华为昇腾-昇思MindSpore《AI计算系统》精品课程

HCIA-AI计算系统

精品课程

实验指导手册

版本:1.0



华为技术有限公司

|  |
| --- |
| 版权所有 © 华为技术有限公司 2021。 保留一切权利。  非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。  商标声明  C:\Users\jwx341670\Desktop\华为标志 Huawei Logo 2018\竖版标志Vertical Version\PNG\HW_POS_RBG_Vertical-150ppi.png 和其他华为商标均为华为技术有限公司的商标。  本文档提及的其他所有商标或注册商标，由各自的所有人拥有。  注意  您购买的产品、服务或特性等应受华为公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为公司对本文档内容不做任何明示或暗示的声明或保证。  由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。 |

|  |  |
| --- | --- |
| 华为技术有限公司 | |
| 地址： | 深圳市龙岗区坂田华为总部办公楼 邮编：518129 |
| 网址： | http://[e](http://e.huawei.com/).huawei.com |

目录

[5 基于CANN构建VGG17 4](#_Toc1216407632)

[5.1实验介绍 4](#_Toc118751584)

[5.1.1 模型知识点的介绍 4](#_Toc845564225)

[5.2 实验环境要求 7](#_Toc1498637376)

[5.3 实验详细设计与实现 7](#_Toc1910166416)

[5.3.1 实验代码目录介绍 7](#_Toc1433914709)

[5.3.2 实验步骤 7](#_Toc743027529)

[5.4 实验总结 27](#_Toc446272598)

[5.5 实验任务与参考解答任务 28](#_Toc1490659262)

[5.5.1 实验任务 28](#_Toc965990532)

[5.5.2 参考答案 28](#_Toc426500004)

# 基于CANN构建VGG17

## 5.1实验介绍

本实验利用CANN算子库完成VGG17的算子构图。

* 本实验基于CANN高性能算子库构建VGG17网络
* 加载提供的权重（实验四得到的VGG17权重），编译graph得到离线模型，并利用实验三的测试样例对构建的模型进行推理验证，得到推理结果。

### 模型知识点的介绍

**5.1.1.1 Tensor**

Tensor是算子中的数据，包括输入数据与输出数据，TensorDesc类用于构建tensor对象，实例化tensor对象的shape、dtype和format。TensorDesc是对输入数据与输出数据的描述，TensorDesc数据结构包含的属性如下表所示：

**5.1.1.2 数据排布格式**

Format为数据的物理排布格式，定义了解读数据的维度，比如1D，2D，3D，4D，5D等。本次实验涉及到的数据排布格式为以下两种，同学们可在每个算子定义的函数内查看。

**1. ND**

ND支持原图中的所有数据排布格式。

**2. NCHW和NHWC**

在深度学习领域，多维数据通过多维数组存储，比如卷积神经网络的特征图（Feature Map）通常用四维数组保存，即4D，4D格式解释如下：

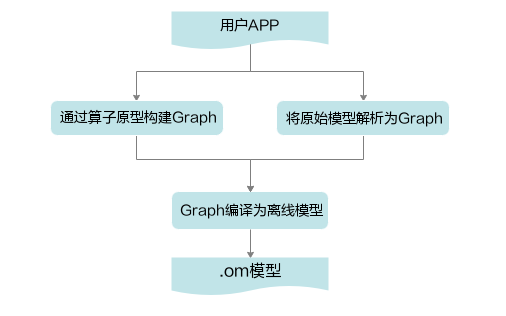
* N：Batch数量，例如图像的数目。
* H：Height，特征图高度，即垂直高度方向的像素个数。
* W：Width，特征图宽度，即水平宽度方向的像素个数。
* C：Channels，特征图通道，例如彩色RGB图像的Channels为3。

由于数据只能线性存储，因此这四个维度有对应的顺序。不同深度学习框架会按照不同的顺序存储特征图数据，比如Caffe，排列顺序为[Batch, Channels, Height,Width]，即NCHW。TensorFlow中，排列顺序为[Batch, Height, Width, Channels]，即NHWC。

**5.1.1.3 算子**

深度学习算法由一个个计算单元组成，我们称这些计算单元为算子（Operator，简称 OP）。在网络模型中，算子对应层中的计算逻辑，例如：卷积层（Convolution Layer）是一个算子；全连接层（Fully-connected Layer， FC layer）中的权值求和过程，是一个算子。

而在模型的推理时，我们需要将网络模型解析为网络图（Graph）或者通过算子原型构造出Graph，Graph编译成离线模型，而后将推理数据送到网络中得到推理结果。



**5.1.1.4 算子原型**

算子原型定义了算子的数学含义，包含定义算子输入、输出、属性。

在开发阶段使用 REG\_OP 宏，以“.”链接 INPUT、OUTPUT、ATTR 等接口注册算子的输入、输出和属性信息，最终以 \*OP\_END\_FACTORY\_REG\* 接口结束，完成算子的定义。REG\_OP 宏相当于定义一个该算子类型的类，后续开发者可以通过实例化该算子类来调用算子。

算子注册得到代码实现如下所示：

namespace ge{

REG\_OP(OpType) //算子类型名称

.INPUT(x1, TensorType({ DT\_FLOAT, DT\_INT32 })) //算子输入信息

.INPUT(x2, TensorType({ DT\_FLOAT, DT\_INT32 })) //算子输入信息

.DYNAMIC\_INPUT(x, TensorType{DT\_FLOAT, DT\_INT32}) //动态多输入场景下算子的输入信息

.OPTIONAL\_INPUT(b, TensorType{DT\_FLOAT}) //可选输入算子的输入信息

.OUTPUT(y, TensorType({ DT\_FLOAT, DT\_INT32 })) //算子的输出信息

.DYNAMIC\_OUTPUT(y, TensorType{DT\_FLOAT, DT\_INT32}) //动态多输入场景下算子的输出信息

.ATTR(x, Type, DefaultValue) //可选算子属性，当开发者不设置算子对象属性时使用默认值

.REQUIRED\_ATTR(x, Type) //必选算子属性，无默认值，开发者必须设置算子对象的属性值

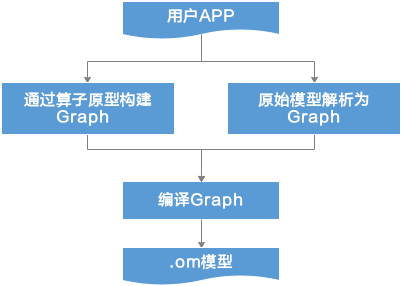
.OP\_END\_FACTORY\_REG(OpType) }

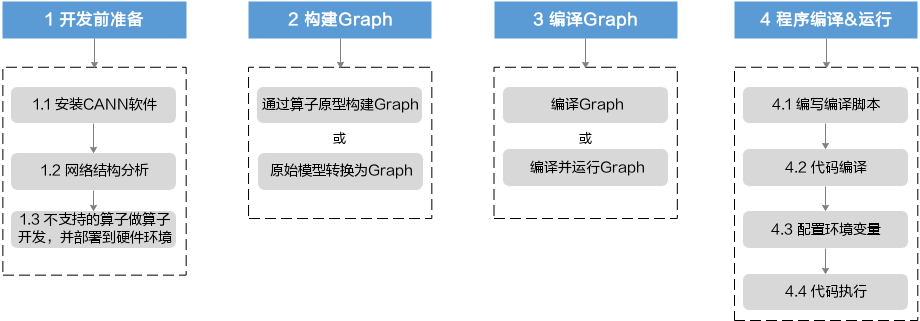
算子原型创建和代码详解请见[CANN算子原型](https://support.huawei.com/enterprise/zh/doc/EDOC1100191908/3e70ef)。

