Project 4: A Class for the Data Blobs in Convolutional Neural Networks

Name: Bryan Anthony
SID: 12112230

Environtment:

- Ubuntu 22.04.1 LTS
- g++ (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0
- AMD A8-7410 processor
- 8GB memory
- x86 platform

Part 1 - Analysis

In this project I need to recreated the same idea for convolutional layer in the previous chapter using cpp. I required to design a class for the data blobs in Convolutional Neural Networks. It can be used for the input, output and kernels of a convolutional layer. The class should contain the data of a data blob and related information such the number of rows, the number of columns, the number of channels, etc.

I implement this class with memory management like CvMat struct using refcount. The basic idea is to keep track of the number of references (i.e., pointers) that point to an object. When an object is created, its reference count is set to 1. Every time a new reference to the object is created or when assign from object to object with softcopy, the reference count is incremented. When a reference to the object is deleted (i.e., the pointer goes out of scope, or is set to null), the reference count is decremented. When the reference count reaches zero, the object is no longer being used, and it can be safely deleted.

In my code I used softcopy for the copy constructor and assign operator. I'm also implement some frequently used operators including but not limited to =, ==, +, -, *, etc.

What special's in Datablob class:

1. Do mathematic operation +,-,/,*
2. can check two DataBlob is equal or not. using operator == and !=
3. can do assign =, and also there's copy constructor
4. can do all those operation above when two Datablobs are not the same type
5. There's relu and add padding function
6. optimization using omp

Here's a brief description of each operator and function and their return types and behaviors in class DataBlob:

- `getRows()`, `getCols()`, `getChannels()`, `getRefCount()`: These are getter functions that return the respective private data members of the class. They have a return type of `size_t` and do not modify the class data.

- `generateRandomMatrix()`: This function generates random data for the `data` member of the class. It doesn't return anything, but it modifies the `data` member.
- `operator+=`: This operator adds the data in two `DataBlob` objects and stores the result in the first object. It returns a reference to the modified object. If one or both of the `DataBlob` objects are empty, it prints an error message and returns the first object.
- `operator=`: This operator assigns the values of one `DataBlob` object to another. It returns a reference to the modified object. If the two objects are the same, it returns the object. Otherwise, it releases the resources held by the current object, makes a soft copy of the data in from other object, and increments the reference count.
- `operator==`: This operator checks if two `DataBlob` objects are equal. It returns `true` if the objects have the same dimensions and data, and `false` otherwise.
- `operator!=`: This operator checks if two `DataBlob` objects are not equal. It returns `true` if the objects have different dimensions or data, and `false` otherwise.
- `operator DataBlob<V>()`: This is a conversion operator that converts a `DataBlob` object to a `DataBlob` object with a different data type `V`. It returns the converted object. It creates a new `DataBlob` object with the same dimensions as the original object and copies the data from the original object, casting it to the new data type.
- `operator-`: This operator subtracts the data in two `DataBlob` objects and returns a new `DataBlob` object containing the result. If one or both of the `DataBlob` objects are empty, it prints an error message and returns an empty `DataBlob` object. If the two objects have different dimensions, it prints an error message and returns an empty `DataBlob` object.
- `operator+`: This operator adds the data in two `DataBlob` objects and returns a new `DataBlob` object containing the result. If one or both of the `DataBlob` objects are empty, it prints an error message and returns an empty `DataBlob` object. If the two objects have different dimensions, it prints an error message and returns an empty `DataBlob` object.
- `operator*`: This operator multiplies the data in two `DataBlob` objects element-wise and returns a new `DataBlob` object containing the result. If one or both of the `DataBlob` objects are empty, it prints an error message and returns an empty `DataBlob` object. If the two objects have different dimensions, it prints an error message and returns an empty `DataBlob` object.
- `T &operator()(int row, int col, int channel)` - This operator returns a reference to the element in the data array at the specified row, column, and channel indices. It allows the element to be accessed and modified.
- `const T *operator()(int row, int col, int channel) const` - This operator returns a reference to the element in the data array at the specified row, column, and channel indices. It allows the element to be accessed without modified.
- `T *operator+(int i)` - This operator returns a pointer to the element in the data array that is positions after the current pointer position.
- `const T *operator+(int i) const` - This operator returns a constant pointer to the element in the data array that is i positions after the current pointer position. It allows access to the data without modification.
- `T &operator()(int ind)` - This operator returns a reference to the element in the data array at the specified index. It allows the element to be accessed and modified.
- `const T &operator()(int ind) const` - This operator returns a constant reference to the element in the data array at the specified index. It allows access to the data without modification.

