针是因为在无参构造时，仅仅初始化了counter指针为1。

4)卷积层操作

4.1)conv函数

```
bool Mat_conv::conv(const Mat_conv &mat1, const conv_param &mat2, Mat_conv &matMul)
```

我这次的卷积操作有两种方法，一种是采用滑动窗口形式的，按照卷积的定义一步一步运算，函数conv和其辅助函数dot便是该方法的实现。另一种是将整个层的矩阵卷积转化为矩阵乘法运算，该方法通过sgemm函数实现。该函数为第一种方法，较为简单,仅贴出部分代码。

```
for (int i = 0; i < mat2.out_channels; i++)
    {
        bool flag = dot(mat1, mat2, matMul, i);
        flagCheck;
    }
```

该函数在对传入参数进行检查后（判断各指针是否为空，是否符合矩阵相乘规则），倘若无误，则执行上面语句，按滑动窗口的思路计算，简单而暴力。mat1是左边矩阵，mat2实际是权重矩阵，为右矩阵，matMul保存结果。

4.2)dot函数（conv辅助函数）

```
bool Mat_conv::dot(const Mat_conv &mat1, const conv_param &mat2, Mat_conv &matMul,
int index_channel)
{
    //参数检查代码重复度高，未贴出
    size_t in_size = mat1.row;              //输入矩阵的大小
    size_t in_channel = mat1.channel;       //输入矩阵的channel
    size_t out_size = (in_size - mat2.kernel_size + 2 * mat2.pad) / mat2.stride + 1;
  //输出矩阵的size大小，由卷积公式得出
    size_t out_channel = mat2.out_channels;//输出矩阵的channel
    size_t size1 = mat1.row * mat1.coloumn;//矩阵换channel时指针偏移量
    size_t size2 = mat2.kernel_size * mat2.kernel_size * in_channel;    //权重矩阵换
channel时指针偏移量
    size_t size3 = index_channel * out_size * out_size;
    __m256 vector1;
    __m256 vector2;
    __m256 result;
    size_t s;//i和j的起始位置
    if (mat2.pad)
    {
        s = 0;
    }
    else
    {
        s = 1;
    }
    for (int k = 0; k < in_channel; k++)
    {
        vector2 = _mm256_loadu_ps(&mat2.p_weight[k * 9 + index_channel * size2]);
        int Mulcounter = 0;
        for (int i = s; i < in_size - s; i += mat2.stride)
        {
            int flag1 = (i - 1) * in_size;//3*3卷积核第一行对应起始位置
            int flag2 = (i)*in_size;//3*3卷积核第二行对应起始位置
            int flag3 = (i + 1) * in_size;//3*3卷积核第三行对应起始位置
            for (int j = s; j < in_size - s; j += mat2.stride)
            {
                float t[9]{0, 0, 0, 0, 0, 0, 0, 0, 0};
         if (mat2.pad && (i == 0 || i == in_size - 1 || j == 0 || j == in_size - 1))
                {//逻辑上补零
          bool flag = setZero(i, j, in_size, t, mat1, flag1, flag2, flag3, k * size1);

                    flagCheck;
```

```
                    vector1 = _mm256_loadu_ps(&t[0]);
                }
            else
                {//直接装载mat1被扫过的数据
                    int i1 = flag1 + j + k * size1;
                    int i2 = flag2 + j + k * size1;
                    int i3 = flag3 + j + k * size1;
                vector1 = _mm256_set_ps(mat1.Matrix[i3], mat1.Matrix[i3 - 1], mat1.Matrix[i2 + 1],
mat1.Matrix[i2], mat1.Matrix[i2 - 1], mat1.Matrix[i1 + 1], mat1.Matrix[i1], mat1.Matrix[i1 - 1]);
                    t[8] = mat1.Matrix[i3 + 1];
                }
                result = _mm256_mul_ps(vector1, vector2);
                float sum = result[0] + result[1] + result[2] + result[3] + result[4] + result[5] + result[6] + result[7] + t[8] * mat2.p_weight[k * 9 + index_channel * size2 + 8];
                matMul.Matrix[Mulcounter + size3] += sum;
                Mulcounter++;
            }
        }
    }
    return true;
};
```

这里采用滑动窗口的方式，i和j表示3*3卷积核的中心，若i和j处于边缘位置，且pad为1，则需要进行补零操作，由setZero函数完成。核心思想是，用一个float数组卷积核将在mat1上扫描的得到的矩阵数据装载，再与weight矩阵点乘，若要补零，则通过判断将float数组对应位置置0，只是逻辑上补零，并未更改mat1。此处采用了访存优化和SIMD指令加速。且吸取教训，在循环中，如有重复用到且某层循环中不变的数据，可先分配一个变量保存它，不用每次循环都计算一遍，改善代码风格和可维护性。

