

AI 计算系统 实验操作手册

实验序号： 七

实验名称： 基于 MindSpore
推理的量化实验

教 师： 朱光辉

学 校： 南京大学

时 间： 2022 年 8 月 23 日

七 基于MindSpore推理的量化实验

7.1 实验目的

本实验的目的是了解神经网络量化操作，能够独立实现 `int8` 量化操作，构建量化 `VGG16` 神经网络，并基于MindSpore框架实现量化推理，能够独立编写量化操作代码。

7.2 背景介绍

7.2.1.量化

量化即以较低的推理精度损失将连续取值（或者大量可能的离散取值）的浮点型模型权重或流经模型的张量数据定点近似（通常为INT8）为有限多个（或较少的）离散值的过程，它是以更少位数的数据类型用于近似表示32位有限范围浮点型数据的过程，而模型的输入输出依然是浮点型。这样的好处是可以减小模型尺寸大小，减少模型内存占用，加快模型推理速度，降低功耗等。

如上所述，与FP32类型相比，FP16、INT8、INT4等低精度数据表达类型所占用空间更小。使用低精度数据表达类型替换高精度数据表达类型，可以大幅降低存储空间和传输时间。而低比特的计算性能也更高，INT8相对比FP32的加速比可达到3倍甚至更高，对于相同的计算，功耗上也有明显优势。

当前业界量化方案主要分为两种：感知量化训练（Quantization Aware Training）和训练后量化（Post-training Quantization）

7.2.2.训练后量化

对于已经训练好的 `float32` 模型，通过训练后量化将其转为 `int8`，不仅能减小模型大小，而且能显著提高推理性能。训练后量化分为两类：

1. 权重量化：对模型的权值进行量化，仅压缩模型大小，推理时仍然执行 `float32` 推理；
2. 全量化：对模型的权值、激活值等统一进行量化，推理时执行 `int` 运算，能提升模型推理速度、降低功耗。

7.2.3.矩阵运算的int8量化

r 代表浮点实数， q 代表量化后的定点整数。浮点数和整型之间的换算公式如下所示，其中 S 代表缩放系数，表示实数和整数之间的比例关系， $r_{max}, r_{min}, q_{max}, q_{min}$ 分布代表浮点实数及定点整数最大最小值， Z 代表实数中的0经过量化对应的整数值，为了在矩阵padding的时候保证浮点数值的0和定点整数的 Z 完全等价，保证定点和浮点之间的表征能够一致。

$$\begin{aligned}r &= S(q - Z) \\ q &= \text{round}(\frac{r}{S} + Z) \\ S &= \frac{r_{max} - r_{min}}{q_{max} - q_{min}} \\ Z &= \text{round}(q_{max} - \frac{r_{max}}{S})\end{aligned}$$

假设卷积的权重 $Weight$ 为 w ， $bias$ 为 b ，输入为 x ，输出的激活值为 a 。由于卷积本质上就是矩阵运算，因此表示成：

$$a = \sum_i^N w_i x_i + b$$

由此得到量化的公式

$$S_a(q_a - Z_a) = \sum_i^N S_w(q_w - Z_w)S_x(q_x - Z_x) + S_b(q_b - Z_b)$$
$$q_a = \frac{S_w S_x}{S_a} \sum_i^N (q_w - Z_w)(q_x - Z_x) + \frac{S_b}{S_a}(q_b - Z_b) + Z_a$$

这里面非整数部分就只有 $\frac{S_w S_x}{S_a}$ 、 $\frac{S_b}{S_a}$ ，因此接下来把这部分变成定点运算，对于 b 由于 $\sum_i^N S_w(q_w - Z_w)S_x(q_x - Z_x)$ 的结果通常会用 `int32` 的整数存储，因此 b 通常也量化到 `int32`，这里可以直接用 $S_w S_x$ 来代表 S_b ，由于 S_w 、 S_x 都是对应 8 个 bit 的缩放比例，因此 $S_w S_x$ 最多就放缩到 16 个 bit，用 32 bit 来存放 b 完全足够， Z_b 直接记为 0。因此公式调整为

$$q_a = \frac{S_w S_x}{S_a} \left(\sum_i^N (q_w - Z_w)(q_x - Z_x) + q_b \right) + Z_a$$
$$= M \left(\sum_i^N q_w q_x - \sum_i^N q_w Z_x - \sum_i^N q_x Z_w + \sum_i^N Z_w Z_x + q_b \right) + Z_a$$

其中 $M = \frac{S_w S_x}{S_a}$ ， M 通常为 $(0, 1)$ 之间的实数，因此可表示为 $M = 2^{-n} M_0$ ，其中 M_0 是一个定点实数，这样可以通过 M_0 的 bit 位移操作实现 $2^{-n} M_0$ ，这样整个过程就都在定点上计算了。

由于 Z_w 、 q_w 、 Z_x 、 q_b 都是可以事先计算的，因此 $\sum_i^N q_w Z_x$ 、 $\sum_i^N Z_w Z_x + q_b$ 也可以事先计算好，实际推理的时候只需要计算 $\sum_i^N q_w q_x$ 、 $\sum_i^N q_x Z_w$ 即可。