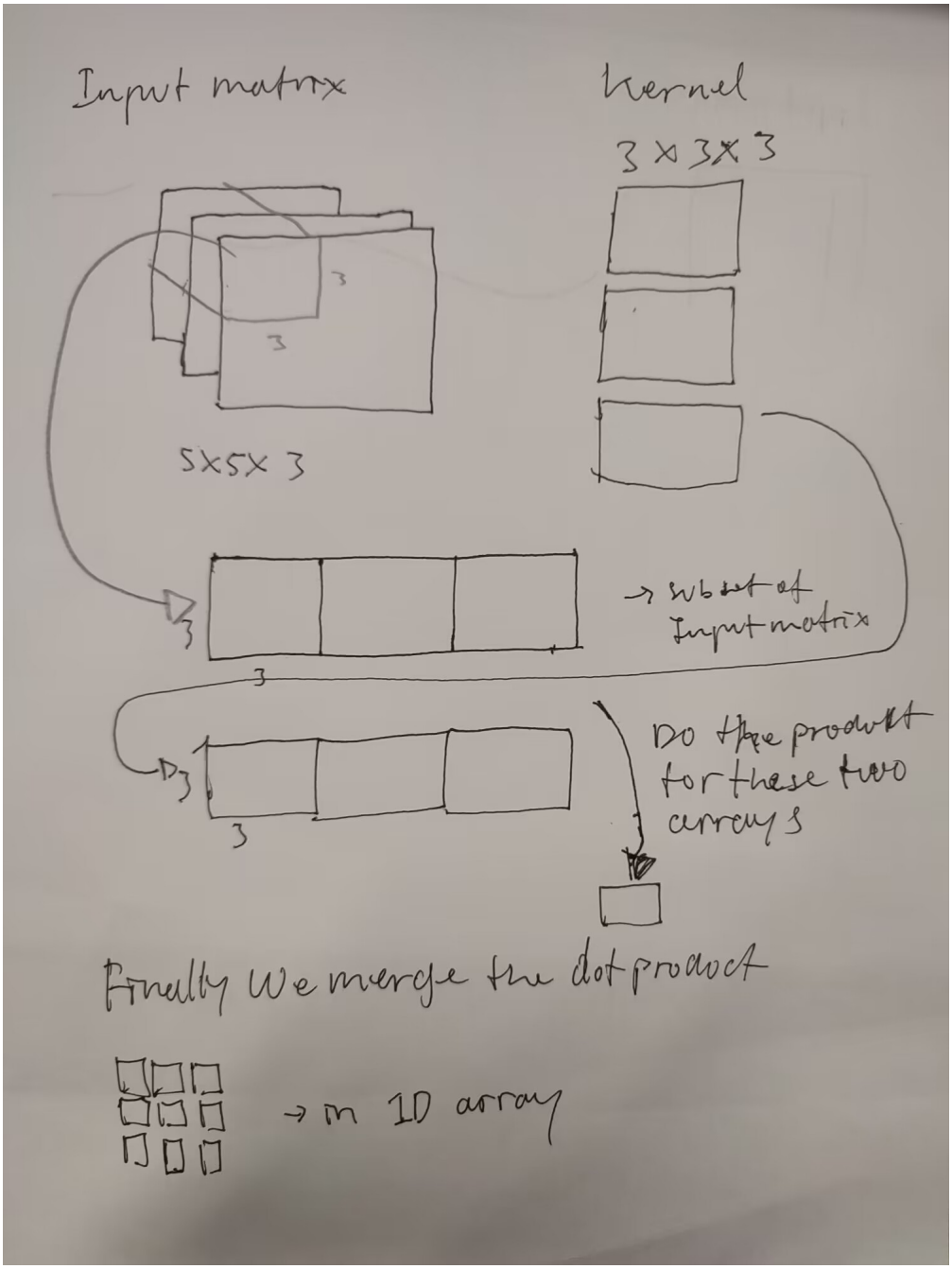
- `operator<<`: This is the insertion operator that is used to overload the `<<` operator to output the data in a `DataBlob` object to an output stream. The `friend` keyword is used to allow this operator to access private members of the `DataBlob` class. The function takes two arguments: an output stream and a reference to a `DataBlob` object. It first checks if the `DataBlob` object is empty by calling the `isEmpty()` function. If the object is empty, it outputs an error message to the output stream and returns the output stream. Otherwise, it loops through each element in the `data` array of the `DataBlob` object and outputs it to the output stream. If the element is the last element in a row, it also outputs a newline character. The function returns the output stream.
- `isEmpty()`: This is an inline member function of the `DataBlob` class that checks if a `DataBlob` object is empty. It returns `true` if any of the following conditions are met: `rows` is less than or equal to 0, `cols` is less than or equal to 0, `channels` is equal to 0, `data` is equal to `nullptr`, or `refcount` is equal to `nullptr`.
- `add_padding()`: This is a member function of the `DataBlob` class that adds padding to a `DataBlob` object. It takes a single argument, `padding`, which specifies the amount of padding to add. If the `DataBlob` object is empty, it returns a new empty `DataBlob` object. The function creates a new `DataBlob` object with the same number of channels as the original object, but with `padding` rows and columns added on all sides. It then loops through each element in the original `DataBlob` object and copies it to the appropriate location in the padded `DataBlob` object. Finally, it replaces the original `DataBlob` object with the padded `DataBlob` object and returns a reference to the original object.
- `relu()`: This is a member function of the `DataBlob` class that applies the ReLU activation function to each element of a `DataBlob` object. If the `DataBlob` object is empty, it outputs an error message to `std::cerr` and returns a reference to the original object. The function loops through each element in the `data` array of the `DataBlob` object and replaces it with its value if it is greater than 0, or with 0 if it is less than or equal to 0. Finally, it returns a reference to the original object.

