AI 计算系统
实验操作手册

实验序号：二

实验名称：搭建 VGG16 神经网络实现图像分类

教　　师：朱光辉

学　　校：南京大学

时　　间：2022 年 8 月 23 日

二 搭建VGG16神经网络实现图像分类

2.1 实验目的

本实验的目的是掌握卷积神经网络的设计原理，能够独立构建卷积神经网络，深入了解基本算子的正向传播及反向传播原理，能够使用`Python`语言构建`VGG16`网络模型来对给定的输入图像进行分类，能够独立编写基本算子的正向传播及反向传播代码。具体包括：

- 深度了解卷积神经网络中卷积算子、最大池化算子等基本算子。
- 能够独立编写卷积算子的正向传播及反向传播代码、最大池化算子的正向传播及反向传播代码。深入了解正向传播、反向传播原理。了解链式求导法则。
- 能够利用`Python`语言实现`VGG16`网络的正向传播计算，加深对卷积神经网络结构的理解。

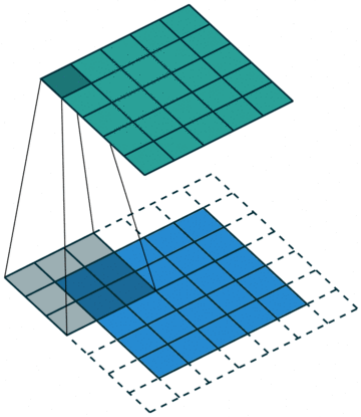
2.2 背景介绍

2.2.1 卷积神经网络里的基本算子

此实验介绍的是经典的卷积神经网络`VGG16`，该网络包含卷积算子(Convolution)、非线性变化算子(Relu)、最大池化算子(MaxPool)、扁平化算子(Flatten)、全连接算子(FullyConnect)。在实验一中已经详细介绍了ReLU算子和FullyConnect算子，这里介绍新增的算子：卷积算子，最大池化算子和扁平化算子。

2.2.1.1 卷积算子

卷积算子是卷积神经网络里最重要的一个算子，通过卷积计算，可以进一步扩大感受野，获得高纬度高层次的特征。卷积神经网络从广义上看就是一个个卷积算子堆叠而成。



对于基本的卷积算子，需要定义输入通道数C_{in}，输出通道数C_{out}，卷积核尺寸K，边界扩充大小P和卷积步长S。其中边界扩充大小是控制应用于输入张量的填充量，可以保证卷积后的输出张量与输入张量的形状保持一致。卷积算子中的参数包括卷积核的权重$Weight$和偏置$Bias$，其中权重$Weight$用四阶张量(C_{out}, C_{in}, K, K)表示，偏置$Bias$用一维向量$(C_{out},)$表示。一般运算流程如下所示：

1.对于输入张量$X(N, C_{in}, H_{in}, W_{in})$，一般会进行边界扩充至$X_{pad}(N, C_{in}, H_{pad}, W_{pad})$

$$X_{pad}(n,c,h,w) = \begin{cases} X(n,c,h-P,w-P) & P \le h < P+H_{in} \quad P \le w < P+W_{in} \\ 0 & other \end{cases}$$

2.进行卷积运算得到输出$Y(N, C_{out}, H_{out}, W_{out})$：

$$Y(N, C_{out}) = Bias(C_{out}) + \sum_{k=0}^{C_{in}-1} Weight(C_{out}, k) * X(N, k)$$

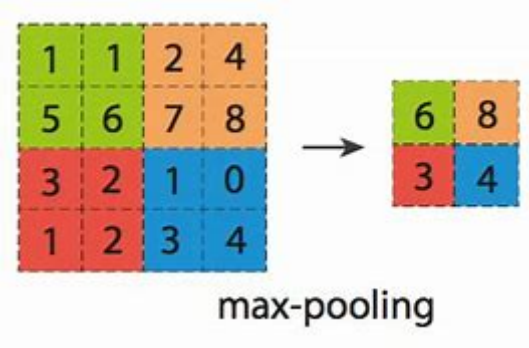
对于输出高度H_{out}、输出宽度W_{out}可根据以下公式计算得到：

$$H_{out} = \lfloor \frac{H_{in} + 2 * P - K}{S} + 1 \rfloor$$
$$W_{out} = \lfloor \frac{W_{in} + 2 * P - K}{S} + 1 \rfloor$$

