CS205 · C/C++ Programming

何泽安 (He Zean)
12011323

Project 5: CNN Face Detector

Dec. 19, 2021

Contents

1 需求分析

在充分学习了面向对象编程的知识后，本次 project 设计了对模型的设计者和使用者都有友好的使用体验的类，其具有健全的异常处理体系以及灵活的参数选择（如激活函数不限定为 Relu），在此基础上修改模型结构和拓展本框架都极为方便。

本 project 也讨论了 CNN 在访存优化及并行计算方面的加速方案，另外，对于具体的模型（如提供的 SimpleCNNbyCPP），在损失一定代码通用性的条件下还可进一步的提速。

此外，提供了方便测试本模型效果的网站 *http://172.18.24.39:8501* [1]，能直观展示结果并与官方 Python 实现进行对比。

2 代码实现

本次 CNN 代码分为两部分：第一部分用于体现面向对象程序设计思维，鲁棒性强，能方便的适配各种结构的神经网络；而第二部分专门为 SimpleCNNbyCPP 的网络结构实现了性能提升，网络参数固定并直接写入代码[2]

2.1 整体架构

核心代码架构（精简起见，仅展示头文件）

[1]需在校园网环境下访问。

[2]要实现通用的 CNN 需要考虑多种情况，且代码实现耗时多于特例化版本，考虑到时间关系，仅以此为例展示优化方案。

利用（纯）虚函数和类的继承，我们在 CnnLayer 类中定义了神经网络层需要支持的基本操作（类似于 Java 中实现某个 Interface），并体现了各种网络层的 *is-a* 关系；在类的实现中重写虚函数来完成具体操作。

各层的参数（若有）由各层各自保管，而 modules/Sequential 封装了网络的细节（保存了 CNN 中的 layers），提供了模型合法性检查和开箱即用的特性。

与 project 4 类似，我们使用 cnnexception 完善并归类了神经网络可能遇到的几种异常；imgdata 通过在拷贝构造器和赋值运算符中管理引用计数器，大大减轻了网络层中内层管理的负担（由 SmartPtr 修改而来，但本次在了解 channel 的作用后将其存为一维数组），也能有效避免忘记释放内存（通过了 project 4 中的 debug_new 的内存泄露检测）；extras 提供了计时器等辅助功能。

2.2 Img & CnnExceptions

```cpp
class Img {
    Data *m_data;  /* { size_t width, height, channels;  int ref_cnt;   float *data; } */
public:
    /* constructors, with try-catch-throw to avoid partial construction */
    Img(const Img &rhs) { this->m_data = rhs.m_data;  ++this->m_data->ref_cnt; }
    ~Img() { try { if (--m_data->ref_cnt == 0) { delete m_data; } } catch (...) {} }
    Img &operator=(const Img &rhs) {
        if (this == &rhs || this->m_data == rhs.m_data) return *this;
        if (--m_data->ref_cnt == 0) delete m_data;
        m_data = rhs.m_data;    ++m_data->ref_cnt;
        return *this;
    }
};
```

Project 4 中通过使用 initializer_list，实现了不定长传参，但这要求参数类型一致。而本次使用了展开

参数包的方式使传入信息更加灵活。

2.3 CnnLayer

```cpp
class CnnLayer {
public:
    virtual ~CnnLayer() = default;
    virtual Img forward(const Img &img) const = 0;
    virtual void summary(ostream &os, const vector<size_t> &input) const = 0;
    virtual bool checkValid(vector<size_t> &last_opt) const = 0;
    virtual vector<size_t> optSize(vector<size_t> &in_size) const = 0;
};
```

下面将以此为基类将用到的多种 layers 写为派生类，不同种类的 layers 间的行为有较大差异，但作为 layer 均需实现基类中规定的纯虚函数；显然，这里基类的析构函数无需有任何实际行为，但我们需要将其指出是虚函数以便在 delete CnnLayer* 时能调用派生类的析构函数，避免内存泄露。

① **forward** 即 layer 向后计算。这里的 Img 与 project 4 中实现的 SmartPtr 类似，唯一的成员变量为一个指向（保管了引用计数器的）struct Data 的指针，并在拷贝构造、赋值操作和析构函数中自动管理内存。在此处，我们在函数内创建保存新结果的 Img 对象 <img_out>，此时其数据区引用计数为 1，通过拷贝构造传出时，数据区引用首先短暂的变为 2，但函数内的栈内对象随即被析构，引用数重新变为 1。而原 Img 的引用数在函数内外均为 1，并在赋值操作时析构原有数据区。

② **summary, checkValid, & optSize** 主要供 Module 调用。optSize 需要输入上一层输出尺寸并计算输出尺寸，checkValid 检查该层是否能合法放入模型中（如上一层输出通道数为 16，则要求 Conv3D 的卷积核深度一致）。

2.4 ImgInput

作为模型的第一层，负责接收各种图片并将其整形为预训练网络所期望的输入大小及通道[3]。并提供多种“resize”方式[4]，仅在实际输入大小与输出大小不一致时使用。此外，考虑到 OpenCV 获取数据的较高效手段是通过 ptr 一次获取一行，且获得的 NHWC 与后面所需的 NCHW 存储顺序差异较大，故仅尝试使用 OpenMP 和循环展开的基本手段提速。