Here's the function outside the class DataBlob

- The first function, `dotproduct()`, is a template function that calculates the dot product between two `DataBlob<T>` objects, where `T` is the data type of the elements in the `DataBlob`. The function takes two `DataBlob<T>` objects and a size `n` as input arguments and returns the dot product of the two `DataBlob<T>` objects as a value of type `T`. The function iterates over the `n` elements of the two `DataBlob<T>` objects and multiplies the corresponding elements of the two `DataBlob<T>` objects and adds the result to a running total. The final running total is returned as the result of the function.
- The second function, `convolutional()`, is a template function that performs a ND convolution between two `DataBlob<T>` objects, where `T` is the data type of the elements in the `DataBlob`. The function takes two `DataBlob<T>` objects as input arguments, an `inputMatrix` and a `kernel`, and returns a `DataBlob<T>` object as the output of the convolution operation. The function first checks if either the `inputMatrix` or the `kernel` is empty, and if so, it returns an empty `DataBlob<T>` object. Next, the function calculates the dimensions of the output `DataBlob<T>` object based on the dimensions of the `inputMatrix` and the `kernel`. The function then creates a `DataBlob<T>` object called `tempMatrix` with the same dimensions as the `kernel`, which is used to perform the convolution operation on a subset of the `inputMatrix`. The function iterates over the subset of the `inputMatrix` and fills in the elements of `tempMatrix` with the corresponding subset of the `inputMatrix`. The function then calls the `dotproduct()` function to compute the dot product between `tempMatrix` and `kernel` and stores the result in the corresponding element of the output `DataBlob<T>` object. The function repeats this process for each subset of the `inputMatrix` and fills in the elements of the output `DataBlob<T>` object with the results of the convolution operation. Finally, the function returns the output `DataBlob<T>` object.

I made the cnn.cpp file is for DataBlob that has a type of float because I can do optimazation using omp when do many operations. And also there is **generateRandomMatrix** for int and float Datablob type.

Convolutional Ilustration:



Part 2 - Code

There are three file in total main.cpp cnn.hpp cnn.cpp

```
#include <iostream>
#include <chrono>
```

```
#include "cnn.hpp"
using namespace std;
#define TIME_START start = std::chrono::steady_clock::now();
#define TIME_END                                                                    \
    end = std::chrono::steady_clock::now();                                         \
    duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count(); \
    cout << "took time = " << duration << "ms" << endl;
int main(int argc, char const *argv[])
{
    std::chrono::steady_clock::time_point start, end;
    long long duration;

    // DataBlob<float> fmat(5, 5, 1);
    // fmat.generateRandomMatrix();
    // cout << "fmat :" << endl;
    // cout << fmat;
    // DataBlob<int> imat(5, 5, 1);
    // imat.generateRandomMatrix();
    // cout << "imat :" << endl;
    // cout << imat;
    // // DataBlob<int> res = fmat + imat;
    // // cout << res;
    // // cout<<fmat;

    // fmat = fmat * imat;
    // cout << fmat;

    int cols, rows, channels;
    cin >> cols >> rows >> channels;
    int matrixSize = cols * rows * channels;
    DataBlob<int> originalMatrix(cols, rows, channels);
    // random generated matrix

    // originalMatrix.generateRandomMatrix();

    // input matrix from by CLI
    for (size_t i = 0; i < matrixSize; i++)
    {
        cin >> originalMatrix(i);
    }
    int kcols, krows, numofkernel;
    cin >> numofkernel;
    cin >> kcols >> krows;
    DataBlob<float> *kernelMatrix = new DataBlob<float>[numofkernel];
    int kSize = kcols * krows * channels;

    for (size_t i = 0; i < numofkernel; i++)
    {

        kernelMatrix[i] = DataBlob<float>(krows, kcols, channels);
        // random generated matrix
        // kernelMatrix[i].generateRandomMatrix();
        // input matrix from by CLI
        for (size_t j = 0; j < kSize; j++)
        {
            cin >> kernelMatrix[i](j);
        }
    }
    // cout << originalMatrix.add_padding(1);

    int outputcols = originalMatrix.getCols() - kcols + 1;
    int outputrows = originalMatrix.getRows() - krows + 1;
    DataBlob<float> outputMatrix(outputcols, outputrows, 1);
    TIME_START
    for (int i = 0; i < numofkernel; i++)
    {
        outputMatrix += convolutional<float>(originalMatrix, kernelMatrix[i]);
    }
    TIME_END
    // cout << outputMatrix.relu();
```

```
    DataBlob<int> res;
    res = outputMatrix;
    cout << res;
    cout << "outputMatrix and res is not the same : " << (outputMatrix != res) << endl;
    delete[] kernelMatrix;
    return 0;
}
```

```
#pragma once
#include <cstdlib>
#include <cstring>
#include <iostream>

template <typename T>
class DataBlob
{
private:
    size_t rows;
    size_t cols;
    size_t channels;
    size_t *refcount;
    T *data;

public:
    size_t getRows() const
    {
        return rows;
    }

    size_t getCols() const
    {
        return cols;
    }

    size_t getChannels() const
    {
        return channels;
    }

    size_t getRefCount() const
    {
        return *refcount;
    }

    DataBlob(size_t _cols = 0, size_t _rows = 0, size_t _channels = 0)
        : rows(_rows),
          cols(_cols),
          channels(_channels),
          refcount(new size_t(1))

    {

        int ret1 = posix_memalign((void **)&data, 256, _rows * _cols * _channels * sizeof(T));

        if (ret1 != 0)
        {
            std::cerr << "memory aligned failed" << std::endl;
            return;
        }
        std::memset(data, 0, _rows * _cols * _channels * sizeof(T));
    }
    DataBlob(const DataBlob &other)
    {

        rows = other.rows;
        cols = other.cols;
        channels = other.channels;
```

```
        data = other.data;
        refcount = other.refcount;

        // Increment the reference count
        (*refcount)++;
    }
    void generateRandomMatrix();
    ~DataBlob()
    {

        if (--(*refcount) == 0)
        {
            free(data);
            delete refcount;
        }
    }
    DataBlob<T> &operator+=(const DataBlob<T> &other)
    {
        if (isEmpty() || other.isEmpty())
        {
            std::cerr << "data blob is empty" << std::endl;
            return *this;
        }
        *(this) = *(this) + other;
        return *this;
    }

    DataBlob<T> &operator=(const DataBlob &other)
    {

        if (this != &other)
        {
            this->~DataBlob();
            rows = other.rows;
            cols = other.cols;
            channels = other.channels;
            data = other.data;
            refcount = other.refcount;
            (*refcount)++;
        }
        return *this;
    }
    // DataBlob &operator=(DataBlob &&other);\

    bool operator==(const DataBlob &other) const
    {

        if (rows != other.rows || cols != other.cols || channels != other.channels)
        {
            return false;
        }
        return std::memcmp(data, other.data, rows * cols * channels * sizeof(T)) == 0;
    }
    bool operator!=(const DataBlob &other) const
    {

        return !(*this == other);
    }