这里解释一下为什么可以用 $S_x S_w$ 来代替 S_b ， Z_b 可以直接记为 0：首先 S 和 Z 只是充当 r 和 q 之间转换的桥梁，只要保证 r 经过 S 、 Z 变换后得到 q ，而这个 q 经过 S 和 Z 可以反变换得到 r 即可。假设 $r \in [-1, 1]$ ， $q \in [0, 255]$ 那么 $S = \frac{2}{255}$ 、 $Z = 128$ ，一个 $[-1, 1]$ 区间的实数，完全可以通过 S 和 Z 可以换算到 $[0, 255]$ 区间的整数。如果对 q 的范围限制到 $[0, 100]$ ， r 和 q 依然可以转换，但会把信息都压缩到更小的 $[0, 100]$ 区间了。所以用 $S_x S_w$ 来代替 S_b ， Z_b 记为 0。假设所有的 $r \in [-1, 1]$ 那么 $S_x S_w = \frac{4}{2^{16}}$ ，而 b 是用 32 bit 存储的，本来 $S_b = \frac{2}{2^{32}}$ ，因此缩放系数砍掉了一半的信息量，会带来一定的精度损失，不过大部分情况下这点损失是可以忽略的。

7.3 实验环境

平台：Modelarts

操作系统：euler (aarch64)

软件环境：编程框架MindSpore1.7.1、异构计算架构CANN5.1，Python3.7.10

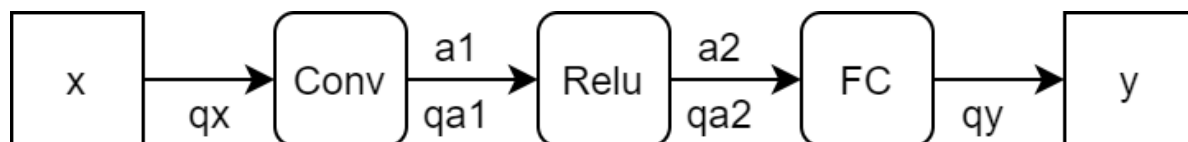
硬件环境：Ascend910A

测试数据集：花卉数据集（雏菊、蒲公英、玫瑰、向日葵、郁金香）

7.4 实验内容

7.5 实验步骤

对于一个简单的卷积网络， x 、 y 代表输入和输出， a_1 、 a_2 是网络中间的特征图， q_x 、 q_{a1} 、 q_{a2} 、 q_y 表示量化后的定点数，在后训练量化中，需要一些样本来统计 x 、 a_1 、 a_2 、 y 的数值范围，在根据量化位数以及量化方法来计算 S 和 Z 。



1. 首先输入部分样本图片进行正向传播，获取输入，输出及中间特征图的最大最小值。其中对于Relu、Maxpool算子来说，会沿用上一层输出的min和max，不需要额外统计。
2. 根据min和max以及量化的位数，计算 S 和 Z 。
3. 在量化推理的时候，会把输入 x 量化成定点整数 q_x ，然后进行卷积计算，得到输出 q_{a1} ，这个结果依然是整型的，然后继续计算Relu的输出 q_{a2} ，对于FC也是矩阵运算，得到输出 q_y ，根据计算出来的 S 和 Z 推算回浮点整数 y ，除了输入输出的量化与反量化操作，其他流程完全可以用定点运算来完成。

7.5.1 量化模块

首先实现基本的量化公式，具体代码如下

$$S = \frac{r_{max} - r_{min}}{q_{max} - q_{min}}$$

$$Z = round(q_{max} - \frac{r_{max}}{S})$$

```

def calculate_scale_zero_point(min_val, max_val, num_bits=8):
    qmin = 0.
    qmax = 2. ** num_bits - 1.
    scale = float((max_val - min_val) / (qmax - qmin)) # S=(rmax-rmin)/(qmax-qmin)
    zero_point = round(qmax - max_val / scale) # Z=round(qmax-rmax/scale)

    if zero_point < qmin:
        zero_point = qmin
    elif zero_point > qmax:
        zero_point = qmax

    return scale, zero_point

def quantize_tensor(x, scale, zero_point, num_bits=8, signed=False):
    # TODO 请根据公式实现张量的量化操作
    if signed:
        qmin = - 2. ** (num_bits - 1)
        qmax = 2. ** (num_bits - 1) - 1
    else:
        qmin = 0.
        qmax = 2. ** num_bits - 1.

    q_x = zero_point + x / scale
    q_x = (q_x.clip(qmin, qmax)).round() # q=round(r/S+Z)
    return q_x.astype(ms.float32) # 由于mindspore不支持int类型的运算，因此我们
    是用float来表示整数

def dequantize_tensor(q_x, scale, zero_point):

```

```
return scale * (q_x - zero_point) #  $r=S(q-Z)$ 
```

在量化过程中，需要先统计样本和中间层的最大最小值，同时也涉及到量化、反量化操作，因此将这些功能封装成一个QParam类

```
class QParam(object):
    def __init__(self, num_bits=8):
        self.num_bits = num_bits
        self.scale = None
        self.zero_point = None
        self.min = None
        self.max = None

    def update(self, tensor):
        # 用来统计 min、max
        if self.max is None or self.max < tensor.max():
            self.max = tensor.max()
        self.max = 0 if self.max < 0 else self.max

        if self.min is None or self.min > tensor.min():
            self.min = tensor.min()
        self.min = 0 if self.min > 0 else self.min

        self.scale, self.zero_point = calculate_scale_zero_point(self.min,
            self.max, self.num_bits)

    def quantize_tensor(self, tensor):
        return quantize_tensor(tensor, self.scale, self.zero_point,
            num_bits=self.num_bits)

    def dequantize_tensor(self, q_x):
        return dequantize_tensor(q_x, self.scale, self.zero_point)
```

