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

[2 搭建VGG16神经网络实现图像分类 4](#_Toc2060443392)

[2.1实验介绍 4](#_Toc1698281469)

[2.1.1 卷积算子介绍 4](#_Toc811497206)

[2.1.2 最大池化算子介绍 5](#_Toc164899145)

[2.1.3 扁平化算子介绍 5](#_Toc1206025385)

[2.1.4 VGG16神经网络介绍 6](#_Toc1717985309)

[2.1.5 实验目录结构介绍 6](#_Toc1261454448)

[2.2 实验环境要求 6](#_Toc1306344352)

[2.3 实验详细设计与实现 7](#_Toc2004200783)

[2.3.1 实验步骤 7](#_Toc1321556686)

[2.4 实验总结 22](#_Toc2127344328)

[2.5 实验任务与参考解答任务 22](#_Toc820881793)

[2.5.1 实验任务 22](#_Toc1125346623)

[2.5.2 参考答案 23](#_Toc812213632)

# 搭建VGG16神经网络实现图像分类

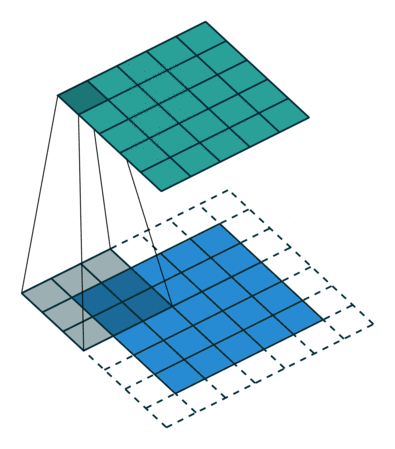
## 2.1实验介绍

本实验实现的是用Python语言构建VGG卷积神经网络。整体流程如下：

* 搭建卷积算子、最大池化算子等基本算子。
* 搭建VGG16卷积神经网络
* 定义数据预处理模块。
* 对单张图片进行正向传播和反向传播，输出正确的类别置信度，并查看中间梯度信息。

### 卷积算子介绍

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，输出通道数，卷积核尺寸，边界扩充大小 和卷积步长。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和偏置，其中权重用四阶张量表示，偏置用一维向量表示。一般运算流程如下所示：

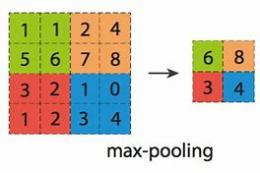
1.对于输入张量，一般会进行边界扩充至

2.进行卷积运算得到输出：

对于输出高度、输出宽度可根据以下公式计算得到：

### 最大池化算子介绍

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，经过最大池化算子（窗口大小, 步长)得到输出，其公式可表述为：