生成的算子信息会被注册到算子原型库中并自动生成对应的算子衍生接口。用户可通过这些接口在Graph中定义算子，然后创建一个Graph实例，并在Graph中设置输入算子、输出算子，从而完成Graph构建。

**5.1.1.5 构建、编译和运行Graph的流程**

用户可以通过开放的Ascend Graph编程接口，基于算子原型进行构图，并编译为离线模型，用于在昇腾AI处理器上进行离线推理。同时，也可以通过框架解析功能将主流的模型格式转换成CANN模型格式，从而隔离上层框架的差异，当前仅支持对Caffe/Tensorflow/ONNX原始框架模型的解析。





## 实验环境要求

* 操作系统：Ubuntu x86
* 编译器：g++
* 芯片：Ascend310
* python及的库：python3.7.5
* 已完成昇腾AI软件栈在开发环境上的部署（CANN环境，需要完成驱动及CANN软件的安装，关于CANN环境的安装参考[官方文档](https://support.huawei.com/enterprise/zh/doc/EDOC1100191934)）。

## 实验详细设计与实现

### 实验代码目录介绍

本实验的代码目录介绍：

./

├── CMakeLists.txt //编译脚本

├── data // 数据集文件夹

├── Makefile //编译脚本

├── out //模型生成目录

├── scripts

├── host\_version.conf

└── testcase\_300.sh // 编译运行脚本

└── src

└── main.cpp // 实现文件

### 实验步骤

构建算子

在构建Graph之前，开发者需要了解算子模型，明确构建Graph所需的算子。

本实验不涉及自定义算子，因此对于已经在算子库注册了算子原型的算子，根据其注册名，从以下两种方式获取算子原型：

* 从算子清单中获得[Ascend310算子清单](https://support.huawei.com/enterprise/zh/doc/EDOC1100180799)
* 从“OPP安装目录/opp/op\_proto/built-in/inc”算子原型定义的头文件中获取，例如：

REG\_OP(SoftmaxV2) //算子类型名称

.INPUT(x, TensorType({DT\_DOUBLE, DT\_FLOAT16, DT\_FLOAT}))

.OUTPUT(y, TensorType({DT\_DOUBLE, DT\_FLOAT16, DT\_FLOAT}))

.ATTR(axes, ListInt, {-1})

.OP\_END\_FACTORY\_REG(SoftmaxV2)

1. **Conv2D**

Conv2D算子原型定义如下：

REG\_OP(Conv2D)

.INPUT(x, TensorType({DT\_FLOAT16, DT\_FLOAT, DT\_INT8})) //定义数据输入

.INPUT(filter, TensorType({DT\_FLOAT16, DT\_FLOAT, DT\_INT8})) //定义卷积核输入

.OPTIONAL\_INPUT(bias, TensorType({DT\_FLOAT16, DT\_FLOAT, DT\_INT32})) //可选输入，卷积核的偏置bias

.OPTIONAL\_INPUT(offset\_w, TensorType({DT\_INT8})) //可选输入，卷积核的offset\_w，仍在算子清单中保留

.OUTPUT(y, TensorType({DT\_FLOAT16, DT\_FLOAT, DT\_INT32}))

.REQUIRED\_ATTR(strides, ListInt) //定义Conv2D的属性strides

.REQUIRED\_ATTR(pads, ListInt) //定义Conv2D的属性pads

.ATTR(dilations, ListInt, {1, 1, 1, 1}) //定义Conv2D的属性dilations

.ATTR(groups, Int, 1) //定义Conv2D属性group

.ATTR(data\_format, String, "NHWC") //定义Conv2D的数据输入格式

.ATTR(offset\_x, Int, 0) //定义Conv2D的

.OP\_END\_FACTORY\_REG(Conv2D) //结束算子注册

从Conv2D算子原型定义可以看到，Conv2D算子包括：

* 两个必选输入（INPUT）：x 和filter
* 两个可选输入（OPTIONAL\_INPUT）：bias 和 offset\_w
* 两个必选属性（ATTR）：strides、pads
* 四个可选属性（REQUIRED\_ATTR）：dilations、groups、data\_format、offset\_x

Conv2D算子定义的代码实现如下：

/\*

>> 生成Conv2D算子：

weight\_shape: 用于定义卷积算子filter的权重shape

conv\_name：卷积算子的名称

data：上一个算子。进行构图的连接

\*/

Operator GenConv2dOp(Shape weight\_shape,string conv\_name,Operator data){

//构造权重算子的描述信息desc\_weight

TensorDesc desc\_weight(weight\_shape, FORMAT\_NCHW, DT\_FLOAT);

//构造tensor

Tensor weight\_tensor(desc\_weight);

//计算出tensor需要的大小

uint32\_t weight\_len = weight\_shape.GetShapeSize() \* sizeof(float);

//从bin文件中加载数据，赋给tensor

bool res = GetConstTensorFromBin(kPath+conv\_name+".weight.bin", weight\_tensor, weight\_len);

if (!res) {

cout << \_\_LINE\_\_ << "GetConstTensorFromBin Failed!" << endl;

}

//创建Const类型的权重算子，通过Const算子的属性value，传入tensor

auto conv\_weight = op::Const(conv\_name+"\_weight")

.set\_attr\_value(weight\_tensor);

//创建卷积算子

auto conv2d = op::Conv2D(conv\_name)

.set\_input\_x(data) //定义输入，传入上一个算子

.set\_input\_filter(conv\_weight) //定义卷积核，传入卷积核的权重

.set\_attr\_strides({ 1, 1, 1, 1 }) //定义strides

.set\_attr\_pads({ 1, 1, 1, 1 }) //定义pads

.set\_attr\_dilations({ 1, 1, 1, 1 }) //定义dilations

.set\_attr\_data\_format("NCHW"); //定义输入数据的格式

TensorDesc conv2d\_input\_desc\_x(ge::Shape(), FORMAT\_NCHW, DT\_FLOAT);

