CS205 Project 5: A Simple CNN Model

Name: Yankai Xu

SID: 12011525

Part 1 - Analysis

Introduction

This project is to implement the forward part of a simple cnn model.

The forward part of cnn includes matrix convolution, ReLU function, max pooling, matrix flatten, full connect layer and softmax function.

The matrix is based on project 4.

Convolution and ReLU

The size of the matrix after convolution is calculated by `1 + (rows + padding * 2 - kernel_size) / stride`.

The convolution part is not much to say, as the implement is 100% based on the definition. I didn't find a way to optimize the access of memory, and simd is not available for the use of template. So, the only way I came out to make it faster is openmp.

The ReLU part is right behind the convolution, which is implemented by std::max.

The convolution supports different data types.

There are some basic checks in this function, and compiles only in debug build.

Max Pooling

This part is still simple with definition based implementation, openmp, custom size and parameter check.

flat

Turn Matrix into std::vector. Maybe it's properer to turn it into another matrix, but I didn't figure out the right mathematical way to do this. The current version is flat the matrix by row, then column, then channel.

full connect

As the result of flat is std::vector, full connect layer is implemented by vector dot product rather than matrix multiplication.

Part 2 - Code

```
// Matrix.hpp
#pragma once

#include "MatrixImpl.hpp"
#include <memory>
#include <iostream>
#include <sstream>
#include <cmath>

template<typename T>
class Matrix {
private:
    std::vector<std::shared_ptr<MatrixImpl<T>>> pImpls;
    size_t rows, cols, channels, offsetX = 0, offsetY = 0;

    template<typename U>
    bool isValueEqual(const Matrix<U> &rhs) const {
        if (channels != rhs.channels) return false;
        if (rows != rhs.rows || cols != rhs.cols) return false;
        for (size_t i = 0; i < channels; i++) {
            for (size_t j = 0; j < pImpls[i]->base.size(); j++) {
                if (!isEqual(pImpls[i]->base[j], rhs.pImpls[i]->base[j])) re
            }
        }
        return true;
    }

    template<typename U>
    void copyImpl(const Matrix<U> &rhs) {
        for (size_t i = 0; i < channels; i++) {
            pImpls[i] = std::make_shared<MatrixImpl<T>>(*rhs.pImpls[i],
                                                        Range{offsetY, offse
        }
    }

    // static methods for operator >>
    static void pushToImplBase(std::shared_ptr<MatrixImpl<T>> &impl, T &val
        impl->base.push_back(val);
    }

    static void setImplSize(std::shared_ptr<MatrixImpl<T>> &impl, size_t row
        impl->rows = rows;
        impl->cols = cols;
    }

public:

    Matrix(const Matrix &rhs) : pImpls(rhs.channels), rows{rhs.rows},

                                cols{rhs.cols}, channels{rhs.channels}, offs
        copyImpl(rhs);
```

```
    }

    template<typename U>
    explicit Matrix(const Matrix<U> &rhs) : pImpls(rhs.channels), rows{rhs.r
                                            cols{rhs.cols}, channels{rhs.cha
                                            offsetY{rhs.offsetY} {
        copyImpl(rhs);
    }

    Matrix(Matrix<T> &&rhs) noexcept = default;

    explicit Matrix(const size_t &rows = 0, const size_t &cols = 0, const s
            : pImpls(channels), rows{rows}, cols{cols}, channels{channels} 
        for (auto &v: pImpls) {
            v = std::make_shared<MatrixImpl<T>>(rows, cols, val);
        }
    }

    Matrix(const Matrix &other, const Range &row, const Range &col)
            : pImpls(other.pImpls), rows(row.size()), cols(col.size()), char
              offsetY(col.start) {
#ifndef NDEBUG
        if (rows + offsetX > pImpls[0]->getNumberOfRows() || cols + offsetY
            throw std::invalid_argument("Invalid Range");
#endif
    }

    std::vector<T> flat() const {
        std::vector<T> res;
        for (auto &v: pImpls) {
            res.insert(res.end(), v->base.begin(), v->base.end());
        }
        return res;
    }

    Matrix<T> &operator=(const Matrix<T> &rhs) {