接着定义基本的量化基类

- `__init__` 函数：指定量化的位数外，还需指定是否提供量化输入 (qi) 及输出参数 (qo)。在前面也提到，不是每一个网络模块都需要统计输入的 min、max，大部分中间层都是用上一层的 qo 来作为自己的 qi 的，另外有些中间层的激活函数也是直接用上一层的 qi 来作为自己的 qi 和 qo。
- `freeze` 函数：这个函数会在统计完 min、max 后发挥作用。正如上文所说的，公式 (4) 中有很多项是可以提前计算好的，freeze 就是把这些项提前固定下来，同时也将网络的权重由浮点实数转化为定点整数。
- `quantize_inference` 函数：这个函数主要是量化 inference 的时候会使用。

```

class QModule(nn.Cell):
    def __init__(self, qi=True, qo=True, num_bits=8):
        super(QModule, self).__init__()
        if qi:
            self.qi = QParam(num_bits=num_bits)
        if qo:
            self.qo = QParam(num_bits=num_bits)

    def freeze(self):
        pass

    def quantize_inference(self, x):
        raise NotImplementedError('quantize_inference should be implemented.')

```

7.5.2 量化卷积模块

量化卷积模块包括

- `__init__` 函数：需要传入 `conv_module` 模块，这个模块对应全精度的卷积层，另外的 `qw` 参数则是用来统计 weight 的 min、max 以及对 weight 进行量化用的。
- `freeze` 函数：这个函数主要就是计算公式中的 M 、 q_w 、 q_b ，其中 M 应该由移位来实现定点化加速，为了实现方便，在此用原始的数学操作进行代替
- `construct` 函数：这个函数和正常的 `construct` 一样，也是在 float 上进行的，只不过需要统计输入输出以及 weight 的 min、max 而已。其中这里需要对 weight 量化到 int8 然后又反量化回 float，这里其实就是所谓的伪量化节点，因为我们在实际量化 inference 的时候会把 weight 量化到 int8，这个过程本身是有精度损失的（来自四舍五入的 round 带来的截断误差），所以在统计 min、max 的时候，需要把这个过程带来的误差也模拟进去。
- `quantize_inference` 函数：这个函数在实际 inference 的时候会被调用。注意，这个函数里面的卷积操作是在 int 上进行的，这是量化推理加速的关键「当然，由于 mindspore 的限制，我们仍然是在 float 上计算，只不过数值都是整数。这也可以看出量化推理是跟底层实现紧密结合的技术」。

```

class QConv2d(QModule):
    def __init__(self, conv_module, qi=True, qo=True, num_bits=8):
        super(QConv2d, self).__init__(qi=qi, qo=qo, num_bits=num_bits)
        self.num_bits = num_bits
        self.conv_module = conv_module
        self.qw = QParam(num_bits=num_bits)
        self.M = None

    def freeze(self, qi=None, qo=None):
        if hasattr(self, 'qi') and qi is not None:
            raise ValueError('qi has been provided in init function.')
        if not hasattr(self, 'qi') and qi is None:
            raise ValueError('qi is not existed, should be provided.')

        if hasattr(self, 'qo') and qo is not None:
            raise ValueError('qo has been provided in init function.')
        if not hasattr(self, 'qo') and qo is None:
            raise ValueError('qo is not existed, should be provided.')

        if qi is not None:
            self.qi = qi
        if qo is not None:
            self.qo = qo

```

```

# TODO 请实现卷积模块权重参数量化
self.M = self.qw.scale * self.qi.scale / self.qo.scale

self.conv_module.weight =
self.qw.quantize_tensor(self.conv_module.weight)
self.conv_module.weight = self.conv_module.weight - self.qw.zero_point

self.conv_module.bias = quantize_tensor(self.conv_module.bias,
                                         scale=self.qi.scale *
self.qw.scale,
                                         zero_point=0, num_bits=32,
signed=True)

def construct(self, x):
    if hasattr(self, 'qi'):
        self.qi.update(x)
        x = self.qi.quantize_tensor(x)
        x = self.qi.dequantize_tensor(x)

    self.qw.update(self.conv_module.weight)
    self.conv_module.weight =
self.qw.quantize_tensor(self.conv_module.weight)
    self.conv_module.weight =
self.qw.dequantize_tensor(self.conv_module.weight)
    x = ops.conv2d(x, self.conv_module.weight,
stride=self.conv_module.stride, pad_mode=self.conv_module.pad_mode)
    if self.conv_module.bias is not None:
        x = ops.bias_add(x, self.conv_module.bias)

    if hasattr(self, 'qo'):
        self.qo.update(x)
        x = self.qo.quantize_tensor(x)
        x = self.qo.dequantize_tensor(x)
    return x

def quantize_inference(self, x):
    # TODO 请实现卷积模块量化推理模块
    x = x - self.qi.zero_point
    x = self.conv_module(x)
    x = self.M * x
    x = x.round()
    x = x + self.qo.zero_point
    x = x.clip(0., 2. ** self.num_bits - 1.).round()
    return x