TensorDesc conv2d\_input\_desc\_filter(ge::Shape(), FORMAT\_NCHW, DT\_FLOAT);

TensorDesc conv2d\_output\_desc\_y(ge::Shape(), FORMAT\_NCHW, DT\_FLOAT);

conv2d.update\_input\_desc\_x(conv2d\_input\_desc\_x); //更新卷积的输入信息

conv2d.update\_input\_desc\_filter(conv2d\_input\_desc\_filter);//更新卷积的filter信息

conv2d.update\_output\_desc\_y(conv2d\_output\_desc\_y); //更新卷积的输出信息

return conv2d;

}

1. 对于算子必选输入和可选输入：调用set\_input\_输入名称 接口

.set\_input\_x(data)

.set\_input\_filter(conv\_weight)

.set\_input\_bias(conv\_bias)

* data为整个graph的输入节点，通过Data算子构造，具体请参考[定义数据节点（Const）](https://support.huawei.com/enterprise/zh/doc/EDOC1100191908?section=j008)。
* conv\_weight为常量数据，通过Const算子构造，具体请参考[定义数据节点（Const）](https://support.huawei.com/enterprise/zh/doc/EDOC1100191908?section=j009)。
* conv\_bias为常量数据，通过Const算子构造，具体请参考[定义数据节点（Const）](https://support.huawei.com/enterprise/zh/doc/EDOC1100191908?section=j009)。

1. 对于算子\*\*必选属性和可选属性\*\*：调用 set\_attr\_属性名称 接口

.set\_attr\_strides({1, 1, 1, 1}) //设置strides属性值

.set\_attr\_pads({1, 1, 1, 1}) //设置pads属性值

.set\_attr\_dilations({1, 1, 1, 1}); //设置dilations属性值

1. 对于Conv2D等卷积类或对C轴处理敏感的算子，建议通过update\_input\_desc\_输入名称 接口将Format信息设置为NCHW或者NHWC等，具体和用户需要处理的Format格式保持一致。
2. **Data**

**Data算子是计算图数据输入节点，也是数据节点，算子原型定义如下：**

REG\_OP(Data)

.INPUT(x, TensorType::ALL()) //定义输入信息，输入支持所有数据类型的输入

.OUTPUT(y, TensorType::ALL()) //定义输出信息，输出支持所有数据类型的输出

.ATTR(index, Int, 0) //定义算子属性

.OP\_END\_FACTORY\_REG(Data) //结束算子注册

REG\_OP(Data)

.INPUT(x, TensorType::ALL()) //定义输入信息，输入支持所有数据类型的输入

.OUTPUT(y, TensorType::ALL()) //定义输出信息，输出支持所有数据类型的输出

.ATTR(index, Int, 0) //定义算子属性

.OP\_END\_FACTORY\_REG(Data) //结束算子注册

根据Data算子原型定义创建Data算子实例，实例代码如下。

定义Data类算子，名为data1，定义data1算子的信息描述为desc\_data，desc\_data定义了输入数据的shape。data1通过“update\_input\_\* desc\_输入名称 ”和“ update\_output\_desc\_输出名称”接口设置算子的Shape、Format和Dtype，更新算子的信息描述。

//实例化Data算子的输入数据的描述信息

auto shape\_data = vector<int64\_t>({1,3,224,224}); //输入数据[N,C,W,H],推理时batchsize为1

TensorDesc desc\_data(ge::Shape(shape\_data), FORMAT\_NCHW, DT\_FLOAT); //实例化一个算子信息描述的TensorDesc类，定义传入数据的shape，并将shape传入desc\_data

// 实例化Data算子，名为data1

auto data = op::Data("data1");

data.update\_input\_desc\_x(desc\_data); //更新data1算子的输入数据信息描述，定义输入数据的shape，format和dtype

data.update\_output\_desc\_y(desc\_data); //更新data1算子的输出数据信息描述，定义输入数据的shape，format和dtype

1. **Const**

权值、偏置等信息为常量Tensor，可以使用Const算子实现。Const算子定义如下：

REG\_OP(Const)

.OUTPUT(y, TensorType({DT\_FLOAT, DT\_FLOAT16, DT\_INT8, DT\_INT16, DT\_UINT16, \

DT\_UINT8, DT\_INT32, DT\_INT64, DT\_UINT32, DT\_UINT64, DT\_BOOL, DT\_DOUBLE})) //定义算子的输出，支持如上类型

.ATTR(value, Tensor, Tensor()) //定义算子的属性，value表示常量算子对应的值

.OP\_END\_FACTORY\_REG(Const) //结束算子定义

Const算子一般从bin文件中读取权重，偏置等数据，构造权重算子的代码实现如下：

//构造权重算子的描述信息desc\_weight\_1

auto weight\_shape = ge::Shape({ 5,17,1,1 }); //实例化一个Shape类

TensorDesc desc\_weight\_1(weight\_shape, FORMAT\_NCHW, DT\_INT8); //实例化一个算子信息描述的TensorDesc类，得到权重的描述信息desc\_weight\_1

//从bin文件中加载数据，构造tensor，为了将tensor传入权重算子里

Tensor weight\_tensor(desc\_weight\_1); //构造tensor

uint32\_t weight\_1\_len = weight\_shape.GetShapeSize(); //bin文件中读取数据，赋给tensor

bool res = GetConstTensorFromBin(PATH+"const\_0.bin", weight\_tensor, weight\_1\_len);

//创建Const类型的权重算子，通过Const算子的属性value，传入tensor

auto conv\_weight = op::Const("const\_0").set\_attr\_value(weight\_tensor);

bool GetConstTensorFromBin(string path, Tensor &weight, uint32\_t len) 函数表示从bin文件中加载数据：

* path：入参，指定权重文件路径，用于到固定目录例如“../data/weight/”下查找权重文件xx.bin，用户需要自行将权重文件解析为bin文件。（\*\*本实验提供好网络每一层转换好的权重，偏置bin文件\*\*）
* weight：出参，从权重文件中读取的Tensor类型的权重数据。
* len：入参，指定权重数据大小。

1. **MaxPool**

MaxPool算子定义如下，相关定义解释请见前三个算子：

REG\_OP(MaxPool)

.INPUT(x, TensorType({DT\_FLOAT16, DT\_FLOAT32, DT\_DOUBLE, DT\_INT8,

DT\_INT16, DT\_INT32, DT\_INT64, DT\_UINT8,

DT\_UINT16, DT\_QINT8}))

.OUTPUT(y, TensorType({DT\_FLOAT16, DT\_FLOAT32, DT\_DOUBLE, DT\_INT8,

DT\_INT16, DT\_INT32, DT\_INT64, DT\_UINT8, DT\_UINT16, DT\_QINT8}))

.REQUIRED\_ATTR(ksize, ListInt)