    template <typename V>
    operator DataBlob<V>() const
    {
        DataBlob<V> result(rows, cols, channels);
        // Perform the type conversion here
        for (size_t i = 0; i < rows * cols * channels; ++i)
        {
            result(i) = static_cast<V>(this->data[i]);
        }
        return result;
    }
```

```
DataBlob<T> operator-(const DataBlob<T> &other) const
{
    if (isEmpty() || other.isEmpty())
    {
        std::cerr << "DataBlob is empty" << std::endl;
        return DataBlob<T>();
    }

    if (rows != other.rows || cols != other.cols || channels != other.channels)
    {
        std::cerr << "size is not the same!" << std::endl;
        return DataBlob<T>();
    }

    size_t n = rows * cols * channels;
    DataBlob<T> result(cols, rows, channels);
    size_t startIndex = 0;

    for (size_t i = startIndex; i < n; i++)
    {
        result.data[i] = data[i] - other.data[i]; // subtraction instead of addition
    }

    return result;
}

DataBlob<T> operator+(const DataBlob &other) const
{
    if (isEmpty() || other.isEmpty())
    {
        std::cerr << "DataBlob is empty" << std::endl;
        return DataBlob<T>();
    }

    if (rows != other.rows || cols != other.cols || channels != other.channels)
    {
        std::cerr << "size is not the same!" << std::endl;
        return DataBlob<T>();
    }
    size_t n = rows * cols * channels;
    DataBlob<T> result(cols, rows, channels);
    size_t startIndex = 0;
    for (size_t i = startIndex; i < n; i++)
    {

        result.data[i] = data[i] + other.data[i];
    }

    return result;
}

DataBlob<T> operator/(const DataBlob &other) const
{
    if (isEmpty() || other.isEmpty())
    {
        std::cerr << "DataBlob is empty" << std::endl;
        return DataBlob<T>();
    }

    if (rows != other.rows || cols != other.cols || channels != other.channels)
    {
        std::cerr << "size is not the same!" << std::endl;
        return DataBlob<T>();
    }

    size_t n = rows * cols * channels;
    DataBlob<T> result(cols, rows, channels);
    size_t startIndex = 0;
    for (size_t i = startIndex; i < n; i++)
```

```
        {
            result.data[i] = data[i] / other.data[i];
        }

        return result;
    }

    DataBlob<T> operator*(const DataBlob &other) const
    {
        if (isEmpty() || other.isEmpty())
        {
            std::cerr << "DataBlob is empty" << std::endl;
            return DataBlob();
        }
        if (rows != other.rows || cols != other.cols || channels != other.channels)
        {
            std::cerr << "size is not the same!" << std::endl;
            return DataBlob();
        }
        DataBlob result(cols, rows, channels);
        size_t n = rows * cols * channels;
        size_t startIndex = 0;

        for (size_t i = startIndex; i < n; i++)
        {

            result.data[i] = data[i] * other.data[i];
        }
        return result;
    }

    T &operator()(size_t row, size_t col, size_t channel)
    {
        return data[row * cols * channels + col * channels + channel];
    }
    T *operator+(size_t i)
    {
        return data + i;
    }
    const T *operator+(size_t i) const
    {
        return data + i;
    }
    T &operator()(size_t ind)
    {
        return data[ind];
    }
    const T &operator()(size_t ind) const
    {
        return data[ind];
    }

    const T &operator()(size_t row, size_t col, size_t channel) const
    {
        return data[row * cols * channels + col * channels + channel];
    }
    friend std::ostream &operator<<(std::ostream &output, DataBlob &dataBlob)
    {
        if (dataBlob.isEmpty())
        {
            output << "Datablob is empty" << std::endl;
            return output;
        }
        size_t mSize = dataBlob.rows * dataBlob.cols * dataBlob.channels;
        for (size_t i = 0; i < mSize; i++)
        {

            output << dataBlob.data[i] << " ";
            if ((i + 1) % dataBlob.cols == 0)
            {
```

```
                output << std::endl;
            }
        }
        return output;
    }
    inline bool isEmpty() const
    {
        return (rows <= 0 || cols <= 0 || channels == 0 || data == nullptr || refcount == nullptr);
    }
    DataBlob<T> &add_padding(size_t padding)
    {
        if (isEmpty())
        {
            return DataBlob();
        }
        size_t padded_cols = this->cols + 2 * padding;
        size_t padded_rows = this->rows + 2 * padding;
        size_t ch = this->channels;
        size_t padded_size = padded_cols * padded_rows * ch;
        DataBlob<T> padded(padded_cols, padded_rows, ch);

        for (size_t c = 0; c < ch; c++)
        {
            for (size_t y = 0; y < this->rows; y++)
            {
                for (size_t x = 0; x < this->cols; x++)
                {
                    size_t index = (c * this->rows + y) * this->cols + x;
                    size_t padded_index = ((c * padded_rows) + (y + padding)) * padded_cols + (x + padding);
                    padded(padded_index) = (*this)(index);
                }