```

7.5.3 量化全连接层模块

与量化卷积模块功能相似，这里不再叙述

```

class QDense(QModule):
    def __init__(self, fc_module, qi=True, qo=True, num_bits=8):
        super(QDense, self).__init__(qi=qi, qo=qo, num_bits=num_bits)
        self.num_bits = num_bits
        self.fc_module = fc_module
        self.qw = QParam(num_bits=num_bits)
        self.M = ms.Tensor([])

```

```

def freeze(self, qi=None, qo=None):
    if hasattr(self, 'qi') and qi is not None:
        raise ValueError('qi has been provided in init function.')
    if not hasattr(self, 'qi') and qi is None:
        raise ValueError('qi is not existed, should be provided.')

    if hasattr(self, 'qo') and qo is not None:
        raise ValueError('qo has been provided in init function.')
    if not hasattr(self, 'qo') and qo is None:
        raise ValueError('qo is not existed, should be provided.')

    if qi is not None:
        self.qi = qi
    if qo is not None:
        self.qo = qo

    # TODO 请实现全连接模块权重参数量化
    self.M = self.qw.scale * self.qi.scale / self.qo.scale

    self.fc_module.weight = self.qw.quantize_tensor(self.fc_module.weight)
    self.fc_module.weight = self.fc_module.weight.data - self.qw.zero_point
    self.fc_module.bias = quantize_tensor(self.fc_module.bias,
                                           scale=self.qi.scale *
self.qw.scale,
                                           zero_point=0,
                                           num_bits=32,
                                           signed=True)

def construct(self, x):
    if hasattr(self, 'qi'):
        self.qi.update(x)
        x = self.qi.quantize_tensor(x)
        x = self.qi.dequantize_tensor(x)

    self.qw.update(self.fc_module.weight)
    self.fc_module.weight = self.qw.quantize_tensor(self.fc_module.weight)
    self.fc_module.weight = self.qw.dequantize_tensor(self.fc_module.weight)
    x = ops.matmul(x, self.fc_module.weight.T)
    x = ops.bias_add(x, self.fc_module.bias)
    if hasattr(self, 'qo'):
        self.qo.update(x)
        x = self.qo.quantize_tensor(x)
        x = self.qo.dequantize_tensor(x)
    return x

def quantize_inference(self, x):
    # TODO 请实现全连接模块量化推理
    x = x - self.qi.zero_point
    x = self.fc_module(x)
    x = self.M * x
    x = x.round()
    x = x + self.qo.zero_point
    x = x.clip(0., 2. ** self.num_bits - 1.).round()
    return x

```


7.5.4 量化ReLU模块

大体内容与量化卷积模块相似，其中需要注意，在 `quantize_inference` 函数中，量化零点非真实的0，需要特别注意。

```
class QReLU(QModule):
    def __init__(self, qi=False, num_bits=None):
        super(QReLU, self).__init__(qi=qi, num_bits=num_bits)

    def freeze(self, qi=None):
        if hasattr(self, 'qi') and qi is not None:
            raise ValueError('qi has been provided in init function.')
        if not hasattr(self, 'qi') and qi is None:
            raise ValueError('qi is not existed, should be provided.')
        if qi is not None:
            self.qi = qi

    def construct(self, x):
        if hasattr(self, 'qi'):
            self.qi.update(x)
            x = self.qi.quantize_tensor(x)
            x = self.qi.dequantize_tensor(x)
        x = ops.relu(x)

        return x

    def quantize_inference(self, x):
        x[x < self.qi.zero_point] = self.qi.zero_point
        return x
```

7.5.5 量化最大池化模块

大体内容与量化卷积模块相似，在量化推理时，因为最大池化原理就是取区域最大值作为输出，故直接进行算子运算即可。

```
class QMaxPooling2d(QModule):
    def __init__(self, max_pool_module, qi=False, num_bits=None):
        super(QMaxPooling2d, self).__init__(qi=qi, num_bits=num_bits)
        self.max_pool_module = max_pool_module

    def freeze(self, qi=None):
        if hasattr(self, 'qi') and qi is not None:
            raise ValueError('qi has been provided in init function.')
        if not hasattr(self, 'qi') and qi is None:
            raise ValueError('qi is not existed, should be provided.')
        if qi is not None:
            self.qi = qi

    def construct(self, x):
        if hasattr(self, 'qi'):
            self.qi.update(x)
            x = self.qi.quantize_tensor(x)
            x = self.qi.dequantize_tensor(x)
        x = self.max_pool_module(x)

        return x
```

```
def quantize_inference(self, x):  
    return self.max_pool_module(x)
```