对于 SimpleCnnByCpp，这里直接使用 BGR 的顺序读入可以获取与 demo.py 一致的结果，能验证其余部分代码的正确性；而转为 RGB 读入（可以通过 ① 使用 cv::COLOR_BGR2RGB ② 调整 79 行 for 循环写入顺序实现，这里选择②）则可获得更高的准确率。

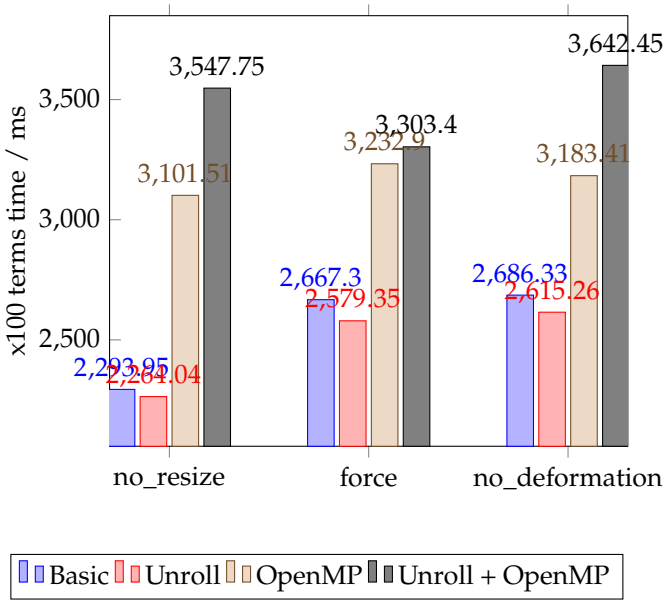
[3]由于图片的伸缩（resize）和通道变换（cvtColor）涉及较多插值等数学内容（没学过啊），这里直接调用 OpenCV。
[4]时间关系，仅实现前两种，即强制变形和无伸缩变形。

```cpp
class ImgInput : public CnnLayer {
public:
    enum ImgResizeType { force, no_deformation, center, left_upper, };
private:
    ImgResizeType resizeOpt;  size_t opt_width;  size_t opt_height;  size_t opt_channels;

public:
    ImgInput(size_t w, size_t h, size_t c, ImgResizeType res = force)
            : resizeOpt(res), opt_width(w), opt_height(h), opt_channels(c) {
        if (w <= 0 || h <= 0 || c <= 0) throw InvalidArgs("(Reshaped) size must > 0", w, h, c);
    }
    void summary(ostream &os, const vector<size_t> &input) const override { /* omitted */ }

    // denoting it is the first layer (no input from "previous layers")
    bool checkValid(vector<size_t> &last_opt) const override { return last_opt.empty(); }
    vector<size_t> optSize(vector<size_t> &in_size) const override { return vector<size_t>{
        opt_width, opt_height, opt_channels}; }

    /**
     * @param cvt -1 -> auto detect: if <in_channels> == <out_channels>, do nothing
     *                               else if <out_channels> == 1, read the file as <gray scale>
     *                               else @throws Fatal(Convert rule undefined)
     *            0 ~ 143 -> force cv::cvtColor, if in/out #channels mismatch @throws Fatal(...)
     *            otherwise -> @throws InvalidArgs(Convert rule not found)
     */
    Img prepare(const char *file, int cvt = -1) const {
        if (cvt < -1 || cvt > 143) throw InvalidArgs("Convert rule not found", cvt);
        cv::Mat image;
        if (cvt == -1) {
            /* if (opt_channels == 1) cv::IMREAD_GRAYSCALE else cv::IMREAD_UNCHANGED */
        } else {
            cv::Mat tmp = cv::imread(file, cv::IMREAD_UNCHANGED);
            if (tmp.data == nullptr) throw Fatal("File not found");
            try {
                cv::cvtColor(tmp, image, cvt);
            } catch (const cv::Exception &e) {
                throw Fatal("Convert failed: convert rule doesn't match #channel", e.what());
            }
            if (image.channels() != opt_channels)
                throw InvalidArgs("Wrong convert rule: output channel mismatch",
                                  image.channels(), opt_channels);
        }  // input channel is well-adjusted now
        if ((resizeOpt == force || resizeOpt == no_deformation) &&
            (static_cast<int>(opt_width) <= 0 || static_cast<int>(opt_height) <= 0))
            throw Fatal("Scale size too large: OpenCV only supports size in range (0,
                2147483647]");  // cv::resize
        Img blob_img(opt_width, opt_height, opt_channels);
        resizer(image, blob_img.ptr());
        return blob_img;
    }
    Img forward(const Img &img) const override { cerr << "Avoid call <forward>";  return img; }
```

```
50
51  private:
52      void resizer(const cv::Mat &ori, float *adj) const {
53          cv::Mat res;
54          if (ori.cols != opt_width || ori.rows != opt_height) {
55              switch (resizeOpt) {
56                  case no_deformation: {
57                      int rc = opt_height / opt_width, inr, inc;
58                      if (ori.rows * rc <= ori.cols) { inr = ori.rows; inc = ori.rows * rc; }
59                      else { inr = ori.cols / rc; inc = ori.cols; }
60                      cv::Mat black(inc, inr, CV_8UC3, cv::Scalar(0, 0, 0));
61                      cv::Mat imageROI;
62                      imageROI = black(cv::Rect((inc - ori.cols) / 2, (inr - ori.rows) / 2,
63                                                ori.cols, ori.rows));
64                      ori.copyTo(imageROI);
65                      cv::resize(black, res, cv::Size(static_cast<int>(opt_width),
66                                                      static_cast<int>(opt_height)));
67                      break;
68                  } /* other cases are omitted */
69              }
70          } else { res = ori; /* using opencv ROI, fast enough */ }
71          size_t opt_ch_step = opt_width * opt_height;
72  // #pragma omp parallel, #pragma omp single [after comparison, we choose not to use omp]
73                  for (size_t r = 0; r < opt_height; ++r) {
74                      uchar *p_rdata = res.ptr<uchar>(r);
75                      for (size_t c = 0; c < opt_width; ++c) {
76  #pragma unroll 3
77                          for (size_t ch = 0; ch < opt_channels; ++ch) {
78                              adj[ch * opt_ch_step + r * opt_width + c] =
79                                      static_cast<float>(p_rdata[c * opt_channels + ch]) / 255.f;
80  };  /* }}}s are omitted */
```