            }
        }

        *(this) = padded;

        return *this;
    }
    DataBlob<T> &relu()
    {
        if (isEmpty())
        {
            std::cerr << "data blob is empty" << std::endl;
            return *this;
        }
        size_t len = this->cols * this->rows * this->channels;

        // prsize_tf("WITH_NORMAL\n");
        for (size_t i = 0; i < len; i++)
            (*this)(i) = (*this)(i) > 0 ? (*this)(i) : 0;
        return *this;
    }
};
template <typename T>
T dotproduct(const DataBlob<T> &p1, const DataBlob<T> &p2, size_t n);
// for not float dot product
template <typename T>
T dotproduct(const DataBlob<T> &p1, const DataBlob<T> &p2, size_t n)
{
    T sum = 0.0f;
    size_t startIndex = 0;

    for (size_t i = startIndex; i < n; i++)
        sum += (p1(i) * p2(i));
    return sum;
}

template <typename T>
DataBlob<T> convolutional(const DataBlob<T> &inputMatrix, const DataBlob<T> &kernel)
{
```

```
    if (inputMatrix.isEmpty() || kernel.isEmpty())
    {
        std::cerr << "Input Matrix or kernel is empty" << std::endl;
        return DataBlob<T>();
    }
    if (inputMatrix.getRows() < kernel.getRows() || inputMatrix.getCols() < kernel.getCols())
    {
        std::cerr << "The input of matrix size cannot be smaller than the kernal size" << std::endl;
        return DataBlob<T>();
    }

    size_t outputcols = inputMatrix.getCols() - kernel.getCols() + 1;
    size_t outputrows = inputMatrix.getRows() - kernel.getRows() + 1;
    size_t outputSize = outputrows * outputcols;
    size_t tempMatrixSize = kernel.getCols() * kernel.getRows() * kernel.getChannels();
    size_t onedtempMatrixSize = kernel.getCols() * kernel.getRows();
    DataBlob<T> outputMatrix(outputcols, outputrows, 1);
    DataBlob<T> tempMatrix(kernel.getCols(), kernel.getRows(), kernel.getChannels());

    size_t temp_matrix_index = 0;

    size_t output_matrix_index = 0;
    size_t start = 0;

    for (size_t k = 1; k <= outputrows; start++)
    {
        // take the temp matrix

        for (size_t i = start; i < (inputMatrix.getCols() * inputMatrix.getRows() * inputMatrix.getChannels()) + start;)
        {
            for (size_t j = i; j < kernel.getCols() + i; j++)
            {

                // tempMatrix.data[temp_matrix_index++] = inputMatrix.data[j];
                tempMatrix(temp_matrix_index++) = inputMatrix(j);
            }

            if (temp_matrix_index == tempMatrixSize)
            {

                temp_matrix_index = 0;
                break;
            }
            else if (temp_matrix_index % onedtempMatrixSize == 0)
            {

                i += inputMatrix.getCols() * inputMatrix.getRows() - ((kernel.getRows() - 1) * inputMatrix.getCols());
            }
            else
            {
                i += inputMatrix.getCols();
            }
        }

        // the dot product is here
        outputMatrix(output_matrix_index++) = dotproduct(tempMatrix, kernel, tempMatrixSize);
        if (start == (inputMatrix.getCols() * k) - kernel.getCols())
        {

            start += kernel.getCols() - 1;
            k++;
        }
    }
    return outputMatrix;
}
```

```
#include <random>
#include <iostream>
#include "cnn.hpp"
#ifdef WITH_AVX2
#include <immintrin.h>
#endif

#ifdef WITH_NEON
#include <arm_neon.h>
#endif

#ifdef _OPENMP
#include <omp.h>
#endif
template <>
void DataBlob<float>::generateRandomMatrix()
{
    // Seed the random number generator with the current time
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<float> distribution(0.0, 100.0);
    for (size_t i = 0; i < rows * cols * channels; i++)
    {
        data[i] = distribution(gen);
    }
}
template <>
void DataBlob<int>::generateRandomMatrix()
{
    // Seed the random number generator with the current time
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<int> distribution(0, 100);
    for (int i = 0; i < rows * cols * channels; i++)
    {
        data[i] = distribution(gen);
    }
}

template <>
DataBlob<float> DataBlob<float>::operator+(const DataBlob &other) const
{
    if (isEmpty() || other.isEmpty())
    {
        std::cerr << "DataBlob is empty" << std::endl;
        return DataBlob<float>();
    }

    if (rows != other.rows || cols != other.cols || channels != other.channels)
    {
        std::cerr << "size is not the same!" << std::endl;
        return DataBlob<float>();
    }
    size_t n = rows * cols * channels;
    DataBlob<float> result(cols, rows, channels);
    size_t startIndex = 0;
#if defined(WITH_AVX2)
    __m256 a, c;
#pragma omp parallel for
    for (size_t i = 0; i < n - (n % 8); i += 8)
    {
        a = _mm256_load_ps(data + i);
        c = _mm256_load_ps(other.data + i);
        c = _mm256_add_ps(a, c);
        _mm256_store_ps(result.data + i, c);
    }
    startIndex = n - (n % 8);
#elif defined(WITH_NEON)
```

```
    float32x4_t a, b, c;
    for (size_t i = 0; i < n - (n % 4); i += 4)
    {
        a = vld1q_f32(data + i);
        b = vld1q_f32(other.data + i);
        c = vaddq_f32(a, b);
        vst1q_f32(result.data + i, c);