4.4)gemm函数

该函数是本次Project的核心部分，贴出完整代码

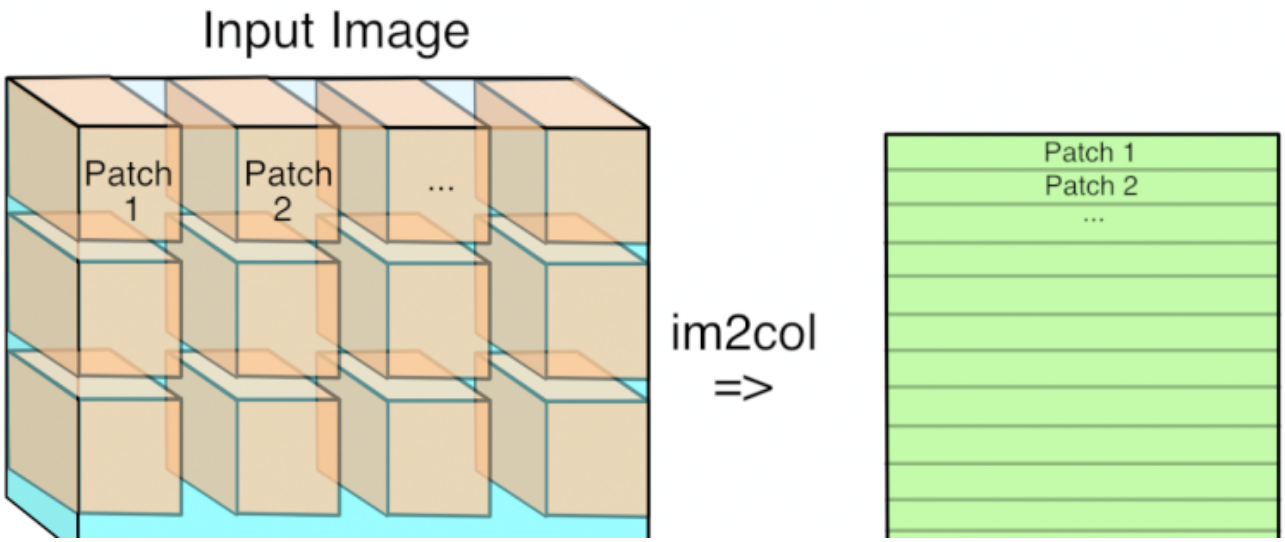
```
bool Mat_conv::gemm(const Mat_conv &mat1, const Mat_conv &gmat1, const conv_param &mat2, Mat_conv &matMul, Mat_conv &gmatMul)
{//首先对参数进行检查
    if (mat1.Matrix == NULL || mat2.p_weight == NULL || mat2.p_bias == NULL || matMul.Matrix == NULL)
    {
        coutError;
        return false;
    }
  size_t r = ((mat1.row - 3 + 2 * mat2.pad) / mat2.stride + 1);
    if (mat1.row != mat1.coloumn || r * r != gmat1.row || gmat1.coloumn != 3 * 3 * mat1.channel || gmatMul.row != r * r || gmatMul.coloumn != mat2.out_channels || matMul.row != r || matMul.coloumn != r || matMul.channel != mat2.out_channels)
    {
        cerr << "----------------------" << endl;
        cerr << "error! invalid argument list" << endl;

        cerr << "mat1 row col channel     " << mat1.row << " " << mat1.coloumn << "
```

```
" << mat1.channel << endl;
14          cerr << "gmat1 row col channel    " << gmat1.row << " " << gmat1.coloumn <<
" " << gmat1.channel << endl;
15          cerr << "matMul row col channel   " << matMul.row << " " << matMul.coloumn
<< " " << matMul.channel << endl;
16          cerr << "gmatMul row col channel  " << gmatMul.row << " " << gmatMul.coloumn
<< " " << gmatMul.channel << endl;
17          cerr << "file = " << __FILE__ << "   fun =  " << __func__ << "  line =  " <<
__LINE__ - 3 << endl;
18          return false;
19      }
20      size_t in_size = mat1.row;     //输入矩阵的大小
21      size_t in_channel = mat1.channel;//输入矩阵的channel
22      size_t out_size = (in_size - mat2.kernel_size + 2 * mat2.pad) / mat2.stride + 1;
23    //输出矩阵的大小
24      size_t out_channel = mat2.out_channels;//输出矩阵的channel
25      size_t size1 = mat1.row * mat1.coloumn;//矩阵换channel时指针偏移量
26      size_t size2 = mat2.kernel_size * mat2.kernel_size * in_channel;
27    //权重矩阵换channel时指针偏移量
28      int mycounter = 0;
29      size_t s;
30    //开始将mat1的数据装入矩阵gmat1
31      if (!mat2.pad)
32      {
33          s = 1;
34      }
35      else
36      {
37          s = 0;
38      }
39      for (int i = s; i < in_size - s; i += mat2.stride)
40      {
41          int flag1 = (i - 1) * in_size;//3*3卷积核第一行对应起始位置
42          int flag2 = (i)*in_size; //3*3卷积核第二行对应起始位置
43          int flag3 = (i + 1) * in_size; //3*3卷积核第三行对应起始位置
44          for (int j = s; j < in_size - s; j += mat2.stride)
45          {
46              for (int k = 0; k < mat1.channel; k++)
47              {
48    //与dot函数不同，该函数k在最内层循环，用临时矩阵gmat1的每一行装载卷积核每次移动扫描到的所有数据
49                  float t[9]{0, 0, 0, 0, 0, 0, 0, 0, 0};
50    if (mat2.pad == 1 && (i == 0 || i == in_size - 1 || j == 0 || j == in_size - 1))
51    {
52        bool flag = setZero(i, j, in_size, t, mat1, flag1, flag2, flag3, k * size1);
53        flagCheck;
54      //用内存拷贝函数将t数组的数据拷贝进临时矩阵gmat1
55        memcpy(&gmat1.Matrix[mycounter], &t, sizeof(float) * 9);
56        mycounter += 9;
57    }
58      else
59    {
60        int i1 = flag1 + j + k * size1;

61        int i2 = flag2 + j + k * size1;
```