本着“局部最优导致全局最优”的贪心思想，我们马上验证以上提到的加速方案效果：

Method 为降低误差，令 ImgInput 准备 1024 * 1024 的三通道图片，预热 10 次，测量 100 次循环耗时。



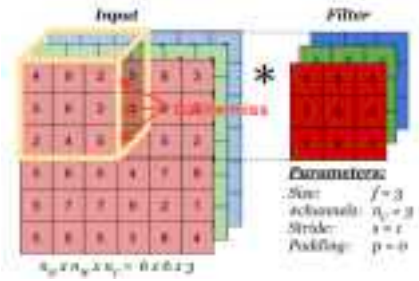
Analysis 由图可见，无需 resize 的耗时自然最少，而补黑边的无拉伸变换耗时最多。使用 OpenMP 的耗时均大幅超过普通情况，这是由于即使是处理长度为 1024 的列这种子任务也“较简单”，管理线程的开销与并行写入的节省相比入不敷出，128 * 128 的图像读取更是如此，IO 性能瓶颈也是导致此现象的一种可能；但循环展开（通道）确实起到了微弱的提速效果。综上，选择仅使用循环展开提速。

2.5 Conv3d

Update on Dec. 16 在定义上，本层正确的叫法似乎是 Conv2d，但仅是名字错误不影响功能，懒得改了...

"Premature optimization is the root of all evil." ——Donald Knuth

首先我们实现了 brute force 版的卷积操作（主要展示其丰富的参数选择与支持，在后续章节讨论优化方案时，为节省精力，不对完整的 CNN 进行优化），此时的卷积依然是多个内存不连续的向量点乘，显然其性能十分不理想：对于高度为 kh 的卷积核，输出的每个卷积结果都会产生 kh 次 **cache miss**；此外，我们知道现代处理器一般采用流水线技术同时执行多条指令的不同阶段，即与将内存数据载入缓存类似，在执行指令的同时将下一条指令载入指令缓存部件，这一设计在理想状态下能将效率加倍[1]。但当遇到条件转移指令时（如 30 行处判断 padding 代码），CPU 猜测性的将其下一条指令也取出并载入缓存，如果转移发生，则需要重新取指令——brute force 版本的指令**流水线效率也极低**。




```
enum ConvPadding { same, full, valid, };
enum ConvActivation { Relu, Sigmoid, Tanh, Linear, };

class Conv3d : public CnnLayer {
```

