# C++实现计算图的前向推理

## 1. 需求定义

> **实现一个包含Max Pooling操作和张量求和操作的计算图的前馈过程:**
>
> $Formulation: dst[32,64,56,56] = add(max_pooling(src1[32,64,112,112]), src2[32,1,56,56])$

## 2. 分析与实现

### 2.1数据结构定义:

在深度学习框架中涉及的数据都是张量(Tensor), 即高维数组，本次C++实现中，我采用嵌套的STL Vector来定义Tensor:

```
//定义Tensor为四维矩阵[b, c, h, w]
typedef vector<vector<vector<vector<int> > > > Tensor;

//定义一个函数获得四维tensor[b, c, h, w]
Tensor getTensor(int batch, int channels, int height, int width, int init){
    vector<int> w(width, init);
    vector<vector<int> > h(height, w);
    vector<vector<vector<int> > > c(channels, h);
    Tensor t(batch, c);
    return t;
}
```

### 2.2 MaxPooling操作

MaxPooling从本质上来说是对Tensor的最后两个维度$(h, w)$，在固定尺寸的滑动窗口内求最大值的过程，包含的参数有:

- kernel_size : 滑动窗口的尺寸
- padding: 边界填充大小
- strides: 淮东窗口每次移动的step

本次实现中，首先根据输入的tensor和参数计算输出的特征图尺寸，输出的尺寸由以下公式得到

$$Out = \frac{Input + 2 * padding - kernel\_size}{stride} + 1 \tag{1}$$

随后将图像的输入填充(resize)成 $[b, c, h + 2 * padding, w + 2 * padding]$, 相当于做了0填充, 最后通过多层for循环完成计算，代码如下:

```
//max_pooling函数
Tensor maxPooling(Tensor& t, vector<int> kernel_size, vector<int> pad,
vector<int> stride){
    int batch_size = t.size();
    int channels = t[0].size();
    int height = t[0][0].size();
    int width = t[0][0][0].size();

    int out_height = (height + 2*pad[0] - kernel_size[0]) / stride[0] + 1;
    int out_width = (width + 2*pad[1] - kernel_size[1]) / stride[1] + 1;
```

```
    Tensor output = getTensor(batch_size, channels, out_height, out_width, 0);

    //先进行0填充,原始 t resize成[b, c, height+2*pad, width+2*pad]
    for(int b=0; b<batch_size; b++){
        for(int c=0; c<channels; c++){
            t[b][c].resize(height + 2*pad[0]);
            for(int h=0; h<height+2*pad[0]; h++){
                t[b][c][h].resize(width + 2*pad[1]);
            }
        }
    }

    //max_pooling功能， 通过循环体求h,w维度上一个 kernel_size邻域内的最大值
    for(int b=0; b<batch_size; b++){
        for(int c=0; c<channels; c++){
            for(int h=0; h<out_height; h++){
                for(int w=0; w<out_width; w++){
                    //滑动窗口的起始位置
                    int sx = h*stride[0];
                    int sy = w*stride[1];
                    int max_element = t[b][c][sx][sy];
                    //内层二重循环寻找最大
                    for(int i=0; i<kernel_size[0]; i++){
                        for(int j=0; j<kernel_size[1]; j++){
                            if(t[b][c][sx+i][sy+j] > max_element){
                                max_element = t[b][c][sx+i][sy+j];
                            }
                        }
                    }
                    output[b][c][h][w] = max_element;
                }
            }
        }
    }
    return output;
}
```

### 2.3 Tensor Add操作

张量的加法定义为对应元素相加，这里会涉及到张量广播的操作，在numpy中的broadcast规则为“**从后往前匹配，维度一致或者其中一个为1才能broadcast**”，本次通过c++ for 循环实现如下:

```
Tensor tensorAdd(Tensor& t1, Tensor& t2){
    //[b, c, h, w]维度
    int b1 = t1.size(), c1 = t1[0].size(), h1 = t1[0][0].size(), w1 = t1[0][0]
[0].size();
    int b2 = t2.size(), c2 = t2[0].size(), h2 = t2[0][0].size(), w2 = t2[0][0]
[0].size();
    int res_b = max(b1, b2), res_c = max(c1, c2), res_h = max(h1, h2), res_w =
max(w1, w2);
    Tensor resTensor = getTensor(res_b, res_c, res_h, res_w, 0);
    //维度检查
    if(w1 != w2 && w1 != 1 && w2 != 1){
        cout<<"Dimension not matched"<<endl;
        return resTensor;
```

```
    }
    if(h1 != h2 && h1 != 1 && h2 != 1){
        cout<<"Dimension not matched"<<endl;
        return resTensor;
    }
    if(c1 != c2 && c1 != 1 && c2 != 1){
        cout<<"Dimension not matched"<<endl;
        return resTensor;
    }
    if(b1 != b2 && b1 != 1 && b2 != 1){
        cout<<"Dimension not matched"<<endl;
        return resTensor;
    }
    //矩阵相加
    for(int b=0; b<res_b; b++){
        int b_1 = min(b, b1-1);
        int b_2 = min(b, b2-1);
        for(int c=0; c<res_c; c++){
            int c_1 = min(c, c1-1);
            int c_2 = min(c, c2-1);
            for(int h=0; h<res_h; h++){
                int h_1 = min(h, h1-1);
                int h_2 = min(h, h2-1);
                //当最后一个维度均为1
                if(w1 == 1 && w2 == 1){
                    resTensor[b][c][h][0] = t1[b_1][c_1][h_1][0] + t2[b_2][c_2]
[h_2][0];
                }else{
                    for(int w=0; w<res_w; w++)
                    if(w1 == 1){
                     resTensor[b][c][h][w] = t1[b_1][c_1][h_1][0] + t2[b_2][c_2]
[h_2][w];
                    }else if(w2 == 1){
                     resTensor[b][c][h][w] = t1[b_1][c_1][h_1][w] + t2[b_2][c_2]
[h_2][0];
                    }else{
                     resTensor[b][c][h][w] = t1[b_1][c_1][h_1][w] + t2[b_2][c_2]
[h_2][w];
                    }
                }
            }
        }
    }
    return resTensor;
}
```

## 3. 代码优化

以上的基本实现可以从以下几个方面进行性能优化：

- 基于OpenMP的并行优化，主要优化了MaxPooling函数内的六层循环；
- 基于SIMD(单指令多数据)优化，主要优化了tensorAdd中的向量加法；
- g++ 编译选项优化, 例如-O3级别优化；

  原始实现： vanillia_implement.cpp

优化实现：optimize_implement.cpp

## 4. 速度对比：

针对样例输入的速度对比如下

| 实现方式 | 速度 |
| --- | --- |
| 原始实现 | 2454 ms |
| OpenMP + SIMD优化 | 1340 ms |
| g++ -O3 优化 | 211ms |

## 5. 编译运行

项目Github链接：Compute_graph

### 1. Vanillia implement

```
g++ .\vanillia_implement.cpp -o vanillia_implement
.\vanillia_implement.exe
```

**output:**

> Result Shape: 32 64 56 56
> Time Consume: 2454 ms

### 2. Optimize implement

```
g++ -fopenmp  .\optimize_implement.cpp -o optimize_implement
.\optimize_implement.exe
```

**output:**

> Result Shape: 32 64 56 56
> Time Consume: 1340 ms

### 3. Optimize by g++

```
g++ -fopenmp -O3 .\optimize_implement.cpp -o optimize_implement
.\optimize_implement.exe
```

**output:**

> Result Shape: 32 64 56 56
> Time Consume: 211 ms