.REQUIRED\_ATTR(strides, ListInt)

.REQUIRED\_ATTR(padding, String)

.ATTR(data\_format, String, "NHWC")

.OP\_END\_FACTORY\_REG(MaxPool)

Maxpool算子的代码实现如下：

\*

>> 生成Maxpool算子

pool\_name：Maxpool算子的名称

data：上一个算子。进行构图的连接

\*/

Operator GenMaxpoolOp(string pool\_name,Operator data){

auto maxpool = op::MaxPoolV3(pool\_name)

.set\_input\_x(data)

.set\_attr\_strides({1,1,2,2}) // 代表在四个维度（batch、 height,、width、channels）所移动的步长

.set\_attr\_ksize({1,1,2,2}) //代表在四个维度（batch、 height,、width、channels）池化的尺寸，一般是[1, height, width, 1]

.set\_attr\_pads({0,0,0,0})

.set\_attr\_data\_format("NCHW")

.set\_attr\_padding\_mode("CALCULATED") //padding\_mode默认CALCULATED，三种模式 "SAME" "VALID" or "CALCULATE"

.set\_attr\_global\_pooling(false)

.set\_attr\_ceil\_mode(false); //是否在计算输出shape时，使用向上整取，默认false

TensorDesc tensor\_desc(ge::Shape(), FORMAT\_NCHW, DT\_FLOAT);

maxpool.update\_input\_desc\_x(tensor\_desc);

maxpool.update\_output\_desc\_y(tensor\_desc);

return maxpool;

}

1. **Relu**

Relu算子的定义如下**：**

REG\_OP(Relu)

.INPUT(x, TensorType({DT\_FLOAT, DT\_FLOAT16, DT\_DOUBLE,

DT\_INT8, DT\_INT32, DT\_INT16, DT\_INT64,

DT\_UINT8, DT\_UINT16, DT\_QINT8}))

.OUTPUT(y, TensorType({DT\_FLOAT, DT\_FLOAT16, DT\_DOUBLE,

DT\_INT8, DT\_INT32, DT\_INT16, DT\_INT64,

DT\_UINT8, DT\_UINT16, DT\_QINT8}))

.OP\_END\_FACTORY\_REG(Relu)

Relu算子的代码实现如下：

/\*

>> 生成Relu算子

relu\_name：relu算子的名称

data：上一个算子。进行构图的连接

\*/

Operator GenReluOp(string relu\_name,Operator data){

// 因为relu算子接在bn算子后面，bn算子有多个输出，得指明是data为"y"的输出传入relu，防止因BN有多个输出造成图不明确

auto relu = op::Relu(relu\_name).set\_input\_x(data, "y");

TensorDesc tensor\_desc(ge::Shape(), FORMAT\_ND, DT\_FLOAT);

relu.update\_input\_desc\_x(tensor\_desc);

relu.update\_output\_desc\_y(tensor\_desc);

}

1. **BatchNorm**

BatchNorm算子定义如下：

REG\_OP(BatchNorm)

.INPUT(x, TensorType({DT\_FLOAT16,DT\_FLOAT}))

.INPUT(scale, TensorType({DT\_FLOAT}))

.INPUT(offset, TensorType({DT\_FLOAT}))

.OPTIONAL\_INPUT(mean, TensorType({DT\_FLOAT}))

.OPTIONAL\_INPUT(variance, TensorType({DT\_FLOAT}))

.OUTPUT(y, TensorType({DT\_FLOAT16,DT\_FLOAT}))

.OUTPUT(batch\_mean, TensorType({DT\_FLOAT}))

.OUTPUT(batch\_variance, TensorType({DT\_FLOAT}))

.OUTPUT(reserve\_space\_1, TensorType({DT\_FLOAT}))

.OUTPUT(reserve\_space\_2, TensorType({DT\_FLOAT}))

.OUTPUT(reserve\_space\_3, TensorType({DT\_FLOAT}))

.ATTR(epsilon, Float, 0.0001)

.ATTR(data\_format, String, "NHWC")

.ATTR(is\_training, Bool, true)

.OP\_END\_FACTORY\_REG(BatchNorm)

BatchNorm算子的代码实现如下：

/\*

>> 生成Batch Normalization算子

weight\_shape：用于定义BN算子的四个输入权重的shape

bn\_name：BN算子的名称

data：上一个算子。进行构图的连接

\*/

Operator GenBNOp(Shape weight\_shape,string bn\_name, Operator data){

TensorDesc desc\_weight\_1(weight\_shape, FORMAT\_NCHW, DT\_FLOAT);

//定义BN算子的四个权重Const算子，分别对应为BN的offset，scale，mean和variance

Tensor offset\_weight\_tensor(desc\_weight\_1);

Tensor scale\_weight\_tensor(desc\_weight\_1);

Tensor mean\_weight\_tensor(desc\_weight\_1);

Tensor variance\_weight\_tensor(desc\_weight\_1);

uint32\_t weight\_1\_len = weight\_shape.GetShapeSize() \* sizeof(float);

//从bin文件中加载BN的offset，offset对应权重文件中的beta，表示输入偏置项

bool res = GetConstTensorFromBin(kPath+bn\_name+".beta.bin", offset\_weight\_tensor, weight\_1\_len);

if (!res) {

cout << \_\_LINE\_\_ << "GetConstTensorFromBin Failed!" << endl;

}

//从bin文件中加载BN的scale，scale对应权重文件中的gamma，表示输入Scalar

res = GetConstTensorFromBin(kPath+bn\_name+".gamma.bin", scale\_weight\_tensor, weight\_1\_len);

if (!res) {

cout << \_\_LINE\_\_ << "GetConstTensorFromBin Failed!" << endl;

}

//从bin文件中加载BN的moving\_mean，表输入的均值

res = GetConstTensorFromBin(kPath+bn\_name+".moving\_mean.bin", mean\_weight\_tensor, weight\_1\_len);

if (!res) {

cout << \_\_LINE\_\_ << "GetConstTensorFromBin Failed!" << endl;

}

//从bin文件中加载BN的moving\_variance，表输入的方差

res = GetConstTensorFromBin(kPath+bn\_name+".moving\_variance.bin", variance\_weight\_tensor, weight\_1\_len);

if (!res) {

cout << \_\_LINE\_\_ << "GetConstTensorFromBin Failed!" << endl;

}

//构造对应的常量算子，用来定义权重

auto bn\_offset = op::Const(bn\_name+"\_beta")

.set\_attr\_value(offset\_weight\_tensor);

auto bn\_scale = op::Const(bn\_name+"\_gamma")

.set\_attr\_value(scale\_weight\_tensor);

auto bn\_mean = op::Const(bn\_name+"\_mean")

.set\_attr\_value(mean\_weight\_tensor);

auto bn\_variance = op::Const(bn\_name+"\_variance")