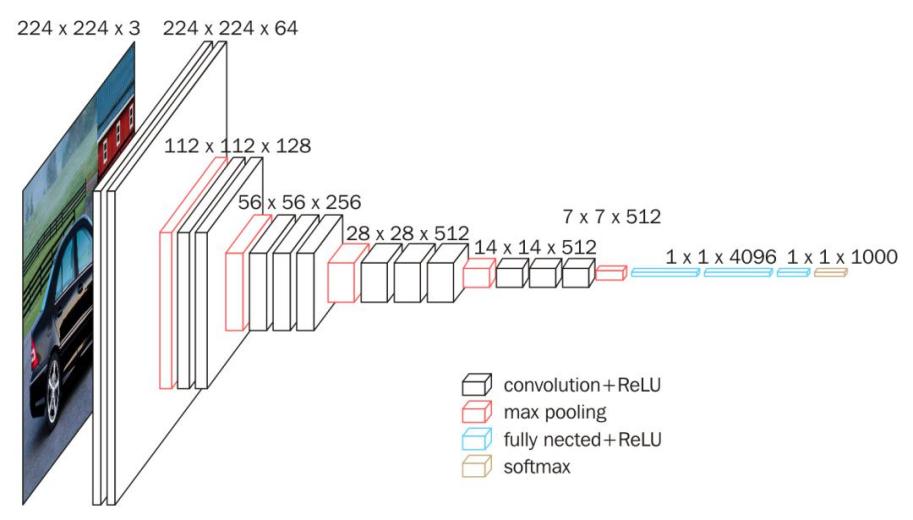


### 扁平化算子介绍

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：

### VGG16神经网络介绍

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。



### 实验目录结构介绍

./

|── file

|    |── tulips\_demo.jpg

|    |── vgg16\_ckpt.npy

|── layer.py

|── main.py

|── vgg.py

|── 搭建VGG16神经网络实现图像分类.ipynb

## 实验环境要求

* 操作系统：Ubuntu X86
* Python版本：Python3.7.5

## 实验详细设计与实现

### 实验步骤

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。

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。

代码：

# 导入相关依赖

import cv2

import sys

import numpy as np

from numba import jit

# 缩放图片

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

# 预处理流程

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)

image = image.transpose((2, 0, 1))

return image

搭建卷积算子

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，输出张量的通道数，卷积核的尺寸，边界扩充大小P，卷积步长。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：

反向传播公式：

代码：

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

搭建最大池化算子

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，池化步长。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

反向传播公式

其中 和 在正向传播中记录下的索引值。

代码：

# 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

搭建扁平化算子

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，扁平化至二维反向传播计算：进行Flatten算子反向传播的计算。将二维梯度映射回四维梯度即可。

代码：

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)

搭建VGG16神经网络

代码：

# 导入其它算子

from layer import ReluLayer, FullyConnectLayer, CrossEntropy

class VGG16(object):

def \_\_init\_\_(self, num\_classes=4):

#=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):

# 构建VGG16网络图

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():

# 正向传播的同时，打印均值和总和，用于核对执行过程是否正确

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())):

# 反向传播的同时，打印均值和总和，用于核对执行过程是否正确

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')

print(metric)

模型预测

加载提供的权重文件，对测试数据进行测试，输出预测结果和过程梯度信息。

代码：

# 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(f'forward outputs: {outputs}')

pred = int(np.argmax(outputs))

print(f'predict class: {CLASSES[pred]}')

# 计算loss

label = np.array([1, ])

loss\_func = CrossEntropy()

loss = loss\_func.forward(outputs, label)

print(f'loss: {loss}')

# 反向传播

grad = loss\_func.backward()

grad = model.backward(grad)

end\_time = time.time()

print(f'current task cost time: {end\_time - start\_time}')