    }
    startIndex = n - (n % 4);

#endif
    for (size_t i = startIndex; i < n; i++)
    {

        result.data[i] = data[i] + other.data[i];
    }

    return result;
}

template <>
float dotproduct(const DataBlob<float> &p1, const DataBlob<float> &p2, size_t n)
{
    float sum = 0.0f;
    size_t startIndex = 0;
#if defined(WITH_AVX2)
    // printf("WITH_AVX2\n");

    __m256 a, b;
    __m256 c = _mm256_setzero_ps();

#pragma omp parallel for
    for (size_t i = 0; i < n - (n % 8); i += 8)
    {
        a = _mm256_load_ps(p1 + i);
        b = _mm256_load_ps(p2 + i);
        c = _mm256_add_ps(c, _mm256_mul_ps(a, b));
    }
    startIndex = n - (n % 8);
    c = _mm256_hadd_ps(c, c);
    c = _mm256_hadd_ps(c, c);
    sum = ((float *)&c)[0] + ((float *)&c)[4];
#elif defined(WITH_NEON)
    // printf("WITH_NEON\n");

    float32x4_t a,
        b;
    float32x4_t c = vdupq_n_f32(0);
    startIndex = n - (n % 4);
    for (size_t i = 0; i < n - (n % 4); i += 4)
    {
        a = vld1q_f32(p1 + i);
        b = vld1q_f32(p2 + i);
        c = vaddq_f32(c, vmulq_f32(a, b));
    }
    sum += vgetq_lane_f32(c, 0);
    sum += vgetq_lane_f32(c, 1);
    sum += vgetq_lane_f32(c, 2);
    sum += vgetq_lane_f32(c, 3);

    // printf("WITH_NORMAL\n");

#endif
    for (size_t i = startIndex; i < n; i++)
        sum += (p1(i) * p2(i));
    return sum;
}
template <>
DataBlob<float> &DataBlob<float>::relu()
{
```

```
    if (isEmpty())
    {
        std::cerr << "data blob is empty" << std::endl;
        return *this;
    }
    int len = this->cols * this->rows * this->channels;
#if defined(WITH_AVX2)

    __m256 a, b;
    b = _mm256_setzero_ps(); // zeros
#pragma omp parallel for
    for (size_t i = 0; i < len; i += 8)
    {
        a = _mm256_load_ps(*this + i);
        a = _mm256_max_ps(a, b);
        _mm256_store_ps(*this + i, a);
    }

#elif defined(WITH_NEON)

    float32x4_t a, b;
    b = vdupq_n_f32(0.0f); // zeros
    for (size_t i = 0; i < len; i += 4)
    {
        a = vld1q_f32(*this + i);
        a = vmaxq_f32(a, b);
        vst1q_f32(*this + i, a);
    }
#else
    // printf("WITH_NORMAL\n");
    for (size_t i = 0; i < len; i++)
        (*this)(i) = (*this)(i) > 0 ? (*this)(i) : 0;
#endif
    return *this;
};

template <>
DataBlob<float> DataBlob<float>::operator-(const DataBlob &other) const
{
    if (isEmpty() || other.isEmpty())
    {
        std::cerr << "DataBlob is empty" << std::endl;
        return DataBlob<float>();
    }

    if (rows != other.rows || cols != other.cols || channels != other.channels)
    {
        std::cerr << "size is not the same!" << std::endl;
        return DataBlob<float>();
    }

    size_t n = rows * cols * channels;
    DataBlob<float> result(cols, rows, channels);
    size_t startIndex = 0;

#if defined(WITH_AVX2)
    __m256 a, c;
#pragma omp parallel for
    for (size_t i = 0; i < n - (n % 8); i += 8)
    {
        a = _mm256_load_ps(data + i);
        c = _mm256_load_ps(other.data + i);
        c = _mm256_sub_ps(a, c); // subtraction instead of addition
        _mm256_store_ps(result.data + i, c);
    }
    startIndex = n - (n % 8);

#elif defined(WITH_NEON)
    float32x4_t a, b, c;
    for (size_t i = 0; i < n - (n % 4); i += 4)
```

```
    {
        a = vld1q_f32(data + i);
        b = vld1q_f32(other.data + i);
        c = vsubq_f32(a, b); // subtraction instead of addition
        vst1q_f32(result.data + i, c);
    }
    startIndex = n - (n % 4);

#endif

    for (size_t i = startIndex; i < n; i++)
    {
        result.data[i] = data[i] - other.data[i]; // subtraction instead of addition
    }

    return result;
}


template <>
DataBlob<float> DataBlob<float>::operator*(const DataBlob &other) const
{
    if (isEmpty() || other.isEmpty())
    {
        std::cerr << "DataBlob is empty" << std::endl;
        return DataBlob();
    }
    if (rows != other.rows || cols != other.cols || channels != other.channels)
    {
        std::cerr << "size is not the same!" << std::endl;
        return DataBlob();
    }
    DataBlob result(cols, rows, channels);
    size_t n = rows * cols * channels;
    size_t startIndex = 0;
#if defined(WITH_AVX2)
    // printf("WITH_AVX2\n");

    __m256 a, b;
    __m256 c = _mm256_setzero_ps();

#pragma omp parallel for
    for (size_t i = 0; i < n - (n % 8); i += 8)
    {
        a = _mm256_load_ps(data + i);
        b = _mm256_load_ps(other + i);
        c = _mm256_mul_ps(a, b);
        _mm256_store_ps(result.data + i, c);
    }
    startIndex = n - (n % 8);
#elif defined(WITH_NEON)
    // printf("WITH_NEON\n");

    float32x4_t a,
        b;
    float32x4_t c = vdupq_n_f32(0);
    startIndex = n - (n % 4);
    for (size_t i = 0; i < n - (n % 4); i += 4)
    {
        a = vld1q_f32(data + i);
        b = vld1q_f32(other.data + i);
        c = vmulq_f32(a, b);
        vst1q_f32(result.data + i, c);
    }

    // printf("WITH_NORMAL\n");

#endif
    for (size_t i = startIndex; i < n; i++)
    {
```

```
        result.data[i] = data[i] * other.data[i];
    }
    return result;
}
```