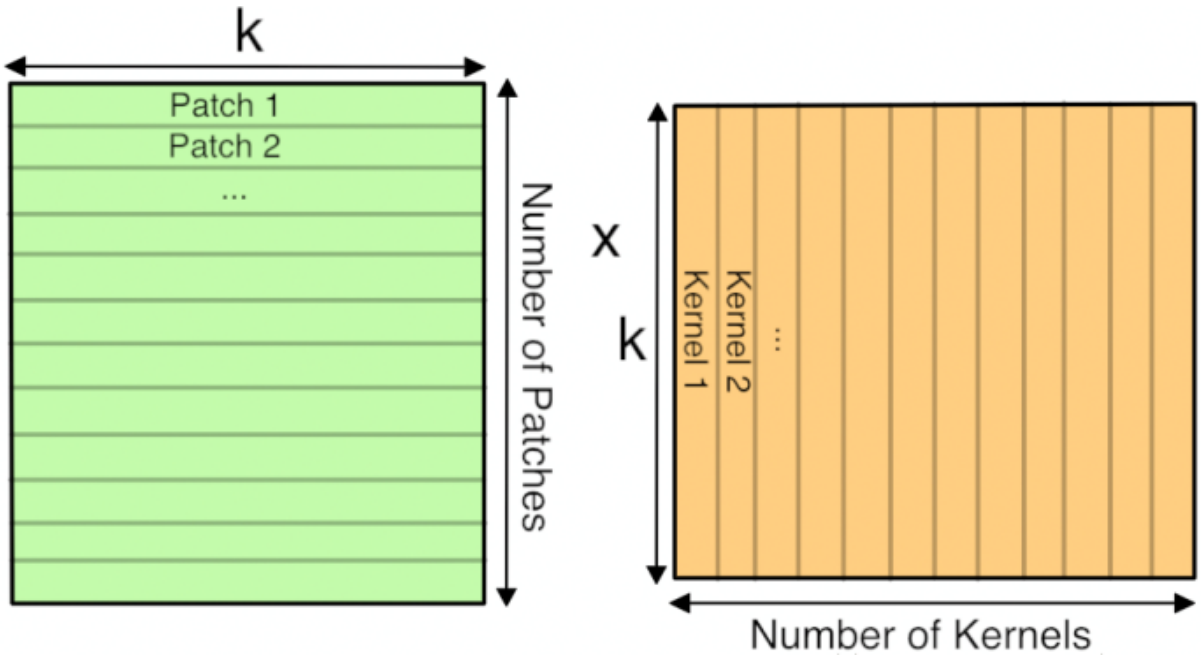
```
62      int i3 = flag3 + j + k * size1;
63      //用内存拷贝函数将t数组的数据拷贝进临时矩阵gmat1
64      memcpy(&gmat1.Matrix[mycounter], &mat1.Matrix[i1 - 1], sizeof(float) * 3);
65      mycounter += 3;
66      memcpy(&gmat1.Matrix[mycounter], &mat1.Matrix[i2 - 1], sizeof(float) * 3);
67      mycounter += 3;
68      memcpy(&gmat1.Matrix[mycounter], &mat1.Matrix[i3 - 1], sizeof(float) * 3);
69      mycounter += 3;
70                  }
71              }
72          }
73      }
74    //此处为调用openBlas库的矩阵乘法运算函数，可与自己写的矩阵乘法，矩阵点乘对比，若要调用此库，需将
75    //下方由------包住的代码片段注释掉
76    //cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasTrans, gmat1.row, out_channel,
    in_channel * 3 * 3, 1.0, gmat1.Matrix, gmat1.coloumn, mat2.p_weight, in_channel * 3
    * 3, 1.0, gmatMul.Matrix, out_channel);
77    //-------------------------------------------------------------------------
78  //开辟新的数组B，装载转置后的权重矩阵mat2
79      float *B = new float[in_channel*3*3 * out_channel];
80      for (int j = 0; j < in_channel*3*3; j++)
81      {
82          for (int i = 0; i < out_channel; i++)
83          {
84              B[j * out_channel + i] = mat2.p_weight[i * in_channel*3*3 + j];
85          }
86      }
87  //调用以往Project的做法，进行乘法运算。此处本来还想引入多线程或调用openmap，但是一采用便会段错误
    或答案错误，未能找出原因，故删除此功能,采用了访存优化与SIMD指令集加速
88      size_t len1 = gmat1.coloumn;
89      size_t len2 = out_channel;
90      for (int i = 0; i < gmat1.row; i++)
91      {
92          size_t skip = i * len2;
93          for (int k = 0; k < gmat1.coloumn; k++)
94          {
95              __m256 vector1 = _mm256_set1_ps(gmat1.Matrix[i * len1 + k]);
96              if (out_channel >= 8)
97              {
98                  for (int j = 0; j < out_channel; j += 8)
99                  {
100                     __m256 vector2 = _mm256_loadu_ps(&B[k * len2 + j]);
101                     __m256 result = _mm256_mul_ps(vector1, vector2);
102                     gmatMul.Matrix[skip + j] += result[0];
103                     gmatMul.Matrix[skip + j + 1] += result[1];
104                     gmatMul.Matrix[skip + j + 2] += result[2];
105                     gmatMul.Matrix[skip + j + 3] += result[3];
106                     gmatMul.Matrix[skip + j + 4] += result[4];
107                     gmatMul.Matrix[skip + j + 5] += result[5];
108                     gmatMul.Matrix[skip + j + 6] += result[6];
109                     gmatMul.Matrix[skip + j + 7] += result[7];
110                 }

111             }
```

```
112                 for (int j = out_channel - out_channel % 8; j < out_channel; j++)
113                 {
114                     //对非八的整数倍规模的矩阵做兼容处理
115             gmatMul.Matrix[i * len2 + j] += mat1.Matrix[i * len1 + k] * B[k * len2 + j];
116                 }
117             }
118         }
119       delete[] B;B=NULL;//对于new 出来的对象，切记当其完成任务后释放它，并将其置空
120     //-------------------------------------------------------------------------------
121     //对gmatMul再次转置，得到最终结果
122       for (int j = 0; j < out_channel; j++)
123         {
124             for (int i = 0; i < out_size * out_size; i++)
125             {
126                 matMul.Matrix[j * out_size * out_size + i] = gmatMul.Matrix[i *
    out_channel + j];
127             }
128         }
129         return true;
130     };
```