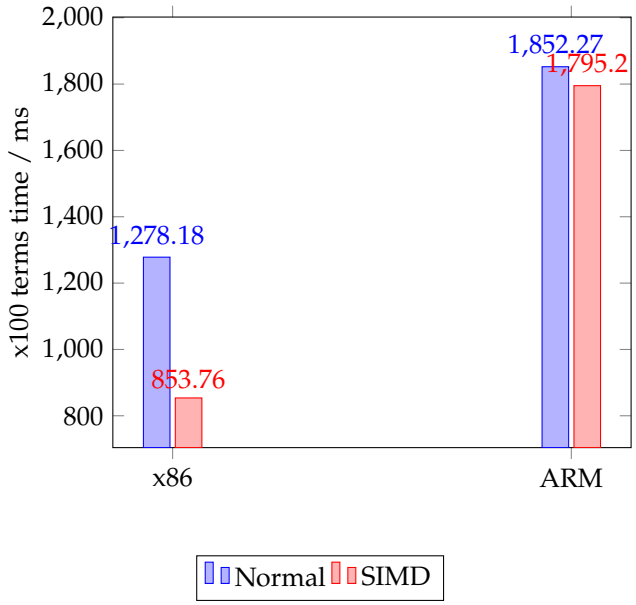
```
    size_t filters;
    size_t kernel_width, kernel_height, kernel_depth;
    size_t stride_w, stride_h;

    ConvPadding padding;
    ConvActivation activation;

    const float *const weight;
    bool useBias;  const float *const bias;

public:
    Conv3d(size_t filters, size_t kernel_w, size_t kernel_h, size_t kernel_d,
           size_t stride_w, size_t stride_h, const float *const weights,
           bool useBias = false, const float *const biases = nullptr,
           ConvPadding padding = valid, ConvActivation activation = Linear)
            : filters(filters), /* ... */ {
        if (!/* vaild */) throw InvalidArgs(/* ... */);  // conv params, filters, kernel, stride
    }

    Img forward(const Img &img) const override {
        // pre-calc several variables: save little time and use (a little) faster stack memory

        Img res(opt_width, opt_height, opt_channels);
        const float *idata = img.ptr();
        float *odata = res.ptr();

        if (padding == valid) { /* similar to the padding mode */ } else {
        for (size_t o_ch = 0; o_ch < opt_channels; ++o_ch) {
            for (size_t i_ch = 0; i_ch < in_channels; ++i_ch) {
                for (size_t o_idx_r = 0; o_idx_r < opt_height; ++o_idx_r) {
                    for (size_t o_idx_c = 0; o_idx_c < opt_width; ++o_idx_c) {
                        for (size_t k_idx_r = 0; k_idx_r < kernel_height; ++k_idx_r) {
                        for (size_t k_idx_c = 0; k_idx_c < kernel_width; ++k_idx_c) {
                            if (/* if using padding and out of bound, skip this turn */)
                                continue;
                            odata[o_ch * o_channel_step + o_idx_r * opt_width + o_idx_c] +=
                                    idata[i_ch * i_channel_step
                                          + (o_idx_r * stride_h - k_bias_r2c + k_idx_r) * i_col
                                          + o_idx_c * stride_w - k_bias_c2c + k_idx_c] *
                                    weight[o_ch * ko_channel_step
                                           + i_ch * ki_channel_step
                                           + k_idx_r * kernel_width + k_idx_c];
        }}}}}}}

        if (useBias) {
            for (size_t o_ch = 0; o_ch < opt_channels; ++o_ch) {
                float _bias = bias[o_ch];
                for (size_t ce = 0; ce < o_channel_step; ++ce)
                    odata[o_ch * o_channel_step + ce] += _bias;
            }
```

```
        }
        switch (activation) {
            case Linear:
                break;
            case Relu: {
                size_t sum = opt_width * opt_height * opt_channels;
#pragma unroll 8
                for (size_t i = 0; i < sum; ++i)
                    if (odata[i] < 0) odata[i] = 0;
                break;
            }  // other cases
        }
        return res;
    }
    /* other functions are omitted */
};
```

根据以前的优化策略，在不大幅修改代码的情况下，我们可以采取 OpenMP 和 SIMD 进行加速（在 brute force 版下，卷积核大小一般较小，当宽度至少为 4 时才能使用 SIMD 达到部分加速效果，考虑到预训练的卷积核大小为 3，为节省时间，本次不对卷积核进行 SIMD；但对 bias 和 relu 仍有较好的效果）。

```
if (useBias) {
#if defined (__AVX512F__)   // similar to works in project 2,3,4
#elif defined (__ARM_NEON)  // similar to works in project 2,3,4
#else      // burte force, with unroll 8 as previously mentioned
#endif
}

switch (activation) {
    case Relu: {
        size_t sum = opt_width * opt_height * opt_channels;
#if defined (__AVX512F__)
        size_t cst = o_channel_step / 16;
        float *p_ubias = odata;
        __m512 zeros = _mm512_setzero_ps();
        for (size_t k = 0; k < cst; ++k) {
            _mm512_store_ps(p_ubias, _mm512_max_ps(zeros, _mm512_load_ps(p_ubias)));
            p_ubias += 16;
        }
#pragma unroll 8
        for (size_t k = sum - sum % 16; k < sum; ++k) {
            if (*p_ubias < 0) *p_ubias = 0.f;
            p_ubias++;
        }
// #elif defined (__ARM_NEON) is similar, #else is as previously mentioned
#endif
            break;
        }  // the following are omitted
```

```
}
```



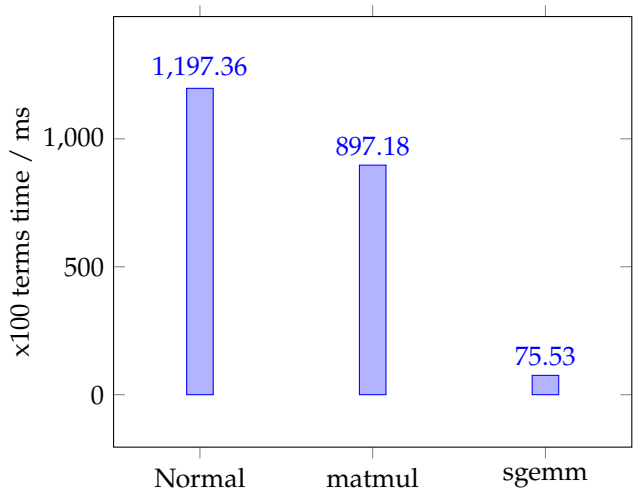
Analysis 当输入尺寸变大时，占主导地位的卷积操作耗时远超于 bias 和激活函数，也为数据分析带来不便，故使用 128 * 128 的图片。注意到在卷积步骤耗时不变的情况下，AVX512 通过将 16 个 float 向量化实现了 33.2% 的大幅提升，而 NEON 也带来了将近 3.1% 的提升，这也意味着在 bias + activate 步骤中，SIMD 带来的提升远大于此比例。