2.2.1.2 最大池化算子

最大池化是从输入特征图中在每个通道上输出每个窗口的最大值。最大池化通常使用2 * 2的窗口且步幅为2，其目的是将特征图下采样2倍，为了在保留更重要特征的同时，降低分辨率，减少需要处理的特征图的元素个数。对于输入特征图$X(N, C, H_{in}, W_{in})$，经过最大池化算子（窗口大小K, 步长S)得到输出$Y(N, C, H_{out}, W_{out})$，其公式可表述为：

$$Y(N, C, h, w) = \max_{m=0,..K-1} \max_{n=0,..K-1} X(N, C, h * S + m, w * S + n)$$
$$H_{out} = \lfloor \frac{H_{in} - K}{S} + 1 \rfloor$$
$$W_{out} = \lfloor \frac{W_{in} - K}{S} + 1 \rfloor$$



2.2.1.3 扁平化算子

扁平化算子用于改变张量的形状，对输入的张量进行展平，即将四维张量转变成二维向量，常用于卷积层与全连接层之间，其公式如下所示：

$$X(N, C, H, W) = X(N, C * H * W)$$

2.2.2 VGG16网络结构

`VGG16`是经典的卷积神经网络结构，包含6个阶段，其中有13个卷积算子和3个全连接算子，其网络结构如下所示，其中每个卷积算子后都会有一个`Relu`算子。除最后一个全连接层之外，其余全连接层后也会有一个`Relu`算子，因为最后一层输出概率可直接判别类别，故不需要再做非线性变化。在验证时，一般会在最后一层加入softmax将输出值划分至0-1区间内，而在训练时，则不需要加入softmax。此实验在实现VGG16网络结构时并没有加入softmax这一层，故作说明。

算子	类型	输入通道数	输出通道数	窗口大小	边界扩充大小	步长	输出张量的高度和宽度
layer1_conv1	卷积	3	64	3	1	1	224x224
layer1_conv2	卷积	64	64	3	1	1	224x224
layer1_maxpool	最大池化	64	64	2	-	2	112x112
layer2_conv1	卷积	64	128	3	1	1	112x112
layer2_conv2	卷积	128	128	3	1	1	112x112
layer2_maxpool	最大池化	128	128	2	-	2	56x56
layer3_conv1	卷积	128	256	3	1	1	56x56
layer3_conv2	卷积	256	256	3	1	1	56x56
layer3_conv3	卷积	256	256	3	1	1	56x56
layer3_maxpool	最大池化	256	256	2	-	2	28x28
layer4_conv1	卷积	256	512	3	1	1	28x28
layer4_conv2	卷积	512	512	3	1	1	28x28
layer4_conv3	卷积	512	512	3	1	1	28x28
layer4_maxpool	最大池化	512	512	2	-	2	14x14
layer5_conv1	卷积	512	512	3	1	1	14x14
layer5_conv2	卷积	512	512	3	1	1	14x14
layer5_conv3	卷积	512	512	3	1	1	14x14
layer5_maxpool	最大池化	512	512	2	-	2	7x7
flatten	扁平化	-	-	-	-	-	-
fullyconnect1	全连接	25088	4096	-	-	-	-
fullyconnect2	全连接	4096	4096	-	-	-	-
fullyconnect3	全连接	4096	4	-	-	-	-

(224,224,3)

preprocess	(1,3,224,224)
Conv+Relu	(1,64,224,224)
Conv+Relu	(1,64,224,224)
MaxPool	(1,64,112,112)
Conv+Relu	(1,128,112,112)
Conv+Relu	(1,128,112,112)
MaxPool	(1,128,56,56)
Conv+Relu	(1,256,56,56)
Conv+Relu	(1,256,56,56)
Conv+Relu	(1,256,56,56)
MaxPool	(1,256,28,28)
Conv+Relu	(1,512,28,28)
Conv+Relu	(1,512,28,28)
Conv+Relu	(1,512,28,28)
MaxPool	(1,512,14,14)
Conv+Relu	(1,512,14,14)
Conv+Relu	(1,512,14,14)
Conv+Relu	(1,512,14,14)
MaxPool	(1,512,7,7)
Flatten	(1,512*7*7)
FullyConnect+Relu	(1,4096)
FullyConnect+Relu	(1,4096)
FullyConnect	(1,10)