将矩阵的卷积操作转化为矩阵乘法，这一步非常的重要。因为如果采用卷积核窗口滑动的方法，只能一步一步的点乘，优化空间较小。而一但进入为矩阵乘法这个科学教钻研了许久的领域，优化空间便能大大的提高。虽然方法一和方法二的运算量相差不大，可是方法二我们可以采用许多经过极致优化后的线性代数运算库来加速我们的程序（本Project采用openblas库来加速），也可以将前几次Project的经验运用进来.更为重要的是，滑动窗口每次填入卷积核的数据内存地址是不连续的，但现在我新开的矩阵为一个一维的连续的数组，可以使得每次操作的数据内存均是连续的，cache始终保持命中。但是该方法有个缺陷，需要额外的开辟内存来存放转化后的矩阵，是一种典型的空间换时间的做法，不过在当下，硬件水平越来越高，很多情况下内存的问题不再是瓶颈。如何让程序跑的快，才是人们关注的重点。本次Project我才用了如图所示的方法来转换



如图，我们可以将矩阵卷积核每次扫过的区域（图中的patch）展开成一维向量的形式，多个这样的向量组装到一起，便形成了一个矩阵，该矩阵我在代码中用gmat1来表示。



同时，由于给出的参数矩阵本身便可以看成是每一个卷积核数据为一行的格式，只需要将其转置一次（本Pro用新开辟的数组B来存储转置后的权重矩阵），便可以直接与转换后的gmat1相乘。这样，矩阵卷积便转化为了矩阵乘法。

这里我采用了内存拷贝的方式来将原矩阵mat1转化为gamt1，更为便捷。在矩阵相乘方面，我沿用了上次Project的方法，进行访存优化和指令集加速。同时，我留下了调用openblas库的函数接口，为我们的程序加速。最后我们需要将相乘之后的结果（矩阵gmatMul）再次转置，才能得到实际上卷积后的结果。

另外，我们还有必要介绍一下调用openblas库的函数接口cblas_sgemm（）的各个参数的含义。

```
1  void cblas_sgemm(CBLAS_ORDER Order, CBLAS_TRANSPOSE TransA, CBLAS_TRANSPOSE TransB,
   blasint M, blasint N, blasint K, float alpha, const float *A, blasint lda, const float
   *B, blasint ldb, float beta, float *C, blasint ldc)
```

Order：表示矩阵展开的顺序，一般为按行展开（本Project采用这种方式）
TransA：矩阵A是否需要转置（本Pro矩阵A无需转置）
TransB：矩阵B是否需要转置（本Pro矩阵B需要转置，若此处转置，则B矩阵直接传入权重矩阵，无需提前将权重矩阵转置再传入）
M: 表示 A或C的行数。如果A转置，则表示转置后的行数
N：表示 B或C的列数。如果B转置，则表示转置后的列数。
K: 表示 A的列数或B的行数（A的列数=B的行数）。如果A转置，则表示转置后的列数
LDA：表示A的列数，与转置与否无关
LDB：表示B的列数，与转置与否无关
LDC：始终=N
alpha，beta： C=alpha ·A ·B+beta · C，一般取alpha等于1，beta等于0

到此，顺利完成最重要的卷积部分

4.5)setZero函数

```
bool Mat_conv::setZero(int i, int j, size_t in_size, float (&t)[9], const Mat_conv &mat1, int flag1, size_t flag2, size_t flag3, size_t Ksize1)
```

setZero函数的定义如上，该函数功能单一，仅做简单介绍。在进入函数后，由于i 和 j是int 类型，先对其进行检查，无误后根据八种可能的临界位置，用以一系列的if else语句将t对应的位置填上0。

4.6)setBias函数

```
bool Mat_conv::setBias(const Mat_conv &mat, float *(&bias))
```

该函数功能简单，经过检验mat和bias不为空指针后，将所得结果加上bias。

4.7)ReLu函数

```
float f=mat.Matrix[i * mat.coloumn + j + size];
 mat.Matrix[i * mat.coloumn + j + size] = f>0?f:0;
```

该函数思路会在Part4说明。

4.8)maxPool函数

```
bool Mat_conv::maxPool(const Mat_conv &mat, const Mat_conv &maxPool, float *(&bias))
{
    if (mat.Matrix == NULL || maxPool.Matrix == NULL)
    {
        coutError;
        return false;
    }
    if (mat.row / 2 != maxPool.row || mat.coloumn / 2 != maxPool.coloumn || mat.channel != maxPool.channel)
    {
        //do something and return fasle
    }
    size_t row = mat.row;
    size_t col = mat.coloumn;
    size_t size = row * col;
    size_t channel = mat.channel;
    size_t mycounter = 0;
    float a, b, c, d;
    int i1, i2;
    for (int k = 0; k < channel; k++)
    {
        float adder = bias[k];
        for (int i = 0; i < row; i += 2)
        {
            for (int j = 0; j < col; j += 2)
            {
                i1 = i * row + j + k * size;
```

```
                i2 = (i + 1) * row + j + k * size;
                a = mat.Matrix[i1];
                b = mat.Matrix[i1 + 1];
                c = mat.Matrix[i2];
                d = mat.Matrix[i2 + 1];
                float f = max(max(a, b), max(c, d)) + adder;//池化与增加bias
                maxPool.Matrix[mycounter++] = f>0?f:0;
            }
        }
    }
    return true;
}
```