2.5.1 优化卷积：Im2Col+GEMM

作为 Caffe 等框架采用的经典提速方法，Im2Col 牺牲了部分空间与将元素赋值重排的时间，却能换取矩阵乘法的高度访存连续与更高的流水线效率，且由于此步已经访存不连续，不妨在转换时直接将其转置，得以进一步减少矩阵乘法的 cache miss——改进后性能提升约 15 倍！

```
for (size_t r = -rst; r < i_row + rst; r += stride_h) {
    for (size_t c = -cst; c < i_col + cst; c += stride_w) {
        for (size_t ch = 0; ch < kernel_depth; ++ch) {
            for (size_t i = 0; i < kernel_height; ++i) {
                for (size_t j = 0; j < kernel_width; ++j) {
                    if (r + i < 0 || r + i >= i_row || c + j < 0 || c + j >= i_col)
                        p_i2c[((r + rst) / stride_h * opt_width + (c + cst) / stride_w) *
                              i2c_col + ch * kernel_width * kernel_height +
                              i * kernel_width + j] = 0.f;
                    else
                        p_i2c[((r + rst) / stride_h * opt_width + (c + cst) / stride_w) *
                              i2c_col + ch * kernel_width * kernel_height +
                              i * kernel_width + j] =
                                idata[ch * i_channel_step + (r + i) * i_col + c + j];
                }
            }
        }
```

```
    }
}
Img res(opt_width, opt_height, opt_channels);
cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasTrans,
            opt_channels, opt_width * opt_height,
            i2c_col, 1, weight, i2c_col, p_i2c,
            i2c_col, 0, res.ptr(), opt_height * opt_width);  // or matmul in previous projects
```



改进：Im2Col 批量处理 padding 当 kernel 较大时，如果有 padding，显然其占据 Im2Col 矩阵的前数行与后数行，逐个元素判断 padding 并赋值较为耗时，如果我们准确计算并用 memset 对集中的 padding 设置，显然能节省大量时间[5]。

```
if (padding == same) {  // 输出的第一行与最后一行做大量padding
    memset(p_i2c, 0, sizeof(float) * im2col.data()->width * kernel_width * (kernel_height / 2));
    memset(p_i2c + (im2col.data()->height - kernel_width * (kernel_height / 2)) * im2col.data()
        ->width,
           0, sizeof(float) * im2col.data()->width * kernel_width * (kernel_height / 2));
} else if (padding == full) {
    memset(p_i2c, 0, sizeof(float) * im2col.data()->width * kernel_width * (kernel_height - 1));
    memset(p_i2c + (im2col.data()->height - kernel_width * (kernel_height - 1)) * im2col.data()
        ->width,
           0, sizeof(float) * im2col.data()->width * kernel_width * (kernel_height - 1));
}
```

针对模型 对于确定结构的模型，我们可以将部分代码整合，如删去兼容性的分支并将 bias 和 relu 和并；此外，由于 CNN 模型一般是不强调创建模型实例的时间，而追求缩短在模型整体（主要是输入层）确定后，我们可以提前为 Im2Col 和输出 Img 申请内存并将其引用传入 forward 函数，这样在每次 predict 时不用频繁的申请与释放内存。其余中间层的辅助 Img 也可预先开好并传入 forward。

2.6 Pooling

[5]但这使代码复杂且分支多，由于并非主流的实现方式，也无参考资料，计算其需要耗费大量精力，考虑到时间关系未实现。

```cpp
enum PoolRule { MaxPooling, AvgPooling, };

class Pooling3D : public CnnLayer {
    size_t kernel_h, kernel_w, kernel_c;
    size_t stride_h, stride_w, stride_c;
    PoolRule rule;
    /* functions */
};
```