        Matrix<T> temp(rhs);
        swap(temp);
        return *this;
    }

    Matrix<T> &operator=(Matrix<T> &&rhs) noexcept = default;

    std::vector<std::reference_wrapper<T>> operator()(const size_t &row, co
#ifndef NDEBUG
        if (row >= rows || col >= cols) throw std::invalid_argument("invalid
#endif
        std::vector<std::reference_wrapper<T>> res;
        for (std::shared_ptr<MatrixImpl<T>> &v: pImpls) {
            res.push_back(v->operator()(offsetX + row, offsetY + col));
        }
        return res;
    }

    std::vector<T> operator()(const size_t &row, const size_t &col) const {
```

```
#ifndef NDEBUG
        if (row >= rows || col >= cols) throw std::invalid_argument("invali
#endif
        std::vector<T> res(channels);
        for (size_t i = 0; i < channels; i++) {
            res[i] = pImpls[i]->operator()(offsetX + row, offsetY + col);
        }
        return res;
    }

    T &operator()(const size_t row, const size_t col, const size_t channel)
#ifndef NDEBUG
        if (row >= rows || col >= cols) throw std::invalid_argument("invali
#endif
        return (*pImpls[channel])(row + offsetX, col + offsetY);
    }

    const T &operator()(const size_t row, const size_t col, const size_t cha
#ifndef NDEBUG
        if (row >= rows || col >= cols) throw std::invalid_argument("invali
#endif
        return (*pImpls[channel])(row + offsetX, col + offsetY);
    }

    Matrix operator()(const Range &row, const Range &col) const {
        return Matrix(*this, row, col);
    }

    bool operator==(const Matrix &rhs) const {
        if (this == &rhs) return true;
        if (channels != rhs.channels) return false;
        bool pImplFlag = true;
        for (size_t i = 0; i < channels; i++) {
            if (pImpls[i] != rhs.pImpls[i]) {
                pImplFlag = false;
                break;
            }
        }
        if (pImplFlag && rows == rhs.rows && cols == rhs.cols && offsetX ==
            return true;
        return isValueEqual(rhs);
    }

    template<typename U>
    bool operator==(const Matrix<U> &rhs) const {
        return isValueEqual(rhs);
    }

    template<typename U>
    auto operator+(const Matrix<U> &rhs) const {
#ifndef NDEBUG
        if (rows != rhs.rows || cols != rhs.cols) throw std::invalid_argumen
        if (channels != rhs.channels)
            throw std::invalid_argument("A.channel != B.channel");
#endif
```

```
        typedef decltype(T() + U()) return_type;
        Matrix<return_type> res(rows, cols, channels);
        for (size_t i = 0; i < channels; i++) {
#pragma omp parallel for
            for (size_t j = 0; j < pImpls[i]->base.size(); j++) {
                res.pImpls[i]->base[j] = pImpls[i]->base[j] + rhs.pImpls[i]-
            }
        }

        return res;
    }

    template<typename U>
    Matrix<T> &operator+=(const Matrix<U> &rhs) {
#ifndef NDEBUG
        if (rows != rhs.rows || cols != rhs.cols) throw std::invalid_argumen
        if (channels != rhs.channels)
            throw std::invalid_argument("A.channel != B.channel");
#endif
        for (size_t i = 0; i < channels; i++) {
#pragma omp parallel for
            for (size_t j = 0; j < pImpls[i]->base.size(); j++) {
                pImpls[i]->base[j] += rhs.pImpls[i]->base[j];
            }
        }
        return *this;
    }

    Matrix<T> &operator+() {
        return *this;
    }

    template<typename U>
    auto operator-(const Matrix<U> &rhs) const {
#ifndef NDEBUG
        if (rows != rhs.rows || cols != rhs.cols) throw std::invalid_argumen
        if (channels != rhs.channels)
            throw std::invalid_argument("A.channel != B.channel");
#endif
        typedef decltype(T() - U()) return_type;
        Matrix<return_type> res(rows, cols, channels);
        for (size_t i = 0; i < channels; i++) {
#pragma omp parallel for
            for (size_t j = 0; j < pImpls[i]->base.size(); j++) {
                res.pImpls[i]->base[j] = pImpls[i]->base[j] - rhs.pImpls[i]-
            }
        }
        return res;
    }

    template<typename U>
    Matrix<T> &operator-=(const Matrix<U> &rhs) {
#ifndef NDEBUG