2.3 实验环境

环境：x86_64CPU

操作系统：Ubuntu、CentOS

依赖	版本
python	3.7.5
numpy	1.19.4
scipy	1.5.4
opencv-python	4.5.3.56
numba	0.56.0

测试数据集：花卉图像数据集。该数据集包含3670张图像，共`雏菊、蒲公英、玫瑰、向日葵、郁金香`5个类别。本实验基于花卉图像数据集中`雏菊、玫瑰、向日葵、郁金香`4类图片训练后得到模型权重参数，不需要进行VGG16模型 的训练。

2.4 实验内容

本实验使用`VGG16`网络进行图像分类。首先搭建`VGG16`网络架构，然后利用已有的模型参数权重对给定的图像进行分类，之后给定指定标签，计算损失后进行反向传播，得到过程梯度。已有的参数模型是`VGG16`在基于花卉部分数据集训练获得的，`VGG16`输出结果对应花卉数据4个类别的概率（`雏菊、玫瑰、向日葵、郁金香`）。

本实验包括数据加载模块，基本算子模块，网络结构模块，网络推理模块，损失计算模块，反向传播模块。需要完成基本算子模块中卷积算子的正向传播和反向传播，最大池化算子的正向传播和反向传播模块，`VGG16`网络架构搭建。

2.5 实验步骤

2.5.1 数据加载模块

数据加载模块实现图片的读取以及预处理。本实验采用花卉部分数据集，该数据集下以`jpg`格式存储图片，且数据集下图片大小不统一，不能适配`VGG16`网络模型结构，故需对图片进行预处理。

第一步：通过`opencv-python`库读取图片，并转为RGB格式。

第二步：将图像等比缩放填充至`224x224x3`大小。

第三步：对输入图像进行标准化。具体公式如下：

$$image = \frac{image - mean}{std}$$

第四步：将标准化后的图像(H, W, C)转换为`VGG16`网络输入的统一形状，即(N, C, H, W)，此处为一张图片，因此$N = 1$，输入图片为RGB格式，故通道数$C = 3$。

```
#file: main.py
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，均值和方差数值为常用数值
    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
```

2.5.2 基本算子模块

本实验可直接使用实验一中已经实现的ReLU算子、FullyConnect算子。此实验重点实现Convolution算子和MaxPool算子的正向传播及反向传播算法。对于Flatten算子仅对张量进行形状变化，仅需了解相关原理即可。

2.5.2.1 卷积算子

卷积算子的实现如下所示，其中定义了以下成员函数：

- 算子初始化：需要定义卷积算子的超参数，包括输入张量的通道数C_{in}，输出张量的通道数C_{out}，卷积核的尺寸K，边界扩充大小P，卷积步长S。此外还需要定义输入张量的形状，用于反向传播。
- 权重初始化：卷积算子的参数包括权重和偏置。通常使用高斯随机数来初始化权重，将偏置值均设为0。
- 正向传播计算：根据公式进行卷积算子正向传播的计算，首先对输入张量`inputs`进行边界填充得到`inputs_pad`，在填充后的张量`inputs_pad`上滑动卷积窗口。
- 反向传播计算：根据公式进行卷积算子反向传播的计算（因为不涉及参数更新，故忽略计算偏置的梯度）。
- 参数加载：通过输入指定卷积算子的权重和偏置参数。

正向传播公式：

$$Y(n,c,h,w) = \sum_{k=0}^{C_{in}-1}\sum_{i=0}^{K-1}\sum_{j=0}^{K-1} Weight(c,k,i,j) * X_{pad}(n,k,h_s+i,w_s+j) + Bias(c)$$

$$n \in [0,N), c \in [0,C_{out}), h \in [0,H_{out}), w \in [0,W_{out})$$

$$h_s = h * S, w_s = w * S$$

反向传播公式：

$$\sum_{c_{in}=0}^{C_{in}-1}\sum_{i=0}^{K-1}\sum_{j=0}^{K-1} \nabla_{in_{pad}}(n,c_{in},h_s+i,w_s+i) = \sum_{c_{out}=0}^{C_{out}-1} \nabla_{out}(n,c_{out},h,w) * Weight(c_{out},c_{in},i,j)$$