7.5.6 量化VGG网络

量化卷积模块包括

- `__init__` 函数：基本的算子定义
- `construct` 函数：网络正向传播模块
- `quantize` 函数：量化网络模块
- `quantize_forward` 函数：量化正向传播模块
- `freeze` 函数：量化参数冻结模块
- `quantize_inference` 函数：量化推理模块

```
import mindspore.nn as nn  
from quant_module import QConv2d, QMaxPooling2d, QDense, QReLU  
  
class Vgg(nn.Cell):  
    def __init__(self, num_classes=4):  
        super(Vgg, self).__init__()  
        self.layer1_conv1 = nn.Conv2d(in_channels=3, out_channels=64,  
kernel_size=3, has_bias=True)  
        self.layer1_relu1 = nn.ReLU()  
        self.layer1_conv2 = nn.Conv2d(in_channels=64, out_channels=64,  
kernel_size=3, has_bias=True)  
        self.layer1_relu2 = nn.ReLU()  
        self.layer1_maxpool = nn.MaxPool2d(kernel_size=2, stride=2)  
  
        self.layer2_conv1 = nn.Conv2d(in_channels=64, out_channels=128,  
kernel_size=3, has_bias=True)  
        self.layer2_relu1 = nn.ReLU()  
        self.layer2_conv2 = nn.Conv2d(in_channels=128, out_channels=128,  
kernel_size=3, has_bias=True)  
        self.layer2_relu2 = nn.ReLU()  
        self.layer2_maxpool = nn.MaxPool2d(kernel_size=2, stride=2)  
  
        self.layer3_conv1 = nn.Conv2d(in_channels=128, out_channels=256,  
kernel_size=3, has_bias=True)  
        self.layer3_relu1 = nn.ReLU()  
        self.layer3_conv2 = nn.Conv2d(in_channels=256, out_channels=256,  
kernel_size=3, has_bias=True)  
        self.layer3_relu2 = nn.ReLU()  
        self.layer3_conv3 = nn.Conv2d(in_channels=256, out_channels=256,  
kernel_size=3, has_bias=True)  
        self.layer3_relu3 = nn.ReLU()  
        self.layer3_maxpool = nn.MaxPool2d(kernel_size=2, stride=2)  
  
        self.layer4_conv1 = nn.Conv2d(in_channels=256, out_channels=512,  
kernel_size=3, has_bias=True)  
        self.layer4_relu1 = nn.ReLU()  
        self.layer4_conv2 = nn.Conv2d(in_channels=512, out_channels=512,  
kernel_size=3, has_bias=True)  
        self.layer4_relu2 = nn.ReLU()  
        self.layer4_conv3 = nn.Conv2d(in_channels=512, out_channels=512,  
kernel_size=3, has_bias=True)
```

```

        self.layer4_relu3 = nn.ReLU()
        self.layer4_maxpool = nn.MaxPool2d(kernel_size=2, stride=2)

        self.layer5_conv1 = nn.Conv2d(in_channels=512, out_channels=512,
kernel_size=3, has_bias=True)
        self.layer5_relu1 = nn.ReLU()
        self.layer5_conv2 = nn.Conv2d(in_channels=512, out_channels=512,
kernel_size=3, has_bias=True)
        self.layer5_relu2 = nn.ReLU()
        self.layer5_conv3 = nn.Conv2d(in_channels=512, out_channels=512,
kernel_size=3, has_bias=True)
        self.layer5_relu3 = nn.ReLU()
        self.layer5_maxpool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.flatten = nn.Flatten()

        self.fullyconnect1 = nn.Dense(512 * 7 * 7, 4096)
        self.relu_1 = nn.ReLU()
        self.fullyconnect2 = nn.Dense(4096, 4096)
        self.relu_2 = nn.ReLU()
        self.fullyconnect3 = nn.Dense(4096, num_classes)

def construct(self, x):
    x = self.layer1_conv1(x)
    x = self.layer1_relu1(x)
    x = self.layer1_conv2(x)
    x = self.layer1_relu2(x)
    x = self.layer1_maxpool(x)

    x = self.layer2_conv1(x)
    x = self.layer2_relu1(x)
    x = self.layer2_conv2(x)
    x = self.layer2_relu2(x)
    x = self.layer2_maxpool(x)

    x = self.layer3_conv1(x)
    x = self.layer3_relu1(x)
    x = self.layer3_conv2(x)
    x = self.layer3_relu2(x)
    x = self.layer3_conv3(x)
    x = self.layer3_relu3(x)
    x = self.layer3_maxpool(x)

    x = self.layer4_conv1(x)
    x = self.layer4_relu1(x)
    x = self.layer4_conv2(x)
    x = self.layer4_relu2(x)
    x = self.layer4_conv3(x)
    x = self.layer4_relu3(x)
    x = self.layer4_maxpool(x)

    x = self.layer5_conv1(x)
    x = self.layer5_relu1(x)
    x = self.layer5_conv2(x)
    x = self.layer5_relu2(x)
    x = self.layer5_conv3(x)
    x = self.layer5_relu3(x)
    x = self.layer5_maxpool(x)