由于本次Pro前两个层之间都可以将setBias，ReLu和maxPool的操作结合在一起，所以这里的maxPool函数同时也包含了这三个功能，前两层之间的转化，只需一个maxPool，无需其他操作，减少了对矩阵数据的访问和操作，加快了速度。

5)全连接层操作

5.1)fullConnect函数

该函数较为简单，未贴出代码

```
bool Mat_conv::fullConnect(float *(&result), const Mat_conv &mat, fc_param &fullC)
```

该函数将mat和fullC（两个一维向量）相乘（由于最后一步工作量较小，并未对该向量乘法进行优化），结果存放在result中。

5.2)softMax函数

该函数较为简单，仅贴出核心部分

```
for (int i = 0; i < size; i++)
    {
        sum += exp(result[i]);
    }
    for (int i = 0; i < size; i++)
    {
        result[i] = exp(result[i]) / sum;
    }
```

该函数的作用是将两个输出转化置0和1之间且相加等于1，可以表示置信概率。

6)CNN（辅助管理上述函数）

```
bool Mat_conv::CNN(const Mat_conv &mat1, conv_param *(conv_params), fc_param
&fc_params)
```

由main函数调用此函数，此函数中进行一系列的操作，完成卷积的全过程，保持main函数的简洁性。

Part3环节将会通过改变CNN函数的代码，分别实现本Pro卷积的两种方法。同时调用openblas库进行对比。

main.cpp（交互文件）

```
int main()
{
    cv::Mat image = cv::imread("face.jpg");//读入矩阵
    Mat_conv mat(128, 128, 3);
    for (int i = 0; i < 128; i++)
    {
        for (int j = 0; j < 128; j++)
        {
            mat.Matrix[i * 128 + j] = (float)image.at<cv::Vec3b>(i, j)[0] / 255.0f;
            mat.Matrix[i * 128 + j + 128 * 128 * 1] = (float)image.at<cv::Vec3b>(i, j)[1] / 255.0f;
            mat.Matrix[i * 128 + j + 128 * 128 * 2] = (float)image.at<cv::Vec3b>(i, j)[2] / 255.0f;
        }
    }
    mat.CNN(mat, conv_params, fc_params);//卷积操作
    return 0;
}
```

Part3 - Result&Verifiction

1)结果正确性检验

```
bg score 0.007086, face score 0.992914
wyh@DESKTOP-7B4UR9S:~/project5$
```

如图，当输入图像是Pro指定的人脸图像时，结果与GitHub上文档结果一致

```
bg score 0.999996, face score 0.000004
wyh@DESKTOP-7B4UR9S:~/project5$
```

当输入图像是飞机场时，结果也一致

```
bg score 0.539383, face score 0.460617
wyh@DESKTOP-7B4UR9S:~/project5$
```

如图，当输入上述图片时，人脸概率不足50%，合理，因为未能检测到鼻子，且眼睛与口等要素点过于卡通化

```
bg score 0.483932, face score 0.516068
wyh@DESKTOP-7B4UR9S:~/project5$
```

但是我也发现一些瑕疵，如上，读入一张纯白的图像，结果置信度就比较低了，人脸概率为51%

```
bg score 0.015722, face score 0.984278
wyh@DESKTOP-7B4UR9S:~/project5$
```

最后，必须得用于老师镇楼，结果很可靠。以上，可以看出本程序的正确性是有一定保障的

补充：我后来想到，如果将这个128*128的窗口从一幅图片上扫过，步长可以设置为64或32等（越小准确度越高），计算每次为人脸的概率，将所有概率集合到一起，高于某一阈值的区域，我们可以认为这里存在人脸，这样，就是一个识别任意图片中所有存在的人脸的程序的一个雏形。但也存在一些缺陷，如不能识别像素大于128 * 128过多的人脸。

2)速度检验

测试工具：联想ThinkPad E14（6核R5）
处理器：AMD Ryzen 5 4500U with Radeon Graphics　　2.38 GHz
注明：测试均开启O3优化，均采用face.jpg作为输入，对时间进行比较，计时器采用 struct timeval结构体

方法一：滑动窗口+点乘

```
wyh@DESKTOP-7B4UR9S:~/project5$ ./opencv_test
First conv 1.274ms
Second conv 2.095ms
Third conv 0.762ms
Full connection 0.009ms
Total Time 4.268ms
```

经过三次实验，结果十分相近，没有出现突兀的数据，取时间的平均值为（4.268+4.248+4.173）/3为4.230ms

方法二：将矩阵卷积转化为矩阵乘法

```
wyh@DESKTOP-7B4UR9S:~/project5$ ./opencv_test
First conv 0.988ms
Second conv 1.488ms
Third conv 0.230ms
Full connection 0.011ms
Total Time 2.826ms
```

经过三次实验，结果十分相近，没有出现突兀的数据，取时间的平均值为（2.826+2.929+2.890）/3为2.881ms

可以发现，转化为矩阵乘法后速度比方法一要快，理论上二者速度应该是差不多的，但是实际上由于我开启了O3优化，矩阵乘法的优化空间远超方法一

补充：在几天后我由于要上ARM平台测试，所以将所有的AVX指令都删除了，我在本地又测试了几次，惊奇的发现不采用AVX指令集加速程序居然跑的更快！如下图.经过多次测试，平均速度达到了1.86ms

```
wyh@DESKTOP-7B4UR9S:~/project5$ ./opencv_test
First conv  0.821ms
Second conv  0.829ms
Third conv  0.133ms
Full connection  0.011ms
Total Time 1.794ms
bg score 0.007086, face score 0.992914
```

这让我比较意外，在经过一番排除后，我发现，我的AVX指令集是嵌套在这一层循环中的

```
1    for (int j = 0; j < out_channel; j += 8)
```