$$n \in [0,N), h \in [0,H_{out}), w \in [0,W_{out})$$

$$h_s = h * S, w_s = w * S$$

$$\nabla in(n,c,h,w) = \nabla in_{pad}(n,c,P:P+h,P:P+w)$$

```
# file: layer.py
class ConvolutionLayer(object):
    def __init__(self, in_channels, out_channels, kernel_size, padding=0,
stride=1):
        # 输入通道数
        self.in_channels = in_channels
        # 输出通道数
        self.out_channels = out_channels
        # 卷积核尺寸
        self.kernel_size = kernel_size
        # 步长
        self.stride = stride
        # 填充长度
        self.padding = padding
```

```
        # 卷积核权重
        self.weight = np.random.normal(loc=0.0, scale=0.01,
                                       size=(self.out_channels,
self.in_channels,
                                             self.kernel_size,
self.kernel_size))
        # 卷积核偏置
        self.bias = np.zeros([self.out_channels])

        # 输入张量的形状，用于反向传播
        self.input_shape = None

    def forward(self, inputs):
        # 记录输入张量的形状，inputs: (N,C,H,W)
        self.input_shape = inputs.shape
        batch, channel, height, width = inputs.shape

        # 获取输入张量填充后的宽高
        pad_height = height + self.padding * 2
        pad_width = width + self.padding * 2

        # 将输入张量进行填充
        inputs_pad = np.zeros((batch, channel, pad_height, pad_width),
dtype=inputs.dtype)
        inputs_pad[:, :, self.padding:height + self.padding, self.padding:width
+ self.padding] = inputs

        # 获取输出张量的宽高，并构建输出张量
        out_height = int((pad_height - self.kernel_size) / self.stride + 1)
        out_width = int((pad_width - self.kernel_size) / self.stride + 1)
        outputs = np.zeros((batch, self.out_channels, out_height, out_width),
dtype=inputs.dtype)

        # 正向传播
        outputs = self._conv(inputs_pad, outputs, self.weight, self.bias,
self.kernel_size, self.stride)
        return outputs

    def backward(self, out_grad):
        # 获得输入张量，填充后输入张量，输出张量的形状
        batch, channel, height, width = self.input_shape
        _, out_channel, out_height, out_width = out_grad.shape
        pad_height = height + self.padding * 2
        pad_width = width + self.padding * 2

        # 构建填充输入张量的梯度
        in_grad = np.zeros((batch, channel, pad_height, pad_width))

        # 反向传播
        in_grad = self._conv_back(out_grad, in_grad, self.weight,
self.kernel_size, self.stride)

        # 返回输入张量梯度
        in_grad = in_grad[:, :, self.padding:height + self.padding,
self.padding:width + self.padding]
        return in_grad
```

```
    def load_params(self, weight, bias):
        assert self.weight.shape == weight.shape
        assert self.bias.shape == bias.shape
        self.weight = weight
        self.bias = bias

    @staticmethod
    @jit(nopython=True)      # 可以将python函数编译为机器代码的JIT编译器，可以极大的加速for循环的运行速度
    def _conv(inputs_pad, outputs, weight, bias, kernel_size, stride):
        # TODO: 根据公式编写下列代码 请用for循环实现
        in_channels = inputs_pad.shape[1]
        batch, out_channels, out_height, out_width = outputs.shape
        for n in range(batch):
            for c in range(out_channels):
                for h in range(out_height):
                    for w in range(out_width):
                        hs, ws = h * stride, w * stride
                        val = 0
                        for k in range(in_channels):
                            for i in range(kernel_size):
                                for j in range(kernel_size):
                                    val += weight[c, k, i, j] * inputs_pad[n, k, hs+i, ws+j]
                        val += bias[c]
                        outputs[n, c, h, w] = val
        return outputs

    @staticmethod
    @jit(nopython=True)
    def _conv_back(out_grad, in_grad, weight, kernel_size, stride):
        # TODO: 根据公式编写下列代码 请用for循环实现
        in_channels = in_grad.shape[1]
        batch, out_channel, out_height, out_width = out_grad.shape
        for n in range(batch):
            for h in range(out_height):
                for w in range(out_width):
                    hs, ws = h * stride, w * stride
                    for c_in in range(in_channels):
                        for i in range(kernel_size):
                            for j in range(kernel_size):
                                val = 0
                                for c_out in range(out_channel):
                                    val += out_grad[n, c_out, h, w] * weight[c_out, c_in, i, j]
                                in_grad[n, c_in, hs + i, ws + j] += val
        return in_grad
```