输出：

reloaded success

forward: layer1\_conv1: -0.06873162090778351 -10346.033203125

forward: layer1\_relu1: -0.05778738483786583 -185570.546875

forward: layer1\_conv2: 0.28507915139198303 915464.4375

forward: layer1\_relu2: -0.013939479365944862 -44763.34765625

forward: layer1\_maxpool: 0.33006376028060913 1059921.875

forward: layer2\_conv1: 0.4298982322216034 345129.1875

forward: layer2\_relu1: 0.017751257866621017 28501.98828125

forward: layer2\_conv2: 0.3358146846294403 539194.8125

forward: layer2\_relu2: 0.029406482353806496 47215.98828125

forward: layer2\_maxpool: 0.34632712602615356 556073.9375

forward: layer3\_conv1: 0.5311101675033569 213191.875

forward: layer3\_relu1: 0.003264831844717264 2621.059326171875

forward: layer3\_conv2: 0.3423604369163513 274852.4375

forward: layer3\_relu2: 0.018941137939691544 15206.248046875

forward: layer3\_conv3: 0.3454575836658478 277338.875

forward: layer3\_relu3: -0.013854059390723705 -11122.2607421875

forward: layer3\_maxpool: 0.32738643884658813 262831.0625

forward: layer4\_conv1: 0.596550703048706 119730.109375

forward: layer4\_relu1: 0.0014814476016908884 594.6649169921875

forward: layer4\_conv2: 0.33410972356796265 134114.3125

forward: layer4\_relu2: 0.0025864504277706146 1038.221923828125

forward: layer4\_conv3: 0.3303680717945099 132612.390625

forward: layer4\_relu3: 0.037846896797418594 15192.046875

forward: layer4\_maxpool: 0.3491385281085968 140147.0

forward: layer5\_conv1: 0.695088267326355 69753.5

forward: layer5\_relu1: 0.01296419557183981 1300.98291015625

forward: layer5\_conv2: 0.35738039016723633 35863.8359375

forward: layer5\_relu2: -0.03126288950443268 -3137.29345703125

forward: layer5\_conv3: 0.35006460547447205 35129.68359375

forward: layer5\_relu3: -0.007918775081634521 -794.6649169921875

forward: layer5\_maxpool: 0.36700019240379333 36829.203125

forward: flatten: 0.7064959406852722 17724.5703125

forward: fullyconnect1: 0.7064959406852722 17724.5703125

forward: relu1: 0.09846813976764679 403.32550048828125

forward: fullyconnect2: 0.3777786195278168 1547.3812255859375

forward: relu2: 0.05010776221752167 205.24139404296875

forward: fullyconnect3: 0.2082405686378479 852.953369140625

[[-2.7988272 0.46775296 -2.3161108 5.406887 ]]

tulips

4.946979522705078

backward: fullyconnect3: 9.98261384665966e-09 3.993045538663864e-08

backward: relu2: 6.210785606519026e-05 0.2543937784430193

backward: fullyconnect2: 0.0012413157711905797 5.0844293987966145

backward: relu1: 0.00024674972688734474 1.010686881330564

backward: fullyconnect1: 0.0006194986565929538 2.537466497404739

backward: flatten: 8.395934443405265e-05 2.106372033161513

backward: layer5\_maxpool: 8.395934443405265e-05 2.106372033161513

backward: layer5\_relu3: 2.0989836108513163e-05 2.106372033161513

backward: layer5\_conv3: 2.941355971560495e-05 2.951709544580388

backward: layer5\_relu2: 8.840234590209373e-06 0.887135221596691

backward: layer5\_conv2: 3.344091892276033e-05 3.3558630957368445

backward: layer5\_relu1: 2.1671756859541487e-05 2.174804144368707

backward: layer5\_conv1: 4.047363868077416e-05 4.061610588893048

backward: layer4\_maxpool: 3.531487219315039e-05 3.5439180543270283

backward: layer4\_relu3: 8.8287180482876e-06 3.5439180543270288

backward: layer4\_conv3: 1.2349069365368748e-05 4.957015235813938

backward: layer4\_relu2: 7.974775863263001e-07 0.32011388297206744

backward: layer4\_conv2: 1.3892953178508677e-05 5.576742549478811

backward: layer4\_relu1: -7.827718956028219e-06 -3.1421090107013754

backward: layer4\_conv1: 9.923590299026063e-06 3.983408534751454

backward: layer3\_maxpool: -7.271325790365325e-06 -1.4593841714294822

backward: layer3\_relu3: -1.81783144759133e-06 -1.4593841714294813

backward: layer3\_conv3: 1.2824905789958983e-06 1.029603956667171

backward: layer3\_relu2: 1.1030134089646165e-05 8.855168129313375

backward: layer3\_conv2: 1.0308848533072198e-05 8.27610854392689

backward: layer3\_relu1: 8.370184564961761e-06 6.719718091704341

backward: layer3\_conv1: 1.2672711583027946e-05 10.173855622240163

backward: layer2\_maxpool: -4.74955677950752e-06 -1.9065100877485546

backward: layer2\_relu2: -1.18738919487688e-06 -1.9065100877485546

backward: layer2\_conv2: 2.617759634026836e-06 4.203158636701777

backward: layer2\_relu1: 1.547701240962475e-07 0.24850386389290607

backward: layer2\_conv1: 4.2199326817632846e-06 6.775658951684946

backward: layer1\_maxpool: -8.022843311243627e-06 -6.440866975759364

backward: layer1\_relu2: -2.00571082781091e-06 -6.440866975759374

backward: layer1\_conv2: -3.652575644294478e-07 -1.1729384673799663

backward: layer1\_relu1: 9.58206653922831e-08 0.3077054532302846

backward: layer1\_conv1: 4.2874589455088885e-07 1.3768162563190656

-3.045751659149646e-06 -0.4584709057484779

123.349449634552

## 实验总结

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。

## 实验任务与参考解答任务

### 实验任务

* 成功搭建VGG16网络并且成功加载参数
* 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
* VGG16可正确反向传播，输出指定梯度

### 参考答案

步骤二：搭建卷积算子中填补TODO代码：

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

步骤三：搭建最大池化算子中填补空缺代码：

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