        if (rows != rhs.rows || cols != rhs.cols) throw std::invalid_argumen
        if (channels != rhs.channels)
```

```
            throw std::invalid_argument("A.channel != B.channel");
#endif
        for (size_t i = 0; i < channels; i++) {
#pragma omp parallel for
            for (size_t j = 0; j < pImpls[i]->base.size(); j++) {
                pImpls[i]->base[j] -= rhs.pImpls[i]->base[j];
            }
        }
        return *this;
    }

    Matrix<T> operator-() const {
        Matrix<T> res(rows, cols, channels);
        for (size_t i = 0; i < channels; i++) {
#pragma omp parallel for
            for (size_t j = 0; j < pImpls[i]->base.size(); j++) {
                res.pImpls[i]->base[j] = -pImpls[i]->base[j];
            }
        }
        return res;
    }

    template<typename U>
    Matrix<T> operator*(const U &rhs) const {
        Matrix<T> res(rows, cols, channels);
        for (size_t i = 0; i < channels; i++) {
#pragma omp parallel for
            for (size_t j = 0; j < pImpls[i]->base.size(); j++) {
                res.pImpls[i]->base[j] = pImpls[i]->base[j] * rhs;
            }
        }
        return res;
    }

    template<typename U>
    auto operator*(const Matrix<U> &rhs) const {
#ifndef NDEBUG
        if (rows != rhs.cols) throw std::invalid_argument("A.rows != B.cols'
        if (channels != rhs.channels)
            throw std::invalid_argument("A.channel != B.channel");
#endif
        typedef decltype(T() + U()) return_type;
        Matrix<return_type> res(rows, rhs.cols, channels);
        for (size_t c = 0; c < channels; c++) {
#pragma omp parallel for
            for (size_t i = 0; i < rows; i++) {
                for (size_t k = 0; k < cols; k++) {
                    const T &tmp = (*this)(i, k, c);
                    for (size_t j = 0; j < rhs.cols; j++) {
                        res(i, j, c) += tmp * rhs(k, j, c);
                    }
                }
            }
        }
        return res;
```

```
    }

    template<typename U>
    Matrix<T> &operator*=(const U &rhs) {
        for (size_t i = 0; i < channels; i++) {
#pragma omp parallel for
            for (size_t j = 0; j < pImpls[i]->base.size(); j++) {
                pImpls[i]->base[j] *= rhs;
            }
        }
        return *this;
    }

    void swap(Matrix<T> &other) {
        using std::swap;
        pImpls.swap(other.pImpls);
        swap(rows, other.rows);
        swap(cols, other.cols);
        swap(offsetY, other.offsetY);
        swap(offsetX, other.offsetX);
        swap(channels, other.channels);
    }

    friend std::istream &operator>>(std::istream &in, Matrix &mat) {
        for (size_t i = 0; i < mat.channels; i++) {
            mat.pImpls[i] = std::make_shared<MatrixImpl<T>>();
        }
#ifndef NDEBUG
        bool first = true;
#endif
        size_t rowCount = 0;
        for (std::string line; std::getline(in, line);) {
            std::istringstream iss(line);
            size_t channel = 0;
            size_t colCount = 0;
            rowCount++;
            for (T num; iss >> num;) {
                pushToImplBase(mat.pImpls[channel++], num);
                channel %= mat.channels;
                colCount++;
            }
#ifndef NDEBUG
            if (channel != 0) {
                throw std::invalid_argument("wrong channel numbers.");
            }
            if (!first && mat.cols != colCount / mat.channels) {
                throw std::invalid_argument("wrong input.");
            }
            first = false;
#endif
            for (size_t i = 0; i < mat.channels; i++) {
                setImplSize(mat.pImpls[i], rowCount, colCount / mat.channel
            }
            mat.cols = colCount / mat.channels;
        }
```

```
        mat.rows = rowCount;
        mat.offsetY = 0;
        mat.offsetX = 0;
        return in;
    }

    size_t getNumberOfRows() const {
        return rows;
    }

    size_t getNumberOfCols() const {
        return cols;
    }

    size_t getNumberOfChannels() const {
        return channels;
    }

    friend std::ostream &operator<<(std::ostream &out, const Matrix<T> &mat
        for (size_t i = 0; i < mat.rows; i++) {