2.5.2.2 最大池化算子

最大池化算子的实现如下所示，其中定义了以下成员函数：

- 算子初始化：需要定义最大池化算子的超参数，包括池化核的尺寸K，池化步长S。此外初始化了用于反向传播的池化索引，输入张量的形状和输出张量的形状。
- 正向传播计算：根据公式进行池化算子正向传播的计算。

- 反向传播计算：根据公式进行池化算子反向传播的计算。在正向传播时，已经记录了池化索引，在反向传播时，只需将池化索引映射回输入张量的位置，将梯度带过去即可，其余位置置为0。

正向传播公式：

$$Y(n,c,h,w) = \max_{m=0,..K-1} \max_{n=0,..K-1} X(n,c,h_s+m,w_s+n)$$
$$n \in [0,N), c \in [0,C), h \in [0,H_{out}), w \in [0,W_{out})$$
$$h_s = h * S, w_s = w * S$$

反向传播公式：

$$\nabla_{in}(n,c,h_s:h_s+K,w_s:w_s+K)[i_{index},j_{index}] = \nabla_{out}(n,c,h,w)$$
$$n \in [0,N), c \in [0,C), h \in [0,H_{out}), w \in [0,W_{out})$$
$$h_s = h * S, w_s = w * S$$

其中 i_{index} 和 j_{index} 在正向传播中记录下的索引值。

```
# file: layer.py
class MaxPoolLayer(object):
    def __init__(self, kernel_size=2, stride=2):
        # 池化核大小
        self.kernel_size = kernel_size
        # 步长
        self.stride = stride
        # 池化索引，用于反向传播
        self.argidx = None
        # 输入张量形状
        self.input_shape = None
        # 输出张量形状
        self.output_shape = None

    def forward(self, inputs):
        # inputs: (N,C,H,W)
        batch, channel, height, width = inputs.shape

        # 获取输出张量的宽高，并构建输出张量
        out_height = int((height - self.kernel_size) / self.stride + 1)
        out_width = int((width - self.kernel_size) / self.stride + 1)
        outputs = np.zeros((batch, channel, out_height, out_width),
dtype=inputs.dtype)

        # 记录输入张量和输出张量的形状，并初始化池化索引
        self.input_shape = inputs.shape
        self.output_shape = outputs.shape
        self.argidx = np.zeros_like(outputs, dtype=np.int32)

        # 正向传播
        outputs, self.argidx = self._pool(outputs, inputs, self.argidx,
self.kernel_size, self.stride)
        return outputs

    def backward(self, out_grad):
        # 构建输入梯度
        in_grad = np.zeros(self.input_shape)
```

```
        # 反向传播
        in_grad = self._pool_back(out_grad, in_grad , self.argidx,
self.kernel_size, self.stride)
        return in_grad

    @staticmethod
    @jit(nopython=True)
    def _pool(outputs, inputs, argidx, kernel_size, stride):
        # TODO: 根据公式编写下列代码 请用for循环实现
        batch, channel, out_height, out_width = outputs.shape
        for n in range(batch):
            for c in range(channel):
                for h in range(out_height):
                    for w in range(out_width):
                        hs, ws = h*stride, w*stride
                        vector = inputs[n, c, hs:hs+kernel_size,
ws:ws+kernel_size]
                        max_value = vector[0][0]
                        for i in range(kernel_size):
                            for j in range(kernel_size):
                                if vector[i, j] > max_value:
                                    max_value = vector[i, j]
                                    # 记录当前索引
                                    argidx[n, c, h, w] = i * kernel_size + j
                        outputs[n, c, h, w] = max_value
        return outputs, argidx

    @staticmethod
    @jit(nopython=True)
    def _pool_back(out_grad, in_grad, argidx, kernel_size, stride):
        # TODO: 根据公式编写下列代码 请用for循环实现
        batch, channel, out_height, out_width = out_grad.shape
        for n in range(batch):
            for c in range(channel):
                for h in range(out_height):
                    for w in range(out_width):
                        hs, ws = h*stride, w*stride
                        # 将索引逆向转换至卷积核位置
                        i = argidx[n, c, h, w] // kernel_size
                        j = argidx[n, c, h, w] % kernel_size
                        in_grad[n, c, hs: hs+kernel_size, ws: ws+kernel_size][i,
j] = out_grad[n, c, h, w]

        return in_grad
```