.set\_attr\_value(variance\_weight\_tensor);

//构建bn算子

auto batchnorm = op::BatchNorm(bn\_name)

.set\_input\_x(data)

.set\_input\_offset(bn\_offset)

.set\_input\_scale(bn\_scale) //设置输入Scalar

.set\_input\_mean(bn\_mean) //设置输入均值

.set\_input\_variance(bn\_variance) //设置输入方差

.set\_attr\_data\_format("NCHW") //设置输入数据的格式NCHW

.set\_attr\_is\_training(false); //此时非训练状态，设置成false

TensorDesc batchnorm\_input\_desc\_x(ge::Shape(), FORMAT\_NCHW, DT\_FLOAT);

TensorDesc batchnorm\_output\_desc\_y(ge::Shape(), FORMAT\_NCHW, DT\_FLOAT);

//更新BN的输入信息

batchnorm.update\_input\_desc\_x(batchnorm\_input\_desc\_x);

batchnorm.update\_input\_desc\_scale(batchnorm\_input\_desc\_x);

batchnorm.update\_input\_desc\_offset(batchnorm\_input\_desc\_x);

batchnorm.update\_input\_desc\_mean(batchnorm\_input\_desc\_x);

batchnorm.update\_input\_desc\_variance(batchnorm\_input\_desc\_x);

//更新BN的输出信息

batchnorm.update\_output\_desc\_y(batchnorm\_output\_desc\_y);

batchnorm.update\_output\_desc\_batch\_mean(batchnorm\_output\_desc\_y);

batchnorm.update\_output\_desc\_batch\_variance(batchnorm\_output\_desc\_y);

}

1. **Flatten**

Flatten算子定义如下：

REG\_OP(Flatten)

.INPUT(x, TensorType({DT\_INT8, DT\_INT16, DT\_INT32, DT\_INT64,

DT\_UINT8, DT\_UINT16, DT\_UINT32, DT\_UINT64,

DT\_FLOAT, DT\_FLOAT16}))

.OUTPUT(y, TensorType({DT\_INT8, DT\_INT16, DT\_INT32, DT\_INT64,

DT\_UINT8, DT\_UINT16, DT\_UINT32, DT\_UINT64,

DT\_FLOAT, DT\_FLOAT16}))

.ATTR(axis, Int, 1)

.OP\_END\_FACTORY\_REG(Flatten)

Flatten算子的代码实现如下：

/\*

>> 生成Flatten算子

flatten\_name：Flatten算子的名称

data：上一个算子。进行构图的连接

\*/

Operator GenFlattenOp(string flatten\_name,Operator data){

//构建Flatten算子

auto flatten = op::FlattenV2(flatten\_name).set\_input\_x(data);

//更新算子输入输出信息

TensorDesc tensor\_desc(ge::Shape(), FORMAT\_ND, DT\_FLOAT);

flatten.update\_input\_desc\_x(tensor\_desc);

flatten.update\_output\_desc\_y(tensor\_desc);

}

1. **Dense**

Dense使用MatMulV2算子实现，MatMulV2算子包含了MatMul和BiasAdd两个部分：

REG\_OP(MatMulV2)

.INPUT(x1, TensorType({DT\_FLOAT, DT\_FLOAT16, DT\_INT32, DT\_INT8, DT\_INT4, DT\_BF16}))

.INPUT(x2, TensorType({DT\_FLOAT, DT\_FLOAT16, DT\_INT32, DT\_INT8, DT\_INT4, DT\_BF16}))

.OPTIONAL\_INPUT(bias, TensorType({DT\_FLOAT, DT\_FLOAT16, DT\_INT32, DT\_BF16}))

.OUTPUT(y, TensorType({DT\_FLOAT, DT\_FLOAT16, DT\_INT32, DT\_BF16}))

.OPTIONAL\_INPUT(offset\_w, TensorType({DT\_INT8, DT\_INT4}))

.ATTR(transpose\_x1, Bool, false)

.ATTR(transpose\_x2, Bool, false)

.ATTR(offset\_x, Int, 0)

.OP\_END\_FACTORY\_REG(MatMulV2)

说明：

* Dense层的偏置通过OPTIONAL\_INPUT的bias输入，bias的值来自bin权重文件
* 属性transpose\_x1，transpose\_x2，在dense层MatMul的计算逻辑是：MatMul的weight的shape为(out\_channels, in\_channels), 运算为，transpose\_x1和transpose\_x2分别对应和。

Dense算子的代码实现如下：

/\*

>> 生成Dense算子

input\_channel：输入通道数，MatMul的weight的shape为(out\_channels, in\_channels)

output\_channel：输出通道数

dense\_name：Dense算子的名称

data：上一个算子，即被flatten后的输入数据，作为MatMul运算中的X。进行构图的连接

\*/

Operator GenDenseOp(uint32\_t input\_channel,uint32\_t output\_channel,string dense\_name,Operator data){

// 构造dense层的权重矩阵，权重来自bin文件

auto matmul\_weight\_shape = ge::Shape({output\_channel, input\_channel});

TensorDesc desc\_matmul\_weight(matmul\_weight\_shape, FORMAT\_ND, DT\_FLOAT);

Tensor matmul\_weight\_tensor(desc\_matmul\_weight);

uint32\_t matmul\_weight\_len = matmul\_weight\_shape.GetShapeSize() \* sizeof(float);

bool res = GetConstTensorFromBin(kPath + dense\_name+".weight.bin", matmul\_weight\_tensor, matmul\_weight\_len);

if (!res) {

cout << \_\_LINE\_\_ << "GetConstTensorFromBin Failed!" << endl;

}

//构造matmul算子的权重常量算子

auto matmul\_weight = op::Const(dense\_name+"\_weight")

.set\_attr\_value(matmul\_weight\_tensor);

//构造偏重常量算子，读取偏置参数，作为OPTIONAL\_INPUT的bias输入

auto bias\_add\_shape = ge::Shape({ output\_channel });

TensorDesc desc\_bias\_add\_const(bias\_add\_shape, FORMAT\_ND, DT\_FLOAT);

Tensor bias\_add\_const\_tensor(desc\_bias\_add\_const);

uint32\_t bias\_add\_const\_len = bias\_add\_shape.GetShapeSize() \* sizeof(float);

res = GetConstTensorFromBin(kPath + dense\_name+".bias.bin", bias\_add\_const\_tensor, bias\_add\_const\_len);

if (!res) {

cout << \_\_LINE\_\_ << "GetConstTensorFromBin Failed!" << endl;

}

auto bias\_add\_const = op::Const(dense\_name+"\_bias")

.set\_attr\_value(bias\_add\_const\_tensor);

// 构造MatMulV2算子，三个输入，权重矩阵W，flatten后的输入数据X，偏置bias

auto matmul = op::MatMulV2(dense\_name+"\_matmul")