由于本次卷积操作的数据比较小，尤其是out_channel，三次分别为16，32，32，所以，一共只调用了四次AVX指令集运算。在本程序运行速度为毫秒级别的情况下，仅仅做32次运算，就要进行：申请寄存器变量，将数据填充入向量中，相乘得到答案，再将答案取出并赋值给gmatMul等一系列操作，显然是有些大题小作了，拖慢了程序的速度。所以，一切的优化都要视具体情况而定，在某些情况下，做了优化而增加的代价超过了优化带来的收益，就得不偿失了。但是，本程序还是沿用了这一套指令，原因有两个，修改比较麻烦且如果出现较大的计算量时AVX指令还是可以派上用场的，同时，留下了朴素算法（被注释），用户可自行选择。

方法三：调用OpenBlas库进行加速

```
wyh@DESKTOP-7B4UR9S:~/project5$ ./opencv_test 2>error.txt
First conv  0.901ms
Second conv  0.379ms
Third conv  0.068ms
Full connection  0.01ms
Total Time 1.358ms
bg score 0.007086, face score 0.992914
```

经过三次实验，取时间平均值为（1.358+1.31+1.469）/3为1.379ms.但是有一个问题，就是我调用openblas库运算后，时间会变得比较不稳定，快的时候为1ms左右，慢的时候会达到10ms甚至以上。我验证后发现，拖慢时间的永远是第一层卷积，其它两层和全连接层耗时几乎每次都一样。但是由于时间问题，我未能查明原因。

以上，可以看出调用了线性代数运算库之后，速度提升了一半以上，这也是为什么要将卷积做成矩阵乘法的原因。

3)ARM平台检验

ARM平台不支持AVX指令集，由于时间原因，我并没有重写ARM的指令集，仅作了访存优化。同时，我在上ARM测试时也遇到了一些问题，在此记录一下。

- 由于ARM平台上没有cmake，所以要自己安装cmake，推荐安装到一个新建的文件夹中，与要注意安装cmake的版本不能低于传入的CMakecache.txt中指定的版本
- 在本地将Project5的文件传输到远端服务器后，在远端服务器输入cmake会报错，原因是因为cmake第一次编译是在本地的环境下进行的，而在远端再次编译时，CMakeCache.txt 文件路径和编译文件记录的 CMakeCache.txt 路径不一样。这时，我们只需删除CMakeCache.txt文件，再次编译即可
- 在远端服务器上，由于没有UI，对于代码的修改我们可以通过文本编辑器vim完成

  下图为在ARM上运行的结果（图像：face.jpg）

```
[WYH@ecs001-0003 project5]$ ./opencv_test
First conv  1.369ms
Second conv  1.836ms
Third conv  0.282ms
Full connection  0.012ms
Total Time 3.499ms
bg score 0.007086, face score 0.992914
```

不出意料，在X86上平均时间为1.86ms时间的程序，在ARM上要慢上87%左右（平均时间为3.490ms）。这是因为ARM的指令集更为精简，优势是低功耗，而X86的优势是高性能。补充：于老师强调了Relu层的if语句，通过查阅为什么if语句效率不高让我知道了CPU是流水线工作的。更近一步，我对两种架构也有了更深的理解。在此记录一下自己的收获

流水线工作　顾名思义就是说CPU会将要执行的指令排好队，然后按照流水线模式处理，这样可以极大提高执行效率。倘若有n个任务，流水线级数为k（即将CPU一个任务分成k个指令完成），若不采用流水线技术，则需要n*k个周期完成。但是倘若采用了流水线技术，仅需要先经过k个周期，这时流水线已经被指令装满了，再经过n-1个周期，便可完成任务，所需周期为k+n-1。这比较像并行技术，流水线各级在指令完成后，并不会闲置，而是接着处理到来的指令 **同时，CPU流水线的级数越多，任务被分的越细，那么每一级完成对应子任务的时间就越短。这样，指令一级级往下传的速度就越快，CPU的频数也就越高**
Intel X86的架构特点为：流水线较少，但是每条流水线级数很大 而 AMD架构特点为：拥有较多的流水线，但是每条生产线的级数较短

在本程序的测试中，流水线的长度占据了主导地位，所以在X86上的测试速度要比在ARM上快。

但是，流水线级数的增加必然导致的是管理难度的增大。倘若某一级出现了错误，要一直到最后一级发现结果不对才意识到出错了。所以只好改进分支预测技术以及增大缓存来改善此问题，但收效甚微，反而由于缓存的增加和漏电流控制不利，使得功耗与发热大大增加，这也是ARM功耗比Intel低的原因。