```

```

        x = self.flatten(x)
        x = self.fullyconnect1(x)
        x = self.relu_1(x)
        x = self.fullyconnect2(x)
        x = self.relu_2(x)
        x = self.fullyconnect3(x)
        return x

    def quantize(self, num_bits=8):
        # 第一个卷积模块需要获取量化输入，其余模块会复用之前的量化输出
        self.qlayer1_conv1 = QConv2d(self.layer1_conv1, qi=True, qo=True,
num_bits=num_bits)
        self.qlayer1_relu1 = QReLU()
        self.qlayer1_conv2 = QConv2d(self.layer1_conv2, qi=False, qo=True,
num_bits=num_bits)
        self.qlayer1_relu2 = QReLU()
        self.qlayer1_maxpool2d = QMaxPooling2d(self.layer1_maxpool)

        self.qlayer2_conv1 = QConv2d(self.layer2_conv1, qi=False, qo=True,
num_bits=num_bits)
        self.qlayer2_relu1 = QReLU()
        self.qlayer2_conv2 = QConv2d(self.layer2_conv2, qi=False, qo=True,
num_bits=num_bits)
        self.qlayer2_relu2 = QReLU()
        self.qlayer2_maxpool2d = QMaxPooling2d(self.layer2_maxpool)

        self.qlayer3_conv1 = QConv2d(self.layer3_conv1, qi=False, qo=True,
num_bits=num_bits)
        self.qlayer3_relu1 = QReLU()
        self.qlayer3_conv2 = QConv2d(self.layer3_conv2, qi=False,
qo=True,num_bits=num_bits)
        self.qlayer3_relu2 = QReLU()
        self.qlayer3_conv3 = QConv2d(self.layer3_conv3, qi=False, qo=True,
num_bits=num_bits)
        self.qlayer3_relu3 = QReLU()
        self.qlayer3_maxpool2d = QMaxPooling2d(self.layer3_maxpool)

        self.qlayer4_conv1 = QConv2d(self.layer4_conv1, qi=False, qo=True,
num_bits=num_bits)
        self.qlayer4_relu1 = QReLU()
        self.qlayer4_conv2 = QConv2d(self.layer4_conv2, qi=False, qo=True,
num_bits=num_bits)
        self.qlayer4_relu2 = QReLU()
        self.qlayer4_conv3 = QConv2d(self.layer4_conv3, qi=False, qo=True,
num_bits=num_bits)
        self.qlayer4_relu3 = QReLU()
        self.qlayer4_maxpool2d = QMaxPooling2d(self.layer4_maxpool)

        self.qlayer5_conv1 = QConv2d(self.layer5_conv1, qi=False, qo=True,
num_bits=num_bits)
        self.qlayer5_relu1 = QReLU()
        self.qlayer5_conv2 = QConv2d(self.layer5_conv2, qi=False, qo=True,
num_bits=num_bits)
        self.qlayer5_relu2 = QReLU()
        self.qlayer5_conv3 = QConv2d(self.layer5_conv3, qi=False, qo=True,
num_bits=num_bits)
        self.qlayer5_relu3 = QReLU()
        self.qlayer5_maxpool2d = QMaxPooling2d(self.layer5_maxpool)

```

```

        self.qfc1 = QDense(self.fullyconnect1, qi=False, qo=True,
num_bits=num_bits)
        self.qfc1_relu = QReLU()
        self.qfc2 = QDense(self.fullyconnect2, qi=False, qo=True,
num_bits=num_bits)
        self.qfc2_relu = QReLU()
        self.qfc3 = QDense(self.fullyconnect3, qi=False, qo=True,
num_bits=num_bits)

def quantize_forward(self, x):
    x = self.qlayer1_conv1(x)
    x = self.qlayer1_relu1(x)
    x = self.qlayer1_conv2(x)
    x = self.qlayer1_relu2(x)
    x = self.qlayer1_maxpool2d(x)

    x = self.qlayer2_conv1(x)
    x = self.qlayer2_relu1(x)
    x = self.qlayer2_conv2(x)
    x = self.qlayer2_relu2(x)
    x = self.qlayer2_maxpool2d(x)

    x = self.qlayer3_conv1(x)
    x = self.qlayer3_relu1(x)
    x = self.qlayer3_conv2(x)
    x = self.qlayer3_relu2(x)
    x = self.qlayer3_conv3(x)
    x = self.qlayer3_relu3(x)
    x = self.qlayer3_maxpool2d(x)

    x = self.qlayer4_conv1(x)
    x = self.qlayer4_relu1(x)
    x = self.qlayer4_conv2(x)
    x = self.qlayer4_relu2(x)
    x = self.qlayer4_conv3(x)
    x = self.qlayer4_relu3(x)
    x = self.qlayer4_maxpool2d(x)

    x = self.qlayer5_conv1(x)
    x = self.qlayer5_relu1(x)
    x = self.qlayer5_conv2(x)
    x = self.qlayer5_relu2(x)
    x = self.qlayer5_conv3(x)
    x = self.qlayer5_relu3(x)
    x = self.qlayer5_maxpool2d(x)

    x = self.flatten(x)
    x = self.qfc1(x)
    x = self.qfc1_relu(x)
    x = self.qfc2(x)
    x = self.qfc2_relu(x)
    x = self.qfc3(x)
    return x

def freeze(self):
    # 冻结网络参数时，除第一个卷积模块不需要指定量化输入，其余模块都需要指定量化输入
    self.qlayer1_conv1.freeze()