.set\_input\_x1(data)

.set\_input\_x2(matmul\_weight)

.set\_attr\_transpose\_x2(true)

.set\_input\_bias(bias\_add\_const);

// 更新算子描述信息

TensorDesc tensor\_desc\_matmul(ge::Shape(), FORMAT\_ND, DT\_FLOAT);

matmul.update\_input\_desc\_x1(tensor\_desc\_matmul);

matmul.update\_input\_desc\_x2(tensor\_desc\_matmul);

matmul.update\_input\_desc\_bias(tensor\_desc\_matmul);

matmul.update\_output\_desc\_y(tensor\_desc\_matmul);

}

1. **Softmax**

Softmax使用SoftmaxV2算子，算子定义如下：

REG\_OP(SoftmaxV2)

.INPUT(x, TensorType({DT\_DOUBLE, DT\_FLOAT16, DT\_FLOAT}))

.OUTPUT(y, TensorType({DT\_DOUBLE, DT\_FLOAT16, DT\_FLOAT}))

.ATTR(axes, ListInt, {-1})

.OP\_END\_FACTORY\_REG(SoftmaxV2)

SoftmaxV2算子的代码实现如下：

/\*

>> 生成SoftmaxV2算子

softmax\_name：Flatten算子的名称

data：上一个算子。进行构图的连接

\*/

Operator GenSoftmaxOp(string softmax\_name, Operator data){

auto softmax = op::SoftmaxV2(softmax\_name).set\_input\_x(data); //softmax默认axes为-1

return softmax;

}

定义参数变量

edict中存放的是模型训练和测试中所需要的各种参数配置。

代码：

cfg = edict({

'data\_path': 'flowers/flower\_photos\_train', #训练数据集路径

'test\_path':'flowers/flower\_photos\_train', #测试数据集路径

'data\_size': 3616,

'HEIGHT': 224, # 图片高度

'WIDTH': 224, # 图片宽度

'\_R\_MEAN': 123.68, # CIFAR10的均值

'\_G\_MEAN': 116.78,

'\_B\_MEAN': 103.94,

'\_R\_STD': 1, # 自定义的标准差

'\_G\_STD': 1,

'\_B\_STD':1,

'\_RESIZE\_SIDE\_MIN': 256, # 图像增强resize最小值

'\_RESIZE\_SIDE\_MAX': 512,

'batch\_size': 32, # 批次大小

'num\_class': 5, # 分类类别

'epoch\_size': 5, # 训练次数

'loss\_scale\_num':1024,

'prefix': 'resnet-ai', # 模型保存的名称

'directory': './model\_resnet', # 模型保存的路径

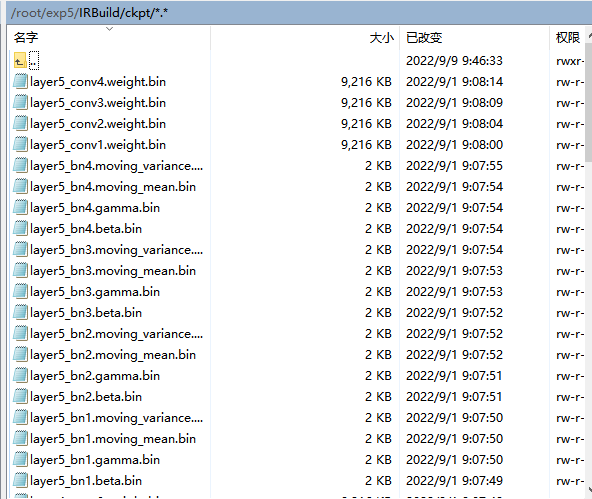
'save\_checkpoint\_steps': 10, # 每隔10步保存ckpt

})

构建Graph

1. 权重文件转bin文件

我们基于实验四得到flowervgg17的权重文件，权重文件已全部转换好，在(./ckpt)目录下，如下图所示。



1. 权重文件转bin文件

GetConstTensorFromBin函数实现：

/\*

>>读取bin文件构造tensor，并赋给常量算子Const

path：指定权重文件路径

weight：从权重文件中读取的Tensor类型的权重数据

len：指定权重数据大小

\*/

bool GetConstTensorFromBin(string path, Tensor &weight, uint32\_t len) {

//获得文件长度

ifstream in\_file(path.c\_str(), std::ios::in | std::ios::binary);

if (!in\_file.is\_open()) {

std::cout << "failed to open" << path.c\_str() << '\n';

return false;

}

in\_file.seekg(0, ios\_base::end); //让文件读指针指向文件结尾

istream::pos\_type file\_size = in\_file.tellg(); //可以用 tellg 函数获取文件读指针的位置，此位置即为文件长度

in\_file.seekg(0, ios\_base::beg); //让文件读指针指向文件开头

if (len != file\_size) {

cout << "Invalid Param.len:" << len << " is not equal with binary size（" << file\_size << ")\n";

in\_file.close();

return false;

}

char\* pdata = new(std::nothrow) char[len];

if (pdata == nullptr) {

cout << "Invalid Param.len:" << len << " is not equal with binary size（" << file\_size << ")\n";

in\_file.close();

return false;

}

//获取文件内容，并赋给Tensor

in\_file.read(reinterpret\_cast<char\*>(pdata), len);

auto status = weight.SetData(reinterpret\_cast<uint8\_t\*>(pdata), len);

if (status != ge::GRAPH\_SUCCESS) {

cout << "Set Tensor Data Failed"<< "\n";

in\_file.close();

return false;

}

in\_file.close();

return true;

}

1. 定义算子

创建算子实例，关于构建VGG17算子的函数均在5.4.1节中详细阐述，包括算子原型，根据原型设置算子的输入输出及属性。

1. 算子构图

基于以上定义好的算子，按照层级堆叠的方式构建VGG17的算子图。

//算子构图，这一部分需要学生实现

bool GenGraph(Graph& graph)

{

auto shape\_data = vector<int64\_t>({1,3,224,224});//输入数据[N,C,W,H],推理时batchsize为1

TensorDesc desc\_data(ge::Shape(shape\_data), FORMAT\_NCHW, DT\_FLOAT); //定义算子信息描述，shape传入desc\_data

// 实例化Data算子，名为data，作为数据输入的算子

auto data = op::Data("data");

data.update\_input\_desc\_x(desc\_data); //更新data1算子的输入数据信息描述，定义输入数据的shape，format和dtype

data.update\_output\_desc\_y(desc\_data); //更新data1算子的输出数据信息描述，定义输入数据的shape，format和dtype

auto layer1\_conv1 = GenConv2dOp(ge::Shape({64,3,3,3}),"layer1\_conv1",data); //卷积算子输入的shape是NCHW

auto layer1\_bn1 = GenBNOp(ge::Shape({64}),"layer1\_bn1",layer1\_conv1);

auto layer1\_relu1 = GenReluOp("layer1\_relu1",layer1\_bn1);2

auto layer1\_conv2 = GenConv2dOp(ge::Shape({64,64,3,3}),"layer1\_conv2",layer1\_relu1);

auto layer1\_bn2 = GenBNOp(ge::Shape({64}),"layer1\_bn2",layer1\_conv2);

auto layer1\_relu2 = GenReluOp("layer1\_relu2",layer1\_bn2);

auto layer1\_maxpool = GenMaxpoolOp("layer1\_maxpool",layer1\_relu2);