            for (size_t j = 0; j < mat.cols; j++) {
                for (size_t k = 0; k < mat.channels; k++) {
                    out << mat(i, j, k) << " ";
                }
            }
            out << std::endl;
        }
        return out;
    }

    template<typename> friend
    class Matrix;

    template<typename> friend
    class MatrixImpl;
};

template<typename T>
void swap(Matrix<T> &a, Matrix<T> &b) {
    a.swap(b);
}
```

```
// MatrixImpl.hpp
#pragma once

#include <vector>
#include <stdexcept>
#include <type_traits>

template<typename L, typename R>
std::enable_if_t<std::disjunction_v<std::is_floating_point<L>, std::is_floa
isEqual(const L &lhs, const R &rhs) {
```

```
        return std::abs(lhs - rhs) <=
               (((std::abs(rhs) < std::abs(lhs)) ? std::abs(rhs) : std::abs(lhs
    }

    template<typename L, typename R>
    std::enable_if_t<!std::disjunction_v<std::is_floating_point<L>, std::is_floa
    isEqual(const L &lhs, const R &rhs) {
        return lhs == rhs;
    }

    struct Range {

        size_t start, end;

        Range(size_t start, size_t end) : start(start), end(end) {
    #ifndef NDEBUG
            if (start > end) throw std::invalid_argument("start > end");
    #endif
        }

        [[nodiscard]] size_t size() const {
            return end - start;
        }
    };

    template<typename T>
    class MatrixImpl {
    private:
        size_t rows, cols;
        std::vector<T> base;

    public:

        template<typename U>
        explicit MatrixImpl(const MatrixImpl<U> &rhs, const Range row, const Ra
                : rows{row.size()}, cols{col.size()}, base(row.size() * col.size
            for (size_t i = 0; i < base.size(); i++) {
                base[i] = static_cast<T>(rhs.base[i]);
            }
        }

        explicit MatrixImpl(const size_t &row = 0, const size_t &col = 0, const
                : rows{row}, cols{col}, base(row * col, val) {}

         size_t getNumberOfRows() const {
            return rows;
         }

        size_t getNumberOfCols() const {
            return cols;
        }

        T &operator()(const size_t row, const size_t col) {
            return base[row * cols + col];
        }
```

```
    T operator()(const size_t row, const size_t col) const {
        return base[row * cols + col];
    }

    template<typename> friend
    class MatrixImpl;

    template<typename> friend
    class Matrix;
};
```

```
// cnn.hpp
#pragma once

#include "Matrix.hpp"

template<typename T, typename U, typename V>
Matrix<T>
convBnReLU(const Matrix<T> &target, const std::vector<U> &kernels, const st
           const size_t &outChannel, const size_t &stride = 1, const size_t
#ifndef NDEBUG
    if (outChannel * target.getNumberOfChannels() * kernel_size * kernel_si
    if (bias.size() != outChannel) throw std::invalid_argument("Bad bias si
        if (stride < 1) throw std::invalid_argument("Bad stride");
#endif
    Matrix<T> res(1 + (target.getNumberOfRows() + padding * 2 - kernel_size
                  1 + (target.getNumberOfCols() + padding * 2 - kernel_size
                  outChannel);

#pragma omp parallel for
    for (size_t out = 0; out < outChannel; out++) {
        for (size_t in = 0; in < target.getNumberOfChannels(); in++) {
            std::vector<U> kernel(kernel_size * kernel_size);
            for (size_t k = 0; k < kernel_size * kernel_size; k++) {
                kernel[k] = kernels[out * target.getNumberOfChannels() * ke
                                    in * kernel_size * kernel_size +
                                    k];

            }
            for (size_t i = 0; i < 1 + target.getNumberOfCols() - kernel_si
                for (size_t j = 0; j < 1 + target.getNumberOfRows() - kerne
                    for (size_t k = 0; k < kernel_size; k++) {
                        for (size_t l = 0; l < kernel_size; l++) {
                            int64_t tmpI = i + k, tmpJ = j + l;
                            tmpI -= padding;
                            tmpJ -= padding;
                            T num = (tmpI < 0 || tmpI >= target.getNumberOf