```

```

self.qlayer1_relu1.freeze(qi=self.qlayer1_conv1.qo)
self.qlayer1_conv2.freeze(qi=self.qlayer1_conv1.qo)
self.qlayer1_relu2.freeze(qi=self.qlayer1_conv2.qo)
self.qlayer1_maxpool2d.freeze(qi=self.qlayer1_conv2.qo)

self.qlayer2_conv1.freeze(qi=self.qlayer1_conv2.qo)
self.qlayer2_relu1.freeze(qi=self.qlayer2_conv1.qo)
self.qlayer2_conv2.freeze(qi=self.qlayer2_conv1.qo)
self.qlayer2_relu2.freeze(qi=self.qlayer2_conv2.qo)
self.qlayer2_maxpool2d.freeze(qi=self.qlayer2_conv2.qo)

self.qlayer3_conv1.freeze(qi=self.qlayer2_conv2.qo)
self.qlayer3_relu1.freeze(qi=self.qlayer3_conv1.qo)
self.qlayer3_conv2.freeze(qi=self.qlayer3_conv1.qo)
self.qlayer3_relu2.freeze(qi=self.qlayer3_conv2.qo)
self.qlayer3_conv3.freeze(qi=self.qlayer3_conv2.qo)
self.qlayer3_relu3.freeze(qi=self.qlayer3_conv3.qo)
self.qlayer3_maxpool2d.freeze(qi=self.qlayer3_conv3.qo)

self.qlayer4_conv1.freeze(qi=self.qlayer3_conv3.qo)
self.qlayer4_relu1.freeze(qi=self.qlayer4_conv1.qo)
self.qlayer4_conv2.freeze(qi=self.qlayer4_conv1.qo)
self.qlayer4_relu2.freeze(qi=self.qlayer4_conv2.qo)
self.qlayer4_conv3.freeze(qi=self.qlayer4_conv2.qo)
self.qlayer4_relu3.freeze(qi=self.qlayer4_conv3.qo)
self.qlayer4_maxpool2d.freeze(qi=self.qlayer4_conv3.qo)

self.qlayer5_conv1.freeze(qi=self.qlayer4_conv3.qo)
self.qlayer5_relu1.freeze(qi=self.qlayer5_conv1.qo)
self.qlayer5_conv2.freeze(qi=self.qlayer5_conv1.qo)
self.qlayer5_relu2.freeze(qi=self.qlayer5_conv2.qo)
self.qlayer5_conv3.freeze(qi=self.qlayer5_conv2.qo)
self.qlayer5_relu3.freeze(qi=self.qlayer5_conv3.qo)
self.qlayer5_maxpool2d.freeze(qi=self.qlayer5_conv3.qo)

self.qfc1.freeze(qi=self.qlayer5_conv3.qo)
self.qfc1_relu.freeze(qi=self.qfc1.qo)
self.qfc2.freeze(qi=self.qfc1.qo)
self.qfc2_relu.freeze(qi=self.qfc2.qo)
self.qfc3.freeze(qi=self.qfc2.qo)

```

```

def quantize_inference(self, x):
    # 对输入x进行量化
    qx = self.qlayer1_conv1.qi.quantize_tensor(x)

    qx = self.qlayer1_conv1.quantize_inference(qx)
    qx = self.qlayer1_relu1.quantize_inference(qx)
    qx = self.qlayer1_conv2.quantize_inference(qx)
    qx = self.qlayer1_relu2.quantize_inference(qx)
    qx = self.qlayer1_maxpool2d.quantize_inference(qx)

    qx = self.qlayer2_conv1.quantize_inference(qx)
    qx = self.qlayer2_relu1.quantize_inference(qx)
    qx = self.qlayer2_conv2.quantize_inference(qx)
    qx = self.qlayer2_relu2.quantize_inference(qx)
    qx = self.qlayer2_maxpool2d.quantize_inference(qx)

    qx = self.qlayer3_conv1.quantize_inference(qx)

```

```

qx = self.qlayer3_relu1.quantize_inference(qx)
qx = self.qlayer3_conv2.quantize_inference(qx)
qx = self.qlayer3_relu2.quantize_inference(qx)
qx = self.qlayer3_conv3.quantize_inference(qx)
qx = self.qlayer3_relu3.quantize_inference(qx)
qx = self.qlayer3_maxpool2d.quantize_inference(qx)

qx = self.qlayer4_conv1.quantize_inference(qx)
qx = self.qlayer4_relu1.quantize_inference(qx)
qx = self.qlayer4_conv2.quantize_inference(qx)
qx = self.qlayer4_relu2.quantize_inference(qx)
qx = self.qlayer4_conv3.quantize_inference(qx)
qx = self.qlayer4_relu3.quantize_inference(qx)
qx = self.qlayer4_maxpool2d.quantize_inference(qx)

qx = self.qlayer5_conv1.quantize_inference(qx)
qx = self.qlayer5_relu1.quantize_inference(qx)
qx = self.qlayer5_conv2.quantize_inference(qx)
qx = self.qlayer5_relu2.quantize_inference(qx)
qx = self.qlayer5_conv3.quantize_inference(qx)
qx = self.qlayer5_relu3.quantize_inference(qx)
qx = self.qlayer5_maxpool2d.quantize_inference(qx)

qx = self.flatten(qx)

qx = self.qfc1.quantize_inference(qx)
qx = self.qfc1_relu.quantize_inference(qx)
qx = self.qfc2.quantize_inference(qx)
qx = self.qfc2_relu.quantize_inference(qx)
qx = self.qfc3.quantize_inference(qx)

# 对输出qx进行反量化
out = self.qfc3.qo.dequantize_tensor(qx)
return out

```

7.5.7 实验流程

第一步：初始化VGG网络并加载权重系数

第二步：构建对应推理数据

第三步：首先进行正常的网络推理，获取模型输出

第四步：构建量化模型，此实验为 int8 量化

第五步：进行量化推理，这里涉及到对中间特征图统计最大最小值

第六步：对网络量化参数进行固定

第七步：进行量化推理

```

import numpy as np
import cv2
import mindspore as ms
from mindspore import ops
from mindspore import load_checkpoint, load_param_into_net
from mindspore import context
from vgg import Vgg