/\*

TO-DO：

1. 参考上述已经搭建好的输入算子和部分VGG17的算子构图部分，请按照VGG17的网络图，同学们自行完成构图的剩余部分的搭建

注意：要包含最后的softmax层

\*/

// 输入算子，名为data

std::vector<Operator> inputs{ data };

// 输出算子，名为pred

std::vector<Operator> outputs{pred};

graph.SetInputs(inputs).SetOutputs(outputs);

return true;

}

编译和运行Graph

* 定义模型

执行./scripts/.testcase300.sh 文件 ，成功编译可以得到以下结果

bash ./scripts/testcase\_300.sh

========== Test Start ==========

Ascend310

gen

========== Generate Graph1 Success! ==========

Build Model1 SUCCESS!

Save Offline Model1 SUCCESS!

run success

生成离线模型。

在out目录下生成离线模型文件ir\_build\_vgg17\_builtin.om。

模型推理

利用实验三，加载om离线模型，查看推理结果。

1. 在app.py中加载离线模型路径

if \_\_name\_\_ == '\_\_main\_\_':

# 加载编译的om离线模型

engine = ACLEngine('./out/ir\_build\_vgg17\_builtin.om')

engine.inference('./data/daisy\_demo.jpg')

engine.inference('./data/roses\_demo.jpg')

engine.inference('./data/dandelion\_demo.jpg')

engine.inference('./data/sunflowers\_demo.jpg')

engine.inference('./data/tulips\_demo.jpg')

engine.release\_resource()

1. 执行推理脚本

python3.7.5 app.py

注：

* 差异性1：实验三的图片预处理中对图片进行标准化，在实验四中并没有，所以将 ./process.py 中的以下两行注释。

# 将图片标准化

# image -= [125.307, 122.961, 113.8575]

# image /= [51.5865, 50.847, 51.255]

* 差异性2：实验三的模型是四分类("daisy", "roses", "sunflowers", "tulips")，实验四的模型是五分类("daisy", "dandelion", "roses", "sunflowers", "tulips")，所以将 ./app.py 中的以下两行进行修改

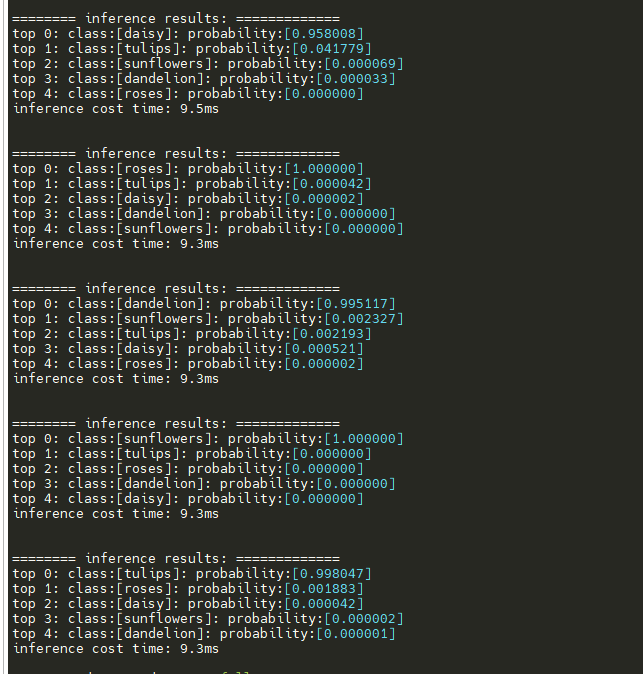
# line8

CLASSES = ("daisy", "dandelion", "roses", "sunflowers", "tulips")

# line60

tuple\_st = struct.unpack("5f", bytearray(inference\_result[0]))

得到正确的推理结果应如下图所示：



## 实验总结

本实验基于CANN高性能算子库，声明VGG17神经网络基本算子后加载所提供的的权重，接着构建VGG17网络模型，随后编译graph得到离线模型，并利用实验三的测试样例对构建的模型进行推理验证，得到推理结果。

本实验对基本算子已经提前声明，只需搭建VGG17神经网络并生成离线模型，利用实验三代码进行推理验证模型正确性。此实验重点是深入了解如何使用高性能算子搭建卷积神经网络。

在ATC模型转换的时候会进行算子融合。所谓算子融合即将多个算子的计算逻辑级联成一个算子的计算逻辑（如下图所示），可以节省数据从内存中搬入搬出的时间，优化执行效率，从而加速计算。试着下载并安装MindStudio，利用菜单栏中的 Ascend-> Model Visualizer 可视化构成的om模型。

## 实验任务与参考解答任务

### 实验任务

本实验基于CANN高性能算子库，声明VGG17神经网络基本算子后加载所提供的的权重，接着构建VGG17网络模型，随后编译graph得到离线模型，并利用实验三的测试样例对构建的模型进行推理验证，得到推理结果。

本实验对基本算子已经提前声明，只需搭建VGG17神经网络并生成离线模型，利用实验三代码进行推理验证模型正确性。此实验重点是深入了解如何使用高性能算子搭建卷积神经网络。在ATC模型转换的时候会进行算子融合。所谓算子融合即将多个算子的计算逻辑级联成一个算子的计算逻辑（如下图所示），可以节省数据从内存中搬入搬出的时间，优化执行效率，从而加速计算。同学可以同时查阅其他相关资料，了解算子融合。

### 参考答案

算子构图的部分代码完整版如下：

//算子构图，这一部分需要学生实现

bool GenGraph(Graph& graph)

{

auto shape\_data = vector<int64\_t>({1,3,224,224});//输入数据[N,C,W,H],推理时batchsize为1

TensorDesc desc\_data(ge::Shape(shape\_data), FORMAT\_NCHW, DT\_FLOAT); //定义算子信息描述，shape传入desc\_data