                                     tmpJ >= target.getNumberOfCols()) ? 0
                            res(i / stride, j / stride, out) += num * kerne
                        }
                    }
```

```
                }
            }
        }
        for (size_t i = 0; i < res.getNumberOfRows(); i++) {
            for (size_t j = 0; j < res.getNumberOfCols(); j++) {
                res(i, j, out) = std::max(T(), res(i, j, out) + bias[out]);
            }
        }
    }
    return res;
}

template<typename T>
Matrix<T> maxPool2D(const Matrix<T> &target, const size_t &size) {
#ifndef NDEBUG
    if (target.getNumberOfRows() % size != 0 || target.getNumberOfCols() % s
#endif
    Matrix<T> res(target.getNumberOfRows() / size, target.getNumberOfCols()
#pragma omp parallel for
    for (size_t c = 0; c < target.getNumberOfChannels(); c++) {
        for (size_t i = 0; i < target.getNumberOfRows(); i += size) {
            for (size_t j = 0; j < target.getNumberOfCols(); j += size) {
                for (size_t m = 0; m < size; m++) {
                    for (size_t n = 0; n < size; n++) {
                        res(i / size, j / size, c) = std::max(res(i / size,
                    }
                }
            }
        }
    }
    return res;
}

template<typename T, typename U, typename V>
std::vector<T> fullConnect(const std::vector<T> &flattenedMat, const std::ve
#ifndef NDEBUG
    if (flattenedMat.size() != weight.size() / bias.size())
        throw std::invalid_argument(
                "bad connect args: " + std::to_string(flattenedMat.size()) -
#endif
    size_t out = bias.size();
    std::vector<T> res(out);
#pragma omp parallel for
    for (int o = 0; o < out; o++) {
        for (int i = 0; i < flattenedMat.size(); i++) {
            float w = weight[o * flattenedMat.size() + i];
            res[o] += w * flattenedMat[i];
        }
        res[o] += bias[o];
    }
    return res;
}

template<typename T>
std::vector<T> softMax(std::vector<T> vec) {
```

```
    // vec has only 2 element, therefore, copy is acceptable.
    std::vector<T> res(vec.size());
    T sum = 0;
    for (size_t i = 0; i < vec.size(); i++) {
        vec[i] = std::exp(vec[i]);
        sum += vec[i];
    }
    for (size_t i = 0; i < vec.size(); i++) {
        res[i] = vec[i] / sum;
    }
    return res;
}
```

```
// face_binary_cls.hpp
// face_binary_cls.cpp is the version in github repo.

#pragma once

extern float conv0_weight[16*3*3*3];
extern float conv0_bias[16];

extern float conv1_weight[32*16*3*3];
extern float conv1_bias[32];

extern float conv2_weight[32*32*3*3];
extern float conv2_bias[32];

extern float fc0_weight[2*2048];
extern float fc0_bias[2];
```

```
// main.cpp
#include <iostream>
#include "Matrix.hpp"
#include "opencv2/opencv.hpp"
#include "face_binary_cls.hpp"
#include "cnn.hpp"

#pragma GCC optimize(3, "Ofast", "inline")

int main() {

    cv::Mat cv_in = cv::imread("../images/face.jpg", cv::ImreadModes::IMREAI
    Matrix<float> in(128, 128, 3);
    for (int i = 0; i < 128; i++) {
        for (int j = 0; j < 128; j++) {
            cv::Vec3b p = cv_in.at<cv::Vec3b>(i, j);
            for (int c = 0; c < 3; c++) {
                in(i, j, c) = static_cast<float>(p[c]) / 255.0f;
            }
        }
```

```
    }

    std::vector<float> weight{conv0_weight, std::end(conv0_weight)};
    std::vector<float> bias{conv0_bias, std::end(conv0_bias)};

    auto out = convBnReLU(in, weight, bias, 3, 16, 2, 1);
    out = maxPool2D(out, 2);

    weight = {conv1_weight, std::end(conv1_weight)};
    bias = {conv1_bias, std::end(conv1_bias)};
    out = convBnReLU(out, weight, bias, 3, 32, 1, 0);
    out = maxPool2D(out, 2);

    weight = {conv2_weight, std::end(conv2_weight)};
    bias = {conv2_bias, std::end(conv2_bias)};
    out = convBnReLU(out, weight, bias, 3, 32, 2, 1);

    auto flattened = out.flat();

    weight = {fc0_weight, std::end(fc0_weight)};

    bias = {fc0_bias, std::end(fc0_bias)};
    auto res = fullConnect(flattened, weight, bias);
    res = softMax(res);

    std::cout << res[0] << " " << res[1] << std::endl;

    return 0;
}
```