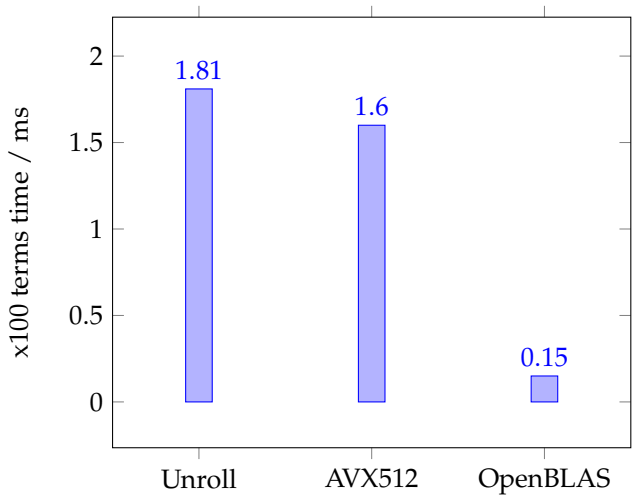
池化层同样具备完备的异常安全机制，且支持多种规则（代码与卷积层类似，故省略）。

2.7 FullConn

池化层包括了数个向量点乘及 Normalization（如本次用到的 SoftMax），显然，可以考虑用 OpenMP、SIMD 与 clbas_sdot 加速。其直接接受上一个 layer 的输出（Img），由于元素在内存布局为 NCHW ([channel][row][col])，直接以 channel * row * col 的一维数组形式读取其即可，无需 flatten 层。

```cpp
enum Normalization { SoftMax, /* ... */ };

class FullConn : public CnnLayer {
    size_t inFeatures;  size_t outFeatures;
    const float *const weights;  bool useBias;  const float *const biases;

    Normalization norm;

public:
    /* constructor */
    Img forward(const Img &img) const override {
        float *pimg = img.ptr();
        Img res(outFeatures, 1, 1);
        float *pres = res.ptr();
#if not defined(__DISABLE_BLAS)
        if (useBias) {
            for (size_t i = 0; i < outFeatures; ++i)
                pres[i] = cblas_sdsdot(inFeatures, biases[i], pimg, 1, weights + i * inFeatures,
                    1);
        } else {
            for (size_t i = 0; i < outFeatures; ++i)
                pres[i] = cblas_sdot(inFeatures, pimg, 1, weights + i * inFeatures, 1);
        }
#elif defined (__AVX512F__)
        size_t ifsimd = inFeatures / 16;
        for (size_t i = 0; i < outFeatures; ++i) {
            float tmp = 0.f;
            for (size_t j = 0; j < ifsimd; ++j) {
                tmp += _mm512_reduce_add_ps(_mm512_mul_ps(
                        _mm512_load_ps(weights + i * inFeatures + j * 16),
                        _mm512_load_ps(pimg + j * 16)));
            }
```

```
            for (size_t j = inFeatures - ifsimd * 16; j < inFeatures; ++j) tmp += weights[i *
                inFeatures + j] * pimg[j];
            pres[i] = tmp;
            if (useBias) pres[i] += biases[i];
        }
#elif defined (__ARM_NEON)  // omitted to save time
#else  // brute force
#endif

        switch (norm) {
            case SoftMax: {
                float sum = 0.f;
#pragma unroll 8
                for (size_t i = 0; i < outFeatures; ++i) {
                    sum += exp(pres[i]);
                }
#pragma unroll 8
                for (size_t i = 0; i < outFeatures; ++i) {
                    pres[i] = exp(pres[i]) / sum;
                }
                break;
            }  // more cases can be added
        }
        return res;
    }  // other functions
};
```



2.8 class Sequential

为了更好的封装模型细节，实现更完备的异常安全机制，我们将 layers 整合进 module，并利用虚函数使 layers 各司其职，序列化 CNN 的模型如下：

```
class Sequential {
    vector<CnnLayer *> m_layers;
```

```
    vector<size_t> m_optSize;

public:
    Sequential() = default;
    Sequential(initializer_list<CnnLayer *> layers) {
        if (dynamic_cast<ImgInput *>(*layers.begin()) == nullptr ||
            dynamic_cast<FullConn *>(*(layers.end() - 1)) == nullptr) {
            for (auto &l : layers)
                try { delete l; } catch (...) {}
            throw InvalidArgs("First layer must be ImgInput && last layer must be FullConn");
        }
        for (auto &layer : layers) {
            /**
             * the first layer <aka. ImgInput> actually doesn't care what inSize is
             * but should only has one in a module
             * since when the first time we check the m_optSize, it is empty
             * we can use this to check whether an ImgInput is added inside the module but not
                 the first layer
             */
            if (!layer->checkValid(m_optSize)) {
                for(auto& l:layers) delete l;
                throw InvalidArgs("Invalid structure, conflicts on", typeid(layer).name());
            }
            m_optSize = layer->optSize(m_optSize);
        }
        m_layers.insert(m_layers.end(), layers);
    }

    ~Sequential() {
        for (auto &layer : m_layers) { try { delete layer; } catch (...) {} }
    }

    void add(CnnLayer *layer) {
        if (m_layers.empty() && !dynamic_cast<ImgInput *>(layer)) {
            delete layer;
            throw InvalidArgs("The first layer must be ImgInput");
        }
        if (!layer->checkValid(m_optSize)) {
            delete layer;
            throw InvalidArgs("Module structure is invalid");
        }
        m_layers.push_back(layer);
        m_optSize = layer->optSize(m_optSize);
    }

    vector<float> predict(const char *img, int cvt = -1) {
        if (m_layers.empty() || !dynamic_cast<FullConn *>(*(m_layers.end() - 1)))
            throw CnnException("Module still incomplete");

        Img res = dynamic_cast<ImgInput *>(m_layers.at(0))->prepare(img, cvt);
```

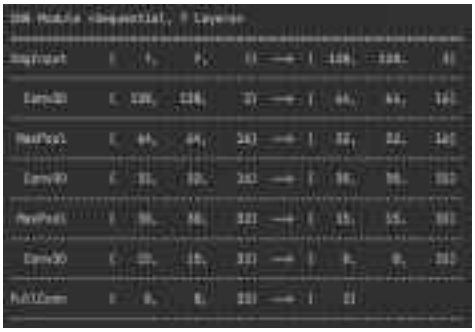
```
        for (int i = 1; i < m_layers.size(); ++i) {
            res = m_layers.at(i)->forward(res);
        }
        float *conf = res.ptr();
        vector<float> pred{conf[0],conf[1]};
        return pred;
    }

    friend ostream &operator<<(ostream &os, const Sequential &module) {/* ... */}
    void summary() { cout << (*this); }
};
```

要创建一个模型，可以使用无参构造器并将 layer 依次 add 进 m_layers，也可直接使用初始化列表；构造完的模型也只需调用 predict 函数即可将任意大小、任意通道（需要 OpenCV 有对应的转换规则）的图片输入模型并输出 vector。