// 实例化Data算子，名为data，作为数据输入的算子

auto data = op::Data("data");

data.update\_input\_desc\_x(desc\_data); //更新data1算子的输入数据信息描述，定义输入数据的shape，format和dtype

data.update\_output\_desc\_y(desc\_data); //更新data1算子的输出数据信息描述，定义输入数据的shape，format和dtype

auto layer1\_conv1 = GenConv2dOp(ge::Shape({64,3,3,3}),"layer1\_conv1",data); //卷积算子输入的shape是NCHW

auto layer1\_bn1 = GenBNOp(ge::Shape({64}),"layer1\_bn1",layer1\_conv1);

auto layer1\_relu1 = GenReluOp("layer1\_relu1",layer1\_bn1);2

auto layer1\_conv2 = GenConv2dOp(ge::Shape({64,64,3,3}),"layer1\_conv2",layer1\_relu1);

auto layer1\_bn2 = GenBNOp(ge::Shape({64}),"layer1\_bn2",layer1\_conv2);

auto layer1\_relu2 = GenReluOp("layer1\_relu2",layer1\_bn2);

auto layer1\_maxpool = GenMaxpoolOp("layer1\_maxpool",layer1\_relu2);

/\*

TO-DO：

1. 参考上述已经搭建好的输入算子和部分VGG17的算子构图部分，请按照VGG17的网络图，同学们自行完成构图的剩余部分的搭建

注意：要包含最后的softmax层

\*/

auto layer2\_conv1 = GenConv2dOp(ge::Shape({128,64,3, 3}),"layer2\_conv1",layer1\_maxpool);

auto layer2\_bn1 = GenBNOp(ge::Shape({128}),"layer2\_bn1",layer2\_conv1);

auto layer2\_relu1 = GenReluOp("layer2\_relu1",layer2\_bn1);

auto layer2\_conv2 = GenConv2dOp(ge::Shape({128,128,3,3}),"layer2\_conv2",layer2\_relu1);

auto layer2\_bn2 = GenBNOp(ge::Shape({128}),"layer2\_bn2",layer2\_conv2);

auto layer2\_relu2 = GenReluOp("layer2\_relu2",layer2\_bn2);

auto layer2\_maxpool = GenMaxpoolOp("layer2\_maxpool",layer2\_relu2);

auto layer3\_conv1 = GenConv2dOp(ge::Shape({256,128,3, 3}),"layer3\_conv1",layer2\_maxpool);

auto layer3\_bn1 = GenBNOp(ge::Shape({256}),"layer3\_bn1",layer3\_conv1);

auto layer3\_relu1 = GenReluOp("layer3\_relu1",layer3\_bn1);

auto layer3\_conv2 = GenConv2dOp(ge::Shape({256,256,3, 3}),"layer3\_conv2",layer3\_relu1);

auto layer3\_bn2 = GenBNOp(ge::Shape({256}),"layer3\_bn2",layer3\_conv2);

auto layer3\_relu2 = GenReluOp("layer3\_relu2",layer3\_bn2);

auto layer3\_conv3 = GenConv2dOp(ge::Shape({256,256,3, 3}),"layer3\_conv3",layer3\_relu2);

auto layer3\_bn3 = GenBNOp(ge::Shape({256}),"layer3\_bn3",layer3\_conv3);

auto layer3\_relu3 = GenReluOp("layer3\_relu3",layer3\_bn3);

auto layer3\_maxpool = GenMaxpoolOp("layer3\_maxpool",layer3\_relu3);

auto layer4\_conv1 = GenConv2dOp(ge::Shape({512,256,3, 3,}),"layer4\_conv1",layer3\_maxpool);

auto layer4\_bn1 = GenBNOp(ge::Shape({512}),"layer4\_bn1",layer4\_conv1);

auto layer4\_relu1 = GenReluOp("layer4\_relu1",layer4\_bn1);

auto layer4\_conv2 = GenConv2dOp(ge::Shape({512, 512, 3, 3}),"layer4\_conv2",layer4\_relu1);

auto layer4\_bn2 = GenBNOp(ge::Shape({512}),"layer4\_bn2",layer4\_conv2);

auto layer4\_relu2 = GenReluOp("layer4\_relu2",layer4\_bn2);

auto layer4\_conv3 = GenConv2dOp(ge::Shape({512, 512, 3, 3}),"layer4\_conv3",layer4\_relu2);

auto layer4\_bn3 = GenBNOp(ge::Shape({512}),"layer4\_bn3",layer4\_conv3);

auto layer4\_relu3 = GenReluOp("layer4\_relu3",layer4\_bn3);

auto layer4\_maxpool = GenMaxpoolOp("layer4\_maxpool",layer4\_relu3);

auto layer5\_conv1 = GenConv2dOp(ge::Shape({512, 512, 3, 3}),"layer5\_conv1",layer4\_maxpool);

auto layer5\_bn1 = GenBNOp(ge::Shape({512}),"layer5\_bn1",layer5\_conv1);

auto layer5\_relu1 = GenReluOp("layer5\_relu1",layer5\_bn1);

auto layer5\_conv2 = GenConv2dOp(ge::Shape({512, 512, 3, 3}),"layer5\_conv2",layer5\_relu1);

auto layer5\_bn2 = GenBNOp(ge::Shape({512}),"layer5\_bn2",layer5\_conv2);

auto layer5\_relu2 = GenReluOp("layer5\_relu2",layer5\_bn2);

auto layer5\_conv3 = GenConv2dOp(ge::Shape({512, 512, 3, 3}),"layer5\_conv3",layer5\_relu2);

auto layer5\_bn3 = GenBNOp(ge::Shape({512}),"layer5\_bn3",layer5\_conv3);

auto layer5\_relu3 = GenReluOp("layer5\_relu3",layer5\_bn3);

auto layer5\_conv4 = GenConv2dOp(ge::Shape({512, 512, 3, 3}),"layer5\_conv4",layer5\_relu3);

auto layer5\_bn4 = GenBNOp(ge::Shape({512}),"layer5\_bn4",layer5\_conv4);

auto layer5\_relu4 = GenReluOp("layer5\_relu4",layer5\_bn4);

auto layer5\_maxpool = GenMaxpoolOp("layer5\_maxpool",layer5\_relu4);

//构建flatten算子

auto flatten = GenFlattenOp("flatten", layer5\_maxpool);

//构建dense算子

auto fullyconnect1 = GenDenseOp(512\*7\*7,4096,"fullyconnect1",flatten);

auto fullyconnect1\_relu1 = GenReluOp("fullyconnect1\_relu1",fullyconnect1);

auto fullyconnect2 = GenDenseOp(4096,4096,"fullyconnect2",fullyconnect1\_relu1);

auto fullyconnect2\_relu1 = GenReluOp("fullyconnect2\_relu1",fullyconnect2);

auto fullyconnect3 = GenDenseOp(4096,5,"fullyconnect3",fullyconnect2\_relu1);

auto pred = GenSoftmaxOp("softmax", fullyconnect3);

// 输入算子，名为data

std::vector<Operator> inputs{ data };

// 输出算子，名为pred

std::vector<Operator> outputs{pred};

graph.SetInputs(inputs).SetOutputs(outputs);

return true;

}