本程序并未在ARM上调用openblas来加速运算，原因是在远端服务器显示无法找到这个库，最终未能实现。

4)安全性能测试（内存管理,调试信息）

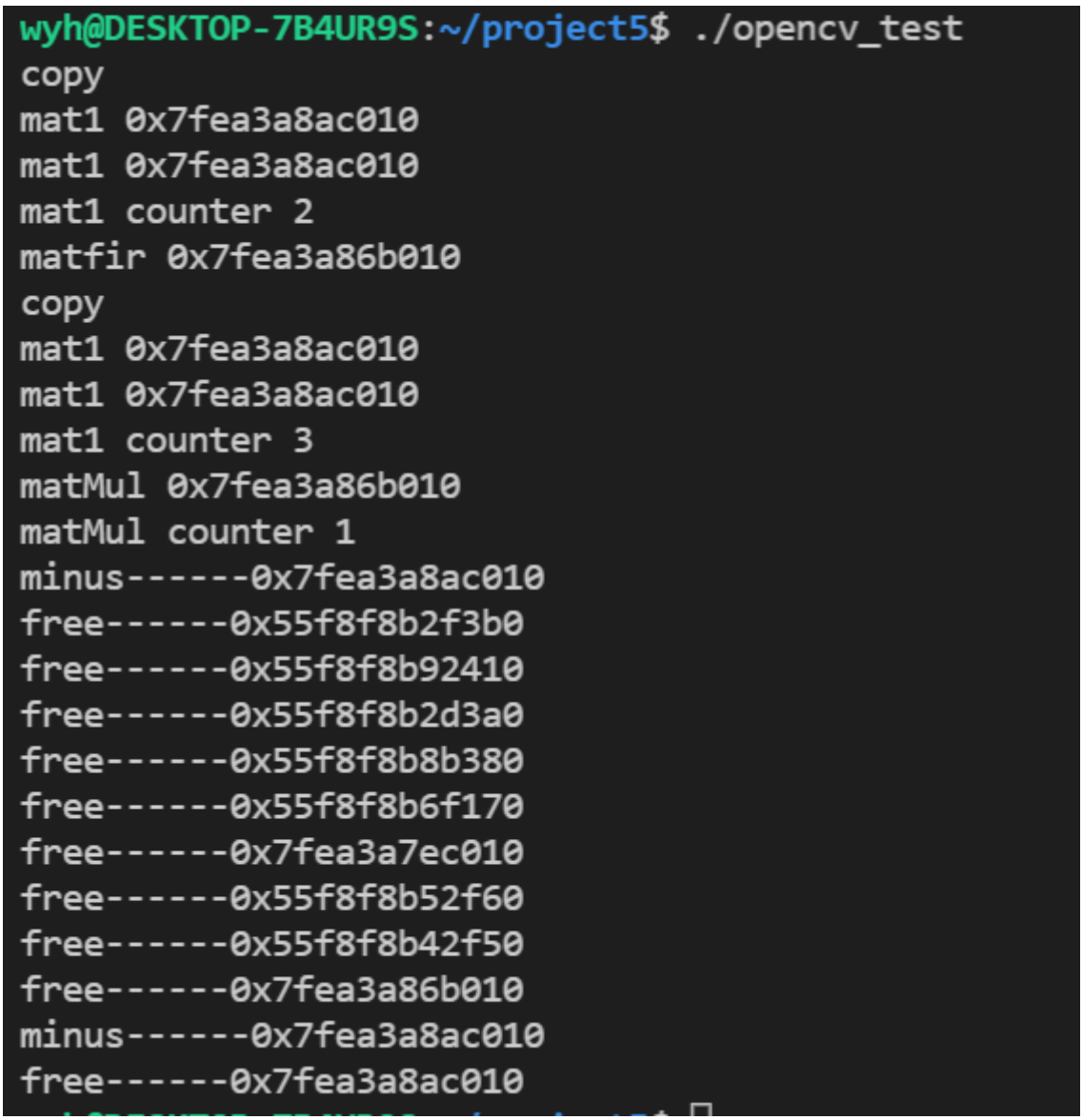
- **内存管理**

  测试采用方法一来进行卷积运算，采用打印内存地址的方式予验证，用到的相关函数如下

```
//在main.cpp 中调用func.cpp中的CNN函数
mat.CNN(mat, conv_params, fc_params);
//在拷贝构造函数中，增添打印语句
 cout<<"copy"<<endl;
 cout<<"mat1 "<<hex<<(void*)(Mat.Matrix)<<endl;
```

```
bool Mat_conv::CNN(const Mat_conv mat1, conv_param *(conv_params), fc_param &fc_params){//do something
Mat_conv matfir(64, 64, 16);//开辟新的对象（matfir）储存第一层卷积后的结果
    cout<<"mat1 "<<hex<<(void*)(mat1.Matrix)<<endl;//mat1采用了拷贝构造，返回时将会被析构
    cout<<"mat1 counter "<<*mat1.counter<<endl;//打印mat1 counter的值
    cout<<"matfir "<<hex<<(void*)(matfir.Matrix)<<endl;//打印matfir成员变量Matrix地址
    flag = matfir.conv(mat1, conv_params[0], matfir);//调用conv函数
}
```

```
bool Mat_conv::conv(const Mat_conv mat1, const conv_param &mat2, Mat_conv &matMul)
{
    cout<<"mat1 "<<hex<<(void*)(mat1.Matrix)<<endl;//mat1采用了拷贝构造，返回时析构一次
    cout<<"mat1 counter "<<*mat1.counter<<endl;//打印mat1的counter
    cout<<"matMul "<<hex<<(void*)(matMul.Matrix)<<endl;//matMul为引用，未调用拷贝构造
    cout<<"matMul counter "<<*matMul.counter<<endl;//应用传递，返回时不析构
}
```



如图，在main函数中调用CNN函数，由于CNN参数列表第一个参数非引用类型，调用了拷贝构造函数，打印“copy”与被拷贝对象的Matrix的首地址。紧接着，CNN中打印拷贝出来的mat1的Matrix首地址，可以看到，与被拷贝对象共享地址，且打印其counter为2。在CNN函数中新建了matfir后，调用conv函数，同理，mat1又被拷贝构造了一份，counter为3，而matfir由于传递的是引用，counter仍然为1，conv函数中的matMul其实就是CNN函数中的matfir。在conv函数返回时，mat1被析构一次，打印minus，在函数CNN返回时，CNN中创建的新对象全部被析构（代码未列出），可以看到，包括了matfir。这时mat1由于counter为2，再次minus，最后到了main函数返回，mat被析构。