3 测试样例及分析

3.1 正确性

模型构造阶段检查正确性：确保能进行 predict 的模型均有效

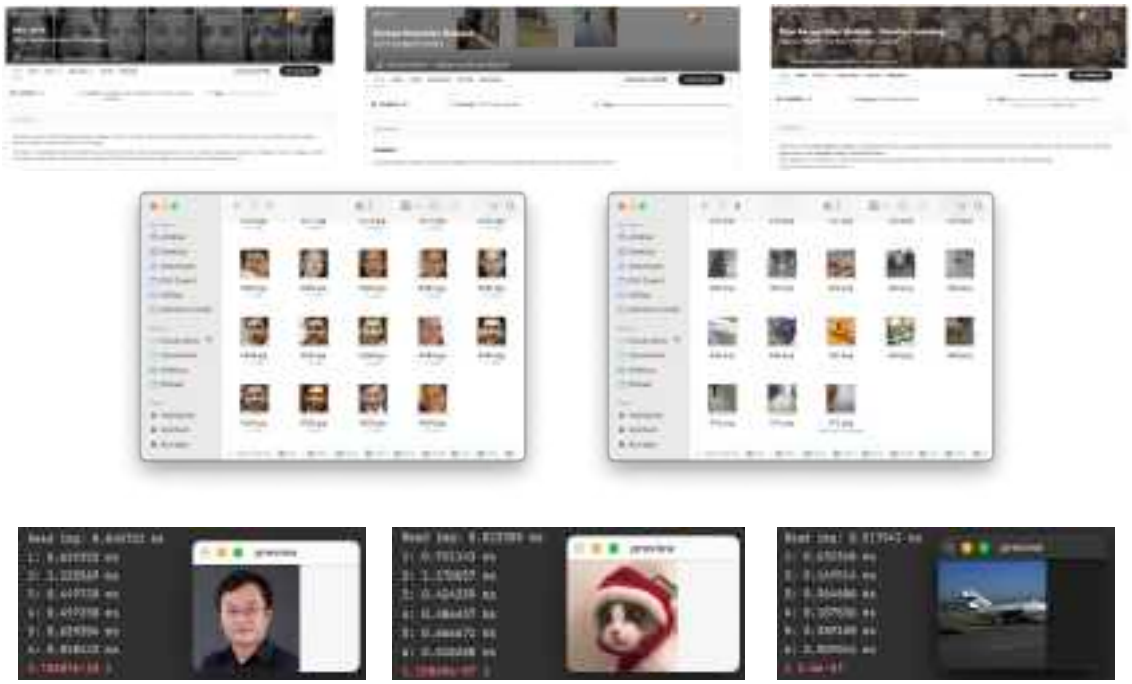


“漂亮的”模型框架 summary

基本的异常安全功能覆盖较全，更多细节请阅读源码

3.2 准确性

Method 从 Kaggle 相关数据集 [2–4] 中整理得含有人脸的正例 6094 张（均为 128 * 128, RGB）与环境图片 472 张（尺寸不一），遍历文件夹，统计 face confidence 落在区间及个数。



整体效果及各层时间分布（10 次预热） (What is this? ↑)

Confidence (+)	<.50	.50 ~.60	.60 ~.70	.70 ~.80	.80 ~.90	.90 ~.95	.95 ~1.0
demo.py	1029	281	344	452	718	629	2461
正例 (BGR)	1028	281	345	452	718	632	2458
正例 (RGB)	3	0	1	2	4	15	6069

Confidence (-)	<.50	.50 ~.60	.60 ~.70	.70 ~.80	.80 ~.90	.90 ~.95	.95 ~1.0
反例 (RGB)	68	8	0	10	24	361	1

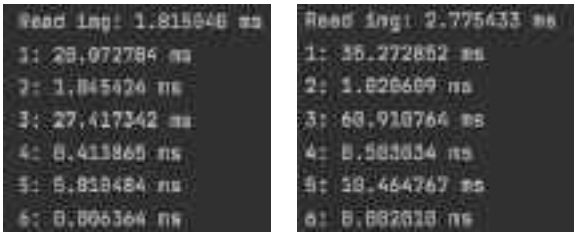
Analysis 从使用 BGR 进行预测与 demo.py 对比可知，本次代码实现正确，少数不一致应为 float 误差造成 confidence 在分类交界处；而作为 RGB 读入并预测，可发现本预训练模型识别人脸的正确性极高[6]，使用白种人大头照，选择 confidence ≥ 0.90 为阈值，可达 99.835% 正确率；而背景图案照片使用无拉伸变换为 128 * 128 且转为 RGB 后，模型表现远不及人脸识别，即使以 0.50 作为阈值，误分类率依然达到了 14.41%，这可能是由于相比起人脸这一种有明显特征（对 CNN 有利）的 object，背景图的 variability 更大，光阴等效果也可能使模型误认为人脸，因此模型仅能模糊的将其分为中间。

3.3 性能

	Brute force	+ SIMD + OMP	+ Im2Col + BLAS	+ -O3	+ Release
x86	59.60	21.47	3.15	1.25	1.03
ARM	109.82	55.32	3.38	2.32	1.05