Part 3 - Result & Verification

I use the previous project testcase

Input format (this input format is inly for inputing by CLI, I also provided only input the matrix by generate it randomly):

The first line contains three integers: the input matrix's **width, height, and channels**.

The next **width * height * channels** numbers represent the input data of the matrix.

After that, the input specifies the number of kernels, denoted by **X**.

On the next line, two integers specify the **kernel's width** and **height.**

For the following **X** lines, input the matrix of each kernel with a size of **kernel width * kernel height * channel**.

(**Note:** The channel of the kernel is following the channel of the input matrix.)

Test case #1:

input

5 5 1
3 3 2 1 0
0 0 1 3 1
3 1 2 2 3
2 0 0 2 2
2 0 0 0 1
1
3 3
0 1 2
2 2 0
0 1 2

output

12 12 17

10 17 19

9 6 14

Test case #2:

input

3 3 3
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
3
3 3

1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3

output

162

Test case #3:

input

7 12 1
174.61136 -171.30317 -13.69410 192.83274 210.87194 -166.49262 29.56846
234.98650 175.06305 42.55915 -22.74195 18.35557 -158.76088 113.59924
-64.31511 -148.61469 -59.16994 137.95201 -123.37056 -191.38510 -73.87436
248.91074 -99.77266 -127.95663 76.24206 164.40580 147.22900 -92.06523
-102.65283 238.20587 -115.31792 172.26555 239.84912 -56.81348 -3.90170
-98.72710 -187.69873 19.88529 231.99691 50.00961 9.98607 -187.31774
33.97372 -73.22712 -250.56467 -236.63036 -241.01910 84.93744 199.23885
191.70495 -19.44642 44.58766 120.05381 -37.92156 -90.75112 78.53444
223.29366 1.21825 46.67164 211.39595 -214.75542 -130.69391 254.55048
-1.16159 -183.71336 -51.57398 244.19721 -243.57829 -90.69351 -191.69700
57.99808 132.60660 181.70882 119.77036 -228.23715 173.72104 174.44084
-162.19071 -118.96106 -210.85550 -182.99310 249.12274 -205.06105 -118.84408
1
5 5
3 2 1 4 32
34 45 223 54 23
234 55 23 5 23 6
234 4 2 34 5 3
4 2 5 4 3

output

-9832.29 -67744.34 3301.55
101013.05 -42976.98 -15187.18
-72802.45 69096.92 72551.24
-53813.21 -36777.17 15788.12
2544.17 20910.34 -22080.71
-36478.02 -73116.86 4032.07
11268.71 14747.92 74160.66
31887.17 35020.78 -11738.94

Test case #4:

input

3 3 1
4 4 4
4 4 4
4 4 4
1
1 1
5

output

20 20 20
20 20 20
20 20 20

Explanation for test case #2

3 3 3 → matrix **width, height and channels**
1 1 1 1 1 1 1 1 1 → 1st channel
1 1 1 1 1 1 1 1 1 → 2nd channel (matrix data with the size of **matrix width*_height_*channels)**
1 1 1 1 1 1 1 1 1 → 3rd channel
3 → number of kernels
3 3 → **kernel width and height**
1 1 1 1 1 1 1 1 1 → 1st channel
1 1 1 1 1 1 1 1 1 → 2nd channel **(first kernel)**
1 1 1 1 1 1 1 1 1 → 3rd channel
2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 **(second kernel)**
2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 **(third kernel)**
3 3 3 3 3 3 3 3 3

in this DataBlob class we can do +,-,/,*,= operation

```
DataBlob<int> A(2, 2, 1);
DataBlob<int> B(2, 2, 1);
A.generateRandomMatrix();
B.generateRandomMatrix();
cout<<"A= \n"<<A;
cout<<"B= \n"<<B;
DataBlob<int> C = A + B;
cout << "A + B= \n" << C;
C = A - B;
cout << "A - B= \n" << C;
C = A * B;
cout << "A * B= \n" << C;
C = A / B;
cout << "A / B= \n" << C;
```

```
nopyesok@DESKTOP-TEL06N
t4$ ./a.out
A=
80 51
36 81
B=
61 29
10 29
A + B=
141 80
46 110
A - B=
19 22
26 52
A * B=
4880 1479
360 2349
A / B=
1 1
3 2
```

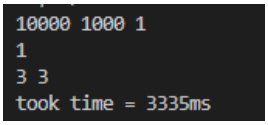
and also we can assign Datablob with different type and do math operation with differen type too.

```
DataBlob<float> A(2, 2, 1);
DataBlob<int> B(2, 2, 1);
A.generateRandomMatrix();
B.generateRandomMatrix();
cout<<"A= \n"<<A;
cout<<"B= \n"<<B;
DataBlob<double> C = A + B;
cout << "A + B= \n" << C;
C = A - B;
cout << "A - B= \n" << C;
C = A * B;
cout << "A * B= \n" << C;
C = A / B;
cout << "A / B= \n" << C;
```

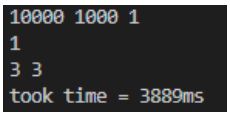
```
nopyesok@DESKTOP-TEL06N0:/mnt/c/
t4$ ./a.out
A=
48.8074 65.7365
31.4708 17.8184
B=
10 76
26 44
A + B=
58.8074 141.737
57.4708 61.8184
A - B=
38.8074 -10.2635
5.47079 -26.1816
A * B=
488.074 4995.97
818.24 784.009
A / B=
4.88074 0.864954
1.21041 0.404963
```

I also doing the comparison between the normal convolutional and the convulutional using optimization using AVX2. To be able to use the optimization I use DataBlob with type float. The size of datablob is 10000 cols 1000 rows and 1 channel with one kernel that has size of 3 cols and 3 cols and 1 channel. The matrix is created using generateRandomMatrix function. And here is the comparison result.

Using AVX2



Without normalization



conclusion is using 1.166 times faster than without normalization.

I've tried to compile using **-fsanitize=address** flag and AddressSanitizer didn't detect any memory errors. So I think all allocated memory has been freed.

Part 4 - Difficulties & Solutions

▼ I want to make the function and operator defined in different file with the header but it keep give an error. After learning about class templat, I solved this error by defined the function and operator in the same file with the header.

▼ It was confusing when I want to assign or do operation between two different data type of DataBlob. I solved this with making one conversion operator that converts a `DataBlob` object to a `DataBlob` object with a different data type.