其实，只要全部采用应用传递即可，但是这里为了显示拷贝构造函数和析构函数是没有问题的，写了这段测试代码。在源代码中，所有的函数参数均采用引用传递。

- **调式信息**

  本程序通过在func.cpp文件修改CNN函数的内容来完成卷积的操作。考虑到实际问题，用户在CNN中调用函数输入的参数十分容易出错，故每次调用方法时，都对这些参数都做了严格检查。下面是几个例子

```cpp
bool Mat_conv::CNN(const Mat_conv mat1, conv_param *(conv_params), fc_param &fc_params)
{
  //do something
    Mat_conv matfir(63, 64, 16);//正确的应为64, 64, 16
    Mat_conv gmat1(64 * 64, 3 * 3 * 3, 1);
    Mat_conv gmatfir(64 * 64, 16, 1);
    flag = matfir.gemm(mat1, gmat1, conv_params[0], matfir, gmatfir);
  flagcheck;
  //do something
}
```

例如：用户在调用gemm函数时，输入的参数matfir（用于接受卷积结果的矩阵）的行数打错了，运行结果如图，程序打印错误提示，并将错误信息出入到了error.txt文件，将其与标准输出分开，便于阅读

```
wyh@DESKTOP-7B4UR9S:~/project5$ make
Scanning dependencies of target opencv_test
[ 33%] Building CXX object CMakeFiles/opencv_test.dir/func.cpp.o
[ 66%] Building CXX object CMakeFiles/opencv_test.dir/main.cpp.o
[100%] Linking CXX executable opencv_test
[100%] Built target opencv_test
wyh@DESKTOP-7B4UR9S:~/project5$ ./opencv_test 2>error.txt
Convolution failed, please see error.txt
```

project5 > error.txt

```
1  ---------------------
2  error! invalid argument list
3  mat1 row col channel     128 128 3
4  gmat1 row col channel    4096 27 1
5  matMul row col channel   63 64 16
6  gmatMul row col channel  4096 16 1
7  file = /home/wyh/project5/func.cpp   fun =  gemm  line =  208
8  ---------------------
9  error! return value is false
10 file = /home/wyh/project5/func.cpp   fun =  CNN  line =  590
11
```

可以见到，方法不断从栈里弹出，同时也回溯地将每一次的错误由深至浅打印了出来。如果发现错误是由前面的函数对参数检查不细致造成的，可以增强前面函数的鲁棒性，从源头便消除错误。

Part4 - Difficult&Solution

- 本程序最难的地方就在于如何将矩阵的卷积对应成矩阵的乘法（点乘或者叉乘），这需要将原矩阵中的每个像素点按照一定的规则，一个一个抠出来参与运算，又或者一个一个抠出来组成一个新的矩阵。同时，还有池化，激活等一系列操作。所以编写的时候头脑一定要清晰，最好能够画一张导图，理清关系。我在编写的时候，遇到的最大困难就是算不出正确的答案。幸好有同学将每层运算后的结果打包发送到了QQ群中，我将自己的结果一步步的与他对比，不断地调试自己的程序，寻找可能的bug
- 该Pro有一个激活的操作，采用ReLu函数。于老师上课时留下了一个问题，能不能不用if?

首先要明确为什么不能用if？原因是因为CPU的执行特点，简单来说就是当CPU在处理当前指令的时候，后面已经有N条指令在后面排队等待你去执行了。这样当你执行完一条马上执行下一条而不用去找，这比较像cache的机制。若程序是顺序结构，那再好不过，直接排好队。但是如果用了if，就会打破这种顺序结构。程序再没运行到if时，它并不知道下一条指令，所以只得执行完后根据实际情况去寻找，拖慢了速度。

我一开始想通过位运算，右移31位得到符号位，再做逻辑运算。但是这需要将float先转为int，且我自己测试表明这样做比if更慢。

后来我想到了如下代码

```
1  data=(data>0)*data;
```

data>0是一个无if结构，但是*操作符速度较慢。经过测试，速度还是慢于if。

接着，我测试发现，但三目运算符是会将两条分支的指令都加入队列的，花费多一次计算的时间，但省去了找下一条指令的时间。但是这没关系，CPU擅长的是计算，我们需要做的就是尽量将指令全部排好队。本机测试结果显示用三目运算符时间和if非常接近，查阅得知，三目的汇编语言比if少一条语句，我猜想可能编译器优化了对if进行了优化

最终，我未能想到一个很好的优化方法，暂时沿用了三目运算符，因为他能避开if拖慢程序速度的最本质原因。

Part5 - Areas where this project can be improved

本次Project不足之处和可以提升的地方较多，大致分为这几个方面

- 支持的卷积操作十分受限，只能支持卷积核大小为3*3的权重矩阵，只能支持pad为1或者0的权重矩阵，下一步的优化目标可以为能够兼容多种形式的卷积
- 在采用矩阵乘法进行卷积时，需要开辟新的空间，可以思考有无更好的办法，无需开辟新的空间。同时，采用矩阵乘法卷积涉及到矩阵的转置操作，由于时间问题，本次Project只进行了基础的优化（方式为读取数据时cache不命中，写入数据时cache命中，经过比较该方式比读取连续，写入不连续要快）。下一次优化目标可以针对矩阵的转置进行优化。
- 平台兼容性不足，AVX指令集在ARM平台上无法使用，时间问题未能加入neon指令集

Part6 - Summary

这次的主题一开始让我觉得无从下手，但实际上了解了后，感觉也没那么难，但是在实现过程中，bug还是不断出现的，不过最后总算是都解决了。这次的Project让我对卷积神经网络的模型有了一个基本的了解并有了手动实现的经历，感觉还是很棒的。在人工智能如此火爆的当下，我们应当少说空话，少吹各种高大上的名词，而应多动手实现，多些深入思考，多些脚踏实地！这样才会有更多的收获。

写在最后：第一次报告写了这么多字，最后一个Project希望可以做的好一些，也感谢老师能够看完。在这次Project中，感谢解答过我问题的同学，学助，以及于老师一个学期的付出！