```

```

context.set_context(mode=context.PYNATIVE_MODE, device_target='CPU')
np.set_printoptions(suppress=True)

def resize_image(image, target_size):
    h, w = image.shape[:2]
    th, tw = target_size
    # 获取等比缩放后的尺寸
    scale = min(th / h, tw / w)
    oh, ow = round(h * scale), round(w * scale)
    # 缩放图片, opencv缩放传入尺寸为(宽, 高), 这里采用线性差值算法
    image = cv2.resize(image, (ow, oh),
        interpolation=cv2.INTER_LINEAR).astype(np.uint8)
    # 将剩余部分进行填充
    new_image = np.ones((th, tw, 3), dtype=np.uint8) * 114
    new_image[:oh, :ow, :] = image
    return new_image

def process_image(img_path):
    # 读取图片, opencv读图后格式是BGR格式, 需要转为RGB格式
    image = cv2.imread(img_path, cv2.IMREAD_COLOR)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    # 将图片等比resize至(224x224)
    image = resize_image(image, (224, 224))
    image = np.array(image, dtype=np.float32)
    # 将图片标准化
    image -= [125.307, 122.961, 113.8575]
    image /= [51.5865, 50.847, 51.255]
    # (h,w,c) -> (c,h,w) -> (1,c,h,w)
    image = image.transpose((2, 0, 1))[None]
    return image

def direct_quantize(model, dataset):
    print('*'*50)
    print('Start quantize')
    for img_path, label in dataset:
        print("Start inference: {}".format(img_path))
        ndarray = process_image(img_path)
        tensor = ms.Tensor(ndarray, ms.float32)
        net_out = model.quantize_forward(tensor)
        prob = ops.Softmax()(net_out)
        print('Predict probability: {}'.format(np.around(prob.asnumpy(), 4)))
        predict_cls = (ops.Argmax()(prob)).asnumpy().item()
        print('Inference result: {}\n'.format(predict_cls == label))

def full_inference(model, dataset):
    print('*' * 50)
    print('Start full inference')
    for img_path, label in dataset:
        print("Start inference: {}".format(img_path))
        ndarray = process_image(img_path)
        tensor = ms.Tensor(ndarray, ms.float32)
        net_out = model(tensor)
        prob = ops.Softmax()(net_out)
        print('Predict probability: {}'.format(np.around(prob.asnumpy(), 4)))

```



```

        predict_cls = (ops.Argmax()(prob)).asnumpy().item()
        print('Inference result: {}\n'.format(predict_cls == label))

def quantize_inference(model, dataset):
    print('*' * 50)
    print('Start quantize inference')
    for img_path, label in dataset:
        print("Start inference: {}".format(img_path))
        ndarray = process_image(img_path)
        tensor = ms.Tensor(ndarray, ms.float32)
        net_out = model.quantize_inference(tensor)
        prob = ops.Softmax()(net_out)
        print('Predict probability: {}'.format(np.around(prob.asnumpy(), 4)))
        predict_cls = (ops.Argmax()(prob)).asnumpy().item()
        print('Inference result: {}\n'.format(predict_cls == label))

if __name__ == '__main__':
    # 初始化VGG网络并加载权重系数
    net = Vgg(num_classes=4)
    load_param_into_net(net, load_checkpoint('vgg.ckpt'), strict_load=True)
    net.set_train(False)

    # 构建对应推理数据
    dataset = [( './data/daisy_demo.jpg', 0),
                ( './data/roses_demo.jpg', 1),
                ( './data/sunflowers_demo.jpg', 2),
                ( './data/tulips_demo.jpg', 3)]

    # 首先进行正常的网络推理，获取模型输出
    full_inference(net, dataset)

    # 构建量化模型，此实验为int8量化
    net.quantize(num_bits=8)

    # 进行量化推理，这里涉及到对中间特征图统计最大最小值
    direct_quantize(net, dataset)

    # 对网络量化参数进行固定
    net.freeze()

    # 进行量化推理
    quantize_inference(net, dataset)

```

7.5.8 实验运行

代码目录介绍

```
EXP
|- data
|   |- daisy_demo.jpg          # 测试图片
|   |- roses_demo.jpg         # 测试图片
|   |- sunflowers_demo.jpg    # 测试图片
|   |- daisy_demo.jpg         # 测试图片
|- quant_inference.py         # 量化推理主函数
|- quant_module.py            # 量化模块
|- vgg.ckpt                   # VGG网络权重
|- vgg.py                     # VGG网络模块
```

1. 实现代码

```
vim quant_module.py
vim vgg.py
```

2. 运行实验

```
python3.7 quant_inference.py
```

7.6 评分指标

评估指标：

80：成功构建量化VGG网络，并输出正确结果

90：完成对量化VGG网络参数固定

100：完成对量化VGG网络进行量化推理，并输出正确结果

7.7 实验思考

在推理的时候，为了实现网络加速优化，通常将Convolution算子、BatchNormalization算子、Relu算子进行算子融合。若不进行融合，请思考如何实现BatchNormalization的量化模块。