Part 3 - Result & Verification

Test case #1:

Test two pictures in github repo.

face.jpg:

```
Conv 1: 25.85ms
pool 1: 0.09901ms
Conv 2: 3.06999ms
pool 2: 2.67851ms
Conv 3: 0.299196ms
flat: 0.004082ms
fc: 0.602153ms
0.00708574 0.992914
total_cal_time: 32.6029ms
```

bg.jpg:

```
Conv 1: 1.44713ms
pool 1: 0.071242ms
Conv 2: 2.00531ms
pool 2: 3.80184ms
Conv 3: 9.411ms
flat: 0.002374ms
fc: 0.03836ms
0.999996 3.7508e-06
total_cal_time: 16.7773ms
```

The result is the same as demo provided.

The time is not accurate as the picture is small.

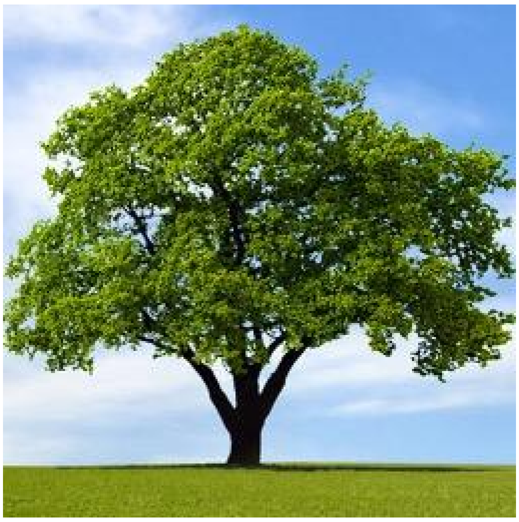
Test case #2:

Test 2 images found on the internet. images are scaled into 128*128 by cv::resize.

image 1:

```
Conv 1: 40.8877ms
pool 1: 2.43414ms
Conv 2: 2.13033ms
pool 2: 0.023901ms
Conv 3: 0.262511ms
flat: 0.002732ms
fc: 0.003791ms
0.0215678 0.978432
total_cal_time: 45.7451ms
```

image 2:

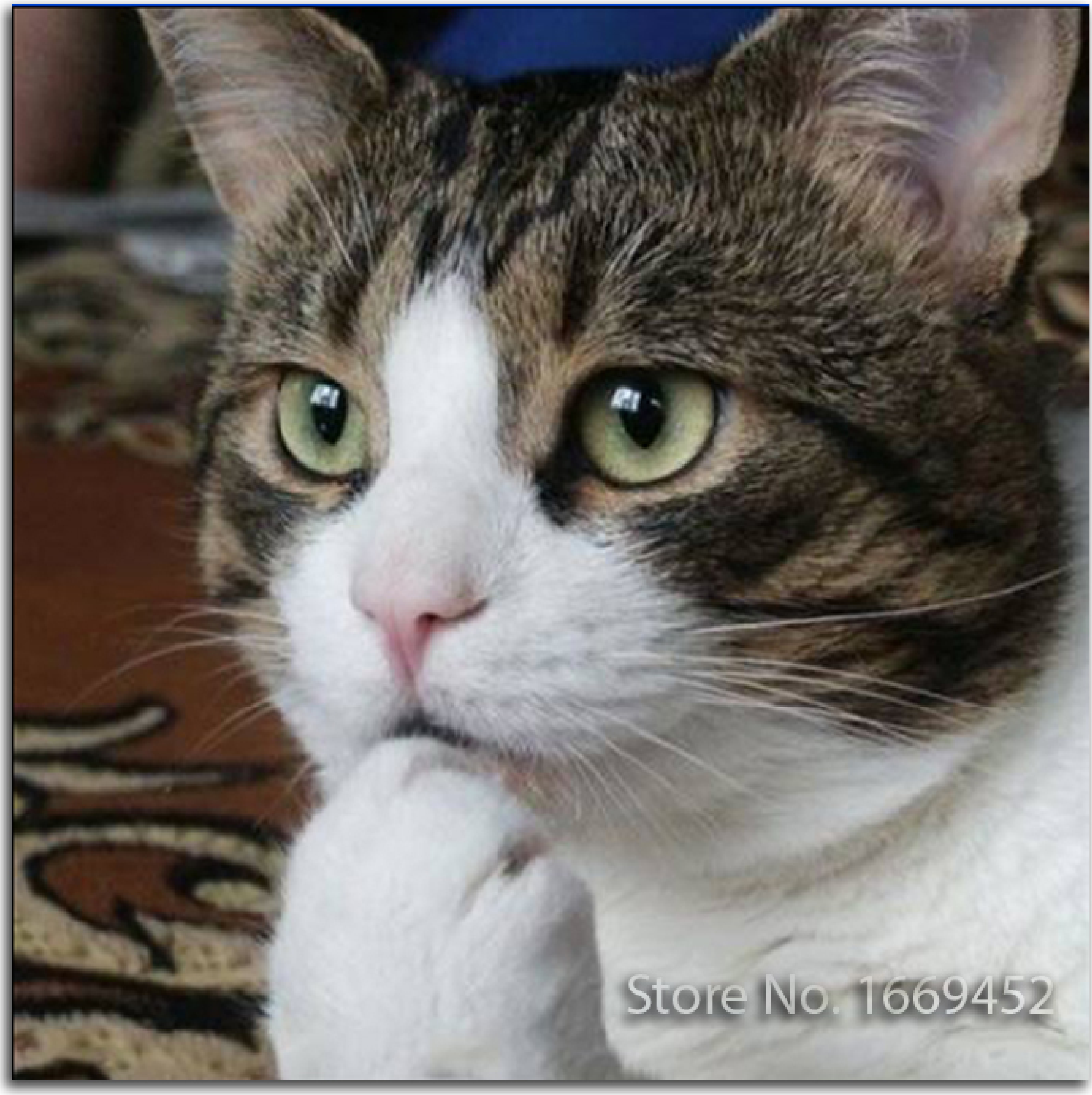


```
Conv 1: 32.1872ms
pool 1: 17.9811ms
Conv 2: 15.0717ms
pool 2: 0.148964ms
Conv 3: 0.305984ms
flat: 0.003176ms
fc: 0.343552ms
0.999998 2.25187e-06
total_cal_time: 66.0417ms
```