2.5.2.3 扁平化算子

扁平化算子的实现如下所示，其中定义了以下成员函数：

- 正向传播计算：进行Flatten算子正向传播的计算。将四维张量(N, C, H, W)，扁平化至二维$(N, C * H * W)$
- 反向传播计算：进行Flatten算子反向传播的计算。将二维梯度$(N, C * H * W)$映射回四维梯度(N, C, H, W)即可。

```
# file: layer.py
class FlattenLayer(object):
    def __init__(self):
        self.input_shape = None

    def forward(self, inputs):
        # inputs: (N,C,H,W) -> (N, C*H*W)
        self.input_shape = inputs.shape
        batch, channel, height, width = inputs.shape
        return inputs.reshape((batch, channel*height*width))

    def backward(self, grad):
        return grad.reshape(self.input_shape)
```

2.5.3 VGG16网络结构模块

```
# file: vgg.py
from layer import ConvolutionLayer, ReluLayer, MaxPoolLayer, FullyConnectLayer,
FlattenLayer


class VGG16(object):
    def __init__(self, num_classes=4):
        # TODO 根据网络图搭建VGG16模型
        self.layer1_conv1 = ConvolutionLayer(in_channels=3, out_channels=64,
kernel_size=3, padding=1)
        self.layer1_relu1 = ReluLayer()
        self.layer1_conv2 = ConvolutionLayer(in_channels=64, out_channels=64,
kernel_size=3, padding=1)
        self.layer1_relu2 = ReluLayer()
        self.layer1_maxpool = MaxPoolLayer(kernel_size=2, stride=2)

        self.layer2_conv1 = ConvolutionLayer(in_channels=64, out_channels=128,
kernel_size=3, padding=1)
        self.layer2_relu1 = ReluLayer()
        self.layer2_conv2 = ConvolutionLayer(in_channels=128, out_channels=128,
kernel_size=3, padding=1)
        self.layer2_relu2 = ReluLayer()
        self.layer2_maxpool = MaxPoolLayer(kernel_size=2, stride=2)

        self.layer3_conv1 = ConvolutionLayer(in_channels=128, out_channels=256,
kernel_size=3, padding=1)
        self.layer3_relu1 = ReluLayer()
        self.layer3_conv2 = ConvolutionLayer(in_channels=256, out_channels=256,
kernel_size=3, padding=1)
        self.layer3_relu2 = ReluLayer()
        self.layer3_conv3 = ConvolutionLayer(in_channels=256, out_channels=256,
kernel_size=3, padding=1)
        self.layer3_relu3 = ReluLayer()
        self.layer3_maxpool = MaxPoolLayer(kernel_size=2, stride=2)

        self.layer4_conv1 = ConvolutionLayer(in_channels=256, out_channels=512,
kernel_size=3, padding=1)
        self.layer4_relu1 = ReluLayer()
```

```
        self.layer4_conv2 = ConvolutionLayer(in_channels=512, out_channels=512,
kernel_size=3, padding=1)
        self.layer4_relu2 = ReluLayer()
        self.layer4_conv3 = ConvolutionLayer(in_channels=512, out_channels=512,
kernel_size=3, padding=1)
        self.layer4_relu3 = ReluLayer()
        self.layer4_maxpool = MaxPoolLayer(kernel_size=2, stride=2)

        self.layer5_conv1 = ConvolutionLayer(in_channels=512, out_channels=512,
kernel_size=3, padding=1)
        self.layer5_relu1 = ReluLayer()
        self.layer5_conv2 = ConvolutionLayer(in_channels=512, out_channels=512,
kernel_size=3, padding=1)
        self.layer5_relu2 = ReluLayer()
        self.layer5_conv3 = ConvolutionLayer(in_channels=512, out_channels=512,
kernel_size=3, padding=1)
        self.layer5_relu3 = ReluLayer()
        self.layer5_maxpool = MaxPoolLayer(kernel_size=2, stride=2)

        self.flatten = FlattenLayer()
        self.fullyconnect1 = FullyConnectLayer(in_features=512 * 7 * 7,
out_features=4096)
        self.relu_1 = ReluLayer()
        self.fullyconnect2 = FullyConnectLayer(in_features=4096,
out_features=4096)
        self.relu_2 = ReluLayer()
        self.fullyconnect3 = FullyConnectLayer(in_features=4096,
out_features=num_classes)

        self.graph_layers = None
        self.create_graph()

    def create_graph(self):
        self.graph_layers = {
            'layer1_conv1': self.layer1_conv1, 'layer1_relu1':
self.layer1_relu1,
            'layer1_conv2': self.layer1_conv2, 'layer1_relu2':
self.layer1_relu2,
            'layer1_maxpool': self.layer1_maxpool,

            'layer2_conv1': self.layer2_conv1, 'layer2_relu1':
self.layer2_relu1,
            'layer2_conv2': self.layer2_conv2, 'layer2_relu2':
self.layer2_relu2,
            'layer2_maxpool': self.layer2_maxpool,

            'layer3_conv1': self.layer3_conv1, 'layer3_relu1':
self.layer3_relu1,
            'layer3_conv2': self.layer3_conv2, 'layer3_relu2':
self.layer3_relu2,
            'layer3_conv3': self.layer3_conv3, 'layer3_relu3':
self.layer3_relu3,
            'layer3_maxpool': self.layer3_maxpool,

            'layer4_conv1': self.layer4_conv1, 'layer4_relu1':
self.layer4_relu1,
            'layer4_conv2': self.layer4_conv2, 'layer4_relu2':
self.layer4_relu2,
```

```
            'layer4_conv3': self.layer4_conv3, 'layer4_relu3':
self.layer4_relu3,
            'layer4_maxpool': self.layer4_maxpool,

            'layer5_conv1': self.layer5_conv1, 'layer5_relu1':
self.layer5_relu1,
            'layer5_conv2': self.layer5_conv2, 'layer5_relu2':
self.layer5_relu2,
            'layer5_conv3': self.layer5_conv3, 'layer5_relu3':
self.layer5_relu3,
            'layer5_maxpool': self.layer5_maxpool,

            'flatten': self.flatten,
            'fullyconnect1': self.fullyconnect1, 'relu1': self.relu_1,
            'fullyconnect2': self.fullyconnect2, 'relu2': self.relu_2,
            'fullyconnect3': self.fullyconnect3,
        }

    def forward(self, x):
        for name in self.graph_layers.keys():
            print(f'forward: {name}: {x.mean()} {x.sum()}')
            x = self.graph_layers[name].forward(x)
        return x

    def backward(self, grad):
        for name in reversed(list(self.graph_layers.keys())):
            print(f'backward: {name}: {grad.mean()} {grad.sum()}')
            grad = self.graph_layers[name].backward(grad)
        return grad

    def resume_weights(self, ckpt):
        for name, params in ckpt.items():
            self.graph_layers[name].load_params(params['weight'],
params['bias'])
        print('reloaded success')
```