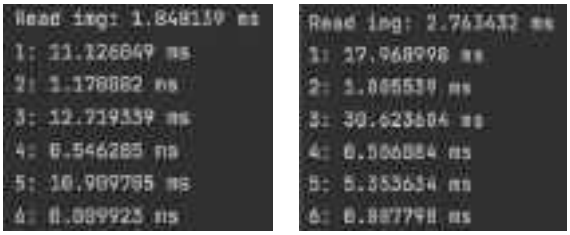
使用标准输入格式（128 * 128, 3 channels），预热 10 次，不计算 ImgInput 层（因其与硬盘性能有关，且观测到此部分时间波动较大），predict 100 次取平均。上表中“+”代表在左侧的优化基础上再对某项进行优化。与 project 4 结论类似，同样的程序在 x86 上能轻松运行，但到 ARM 上由于其架构不同则远慢于 x86；SIMD+OMP 的优化组合不出预料的将性能将近翻倍（x86 上由于使用了 AVX512 远优于 NEON，故提升更多）；但出人意料的是在使用了 OpenBLAS 后两者的性能接近持平！（OpenBLAS 针对各个平台均做了良好的适配优化，可见服务器的性能并非想象中的不堪）；而在使用 Im2Col 前开启 O3 优化效果甚至不如手动 SIMD+OMP，而在 Im2Col 明确“提示”编译器我们希望通过这种方式优化访存，其能进一步帮我们优化 Im2Col 的转换过程以及轻微的优化其余部分（但本身耗时就不多）；而 Release 模式下，不仅可能添加更多的优化方案，也移除了各种用于调试的代码，程序体积变小的同时再次提升性能。

```
SET(CMAKE_BUILD_TYPE "Release")
```

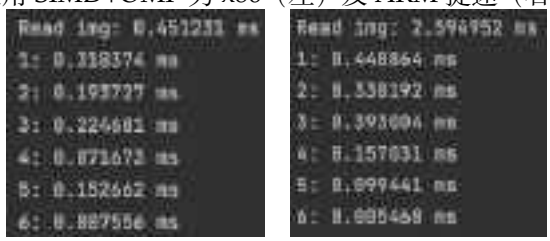


x86（左）与 ARM（右）Brute force 版本：可见主要时间开销（第 1，3，5 层）为卷积层

[6]好吧，在天真的模型看来，其实人脸无非就是点了两个点（眼睛），开了一条缝（嘴）的“球”——life is simple.



使用 SIMD+OMP 为 x86（左）及 ARM 提速（右）



使用 Im2Col + GEMM 有效的将卷积耗时提速十倍有余

4 困难及解决

4.1 内联函数

选择尽可能压榨算力还是提高程序拓展性并使代码简洁，是件值得斟酌的令人纠结的事。在设计通用的卷积层时，提供了多种激活函数的选择，在不考虑性能时，显然下面使用函数指针的代码拓展性较强：

```
enum ConvActivation { Relu, /* ... */ };
namespace conv::activation {
    inline float relu(float x) { return x > 0 ? x : 0; }  // ...
}

class Conv3d : public CnnLayer {
    float (*activate)(float);
};
```

这样只需在构造器中为函数指针赋值，在激活函数时只需要不断调用其即可。这时我们只会进行一次 switch，远优于封装成一个函数并在每次函数调用时 switch 激活函数的选择（即使将其声明为内联函数）。实际上，这种写法会强行使编译器为函数生成非内联版本，自然，性能不如内联函数版本。

上图对比函数指针与内联函数的性能差异（简单函数，运行 10^7 次）。而我们知道这方面的差异主要来源于调用函数这一过程，因此，在需要选择性的大量重复调用简单函数是，最好的方法是减少判断次数并使用内联函数，即本次 Conv3d 所用方案。

4.2 One more thing

C++ 的性能固然强大，然而其语法远没有 Python 简洁。最好的搭配是使用 C++ 写底层并供 Python 调用，这也是 Keras 等库所采取的策略，也是我们未来很有可能需要具备的一项技能（使用 C++ 实现人工智能相关的底层，并用 Python 拼接逻辑）。本次的前端使用 Python 完成，并在 Python 调用生成的动态库。

```cpp
// 1. 封装C接口   2. 打包动态库  3. Python配置接入动态库并使用
extern "C" {
    /**
     * @return the confidence of <face>
     */
    float pyjudge(const char *filename, int cvt) {
        vector<float> res = model.predict(filename, cvt);
        return res[1];
    }
}
```


```python
import ctypes

lib = ctypes.cdll.LoadLibrary('../libfacedetect.so')
lib.pyjudge.restype = ctypes.c_float

print(lib.pyjudge('../test/1.jpg'))
```

5 总结

本次 project 抛出看似宏伟的题目，但实际做下来发现遇到的许多小困难在耐心与细致的检查下均能迎刃而解，在这过程中我也学到了许多有趣的知识，这也为我以后其他课程的学习增添了信心。本次 project 也是本学期做过的“最有趣”的 project 之一，如果时间允许，或许能探索 FFT、数据打包 (Pack)、MEC 等优化策略。

References

[1] Wikipedia contributors, "Pipeline (computing)," 10 2021. [Online]. Available: https://en.wikipedia.org/wiki/Pipeline_(computing)

[2] "FER-2013," 07 2020. [Online]. Available: https://www.kaggle.com/msambare/fer2013

[3] "FaceMask Dataset," 06 2020. [Online]. Available: https://www.kaggle.com/sumansid/facemask-dataset

[4] "Human Detection Dataset," 08 2021. [Online]. Available: https://www.kaggle.com/constantinwerner/human-detection-dataset

Disclaimer: All references used in this project are only for the purpose of guiding the direction of exploration and ensuring correctness, any code appearing in the referenced web pages has not been used directly, unless explicitly stated.