Test case #3:

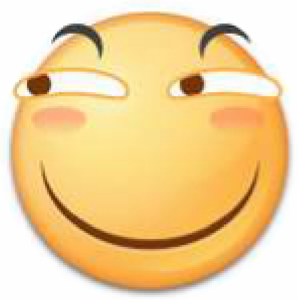
Test pictures which look like human face.

image 3:




```
Conv 1: 22.8513ms
pool 1: 13.9516ms
Conv 2: 15.8444ms
pool 2: 0.058001ms
Conv 3: 0.298ms
flat: 0.004131ms
fc: 0.004444ms
0.422723 0.577277
total_cal_time: 53.0119ms
```

image 4:



```
Conv 1: 24.1399ms
pool 1: 10.0605ms
Conv 2: 2.10686ms
pool 2: 0.023915ms
Conv 3: 0.260045ms
flat: 0.002745ms
fc: 0.003858ms
0.122099 0.877901
total_cal_time: 36.5979ms
```

image 5:


```
Conv 1: 1.16787ms
pool 1: 0.06786ms
Conv 2: 1.94794ms
pool 2: 0.022763ms
```

```
Conv 3: 0.312046ms
flat: 0.003555ms
fc: 0.728692ms
0.0292498 0.97075
total_cal_time: 4.25073ms
```

It seems that this model is not predicting whether the input image is a person (upper body only) but detecting whether there's an eye in the image...

Test case #4:

Tsest the performance on ARM server.

```
Conv 1: 3.74597ms
pool 1: 0.181752ms
Conv 2: 8.41709ms
pool 2: 0.081911ms
Conv 3: 1.16886ms
flat: 0.00663ms
fc: 0.00696ms
0.999996 3.7508e-06
total_cal_time: 13.6092ms
```

I tried servel times and the time remains at 13ms. That's quite different from my computer, which the time result varies from 4ms to 100ms. I guess that's because the server is a lot more stable than my computer. The char difference between my computer and ARM server doesn't matter because opencv has its own uchar type, which is an alias of unsigned char.

Part 4 - Difficulties & Solutions

CNN

The most difficult part for me is understand cnn and figure out how matries size varies. As I started the project before the lecture, I took some time searched on the internet and did some calculations. In the end, I took a brief understanding of how cnn works, which is enough for my project. But I still have lots of questions since I hadn't attend some math classes which is necessary to machine learning. So I'll consider studing them in the next term.

debug

The algorithm I use is not complicated at all, but I still meet some bugs there. As I mentioned before, when I was writing my code, I still have some questions about cnn. So, I wasn't that confident to tell that my algorithm is right. In the end, I installed pytorch and modified the demo to get the intermediate result. In the end, I found I made a mistake in two variable names. I fixed them and get the right answer.