2.5.4 实验流程

```
# file: main.py
if __name__ == "__main__":
    import time
    # 分类类别
    CLASSES = ('daisy', 'roses', 'sunflowers', 'tulips')

    # 网络初始化、加载权重参数
    model = VGG16(4)
    ckpt = np.load('./file/vgg16_ckpt.npy', allow_pickle=True).item()
    model.resume_weights(ckpt)

    start_time = time.time()

    # 输入图片预处理
    image_path = './file/tulips_demo.jpg'
    tensor = process_image(image_path)
```

```
    # 模型正向传播
    outputs = model.forward(tensor)
    print(outputs)
    pred = int(np.argmax(outputs))
    print(CLASSES[pred])

    # 计算loss
    label = np.array([1, ])
    loss_func = CrossEntropy()
    loss = loss_func.forward(outputs, label)
    print(loss)

    # 反向传播
    grad = loss_func.backward()
    grad = model.backward(grad)
    print(grad.mean(), grad.sum())

    end_time = time.time()
    print(end_time - start_time)
```

2.5.5 实验运行

代码目录介绍

```
EXP
|-- file
|   |-- tulips_demo.jpg      //测试图片
|   |-- vgg16_ckpt.npy       //预训练权重
|-- layer.py                 //算子实现代码
|-- main.py                  //主函数代码
|-- vgg.py                   //vgg网络代码
```

1.实现代码

```
vim layer.py
vim vgg.py
```

2.运行实验

```
python3.7 main.py
```

2.6 评分指标

评估指标：

60：成功搭建VGG16网络并且成功加载参数

80：对于指定图片，VGG16网络正向传播可输出指定输输出，且对目标图片分类成功

100：VGG16可正确反向传播，输出指定梯度

2.7 实验思考

如何计算卷积神经网络的参数量和计算量?

在现有实现的基础上，考虑如何进一步提升卷积算子和池化算子计算性能?