****

**智能计算系统期末大作业**

**（2026届）**

|  |  |
| --- | --- |
| **题 目** | 基于VGG19实现图像分类 |
| **学 院** | 计算机学院 |
| **专 业** | 计算机科学与技术专业 |
| **班 级** |  |
| **学 号** |  |
| **学生姓名** |  |
| **指导教师** | 戴玮辰 |
| **完成日期** | 2024年6月 |

目录

[一、实验目的 2](#_Toc169634838)

[二、实验环境 2](#_Toc169634839)

[三、实验内容 2](#_Toc169634840)

[四、代码实现 3](#_Toc169634841)

[4.1 一级目录代码 3](#_Toc169634842)

[4.2 二级目录代码 5](#_Toc169634843)

[五、结果分析 16](#_Toc169634844)

[5.1实验评估标准 16](#_Toc169634845)

[5.2 实验结果分析 17](#_Toc169634846)

[六、实验思考 17](#_Toc169634847)

[七、实验总结 18](#_Toc169634848)

[八、成员分工 19](#_Toc169634849)

# 一、实验目的

掌握卷积神经网络的设计原理，掌握卷积神经网络的使用方法，能够使用Python语言

实现 VGG19B]网络模型对给定的输入图像进行分类。具体包括:

1)加深对深度卷积神经网络中卷积层、最大池化层等基本单元的理解。

2)利用Python 语言实现 VGG19的前向传播计算，加深对 VGG19 网络结构的理解，为

后续风格迁移中使用 VGG19网络计算风格损失奠定基础。

3)在第2.1节实验的基础上将三层神经网络扩展为VGG19网络，加深对神经网络工程实

现中基本模块演变的理解，为后续建立更复杂的综合实验(如风格迁移)定基础。

# 二、实验环境

硬件环境:CPU。

软件环境:Python编译环境及相关的扩展库，包括Python2.7.9，Pillow 3.4.2，SciPy

0.18.1，NumPy1.11.2(本实验不需使用 TensorFlow等深度学习框架)。

数据集:官方训练VGG19使用的数据集为ImageNet44。该数据集包括约128万训练

图像和5万张验证图像，共有1000个不同的类别。本实验使用了官方训练好的模型参数，

并不需要直接使用 ImageNet 数据集进行 VGG19 模型的训练。

# 三、实验内容

本实验利用 VGG19网络进行图像分类。首先建立VGG19的网络结构,然后利用 VGG19

的官方模型参数对给定图像进行分类。VGG19网络的模型参数是在ImageNet[41数据集上

训练获得，其输出结果对应ImageNet数据集中的1000个类别概率。

在工程实现中,依然按照第2.1节实验的模块划分方法,每个模块的具体实现基于第2.1节

实验进行改进。由于本实验只涉及VGG19网络的推断过程，因此本实验仅包括数据加载

模块、基本单元模块、网络结构模块、网络推断模块，不包括网络训练模块

# 四、代码实现

## 4.1 一级目录代码

1.main\_exp\_3\_1.py

# Import necessary libraries  
from stu\_upload.vgg\_cpu import VGG19 # Import the VGG19 class from the specified module  
import numpy as np # Import NumPy for numerical operations  
import time # Import time module to measure inference time  
  
  
# Function to compute Mean Squared Error (MSE) between two data arrays  
def computeMse(data1, data2):  
 errors = []  
 # Calculate the difference between corresponding elements in the two arrays  
 for i in range(len(data1)):  
 errors.append(data1[i] - data2[i])  
  
 squared\_error = []  
 # Square each error  
 for val in errors:  
 squared\_error.append(pow(val, 2))  
  
 # Return the mean of the squared errors  
 return sum(squared\_error) / len(squared\_error)  
  
  
# Function to perform forward propagation through the VGG network  
def forward(vgg):  
 print('Inferencing...')  
 start\_time = time.time() # Record the start time of inference  
 current = vgg.input\_image # Start with the input image  
 pool5 = np.array([]) # Initialize pool5 as an empty array  
  
 # Iterate through each layer in the VGG model  
 for idx in range(len(vgg.param\_layer\_name)):  
 print('Inferencing layer: ' + vgg.param\_layer\_name[idx])  
  
 # Perform forward propagation through the current layer  
 current = vgg.layers[vgg.param\_layer\_name[idx]].forward(current)  
  
 # Save the output of the pool5 layer  
 if 'pool5' in vgg.param\_layer\_name[idx]:  
 pool5 = current  
  
 # Print the total inference time  
 print('Inference time: %f' % (time.time() - start\_time))  
 return current, pool5 # Return the final output and pool5 layer output  
  
  
# Function to check the correctness of the pool5 layer output  
def check\_pool5(stu\_pool5):  
 data = np.load('pool5\_dump.npy') # Load the correct pool5 output  
 pool5\_mse = computeMse(stu\_pool5.flatten(), data.flatten()) # Compute MSE between student and correct outputs  
 print('test pool5 mse: %f' % pool5\_mse)  
  
 # Check if the MSE is below a threshold  
 if pool5\_mse < 0.003:  
 print('CHECK POOL5 PASS.')  
 else:  
 print('CHECK POOL5 FAILED.')  
 exit() # Exit if the check fails  
  
  
# Function to evaluate the VGG model on the input image  
def evaluate(vgg):  
 prob, pool5 = forward(vgg) # Perform forward propagation  
 print('--------------检测结果------------------------')  
  
 # Get the index of the highest probability class  
 top1 = np.argmax(prob[0])  
 print('Classification result: id = %d, prob = %f' % (top1, prob[0, top1])) # Print the classification result  
 return pool5 # Return the pool5 layer output  
  
  
# Main function  
if \_\_name\_\_ == '\_\_main\_\_':  
 print('-------------------------------')  
 vgg = VGG19(param\_path='../imagenet-vgg-verydeep-19.mat') # Initialize the VGG19 model with the parameter file  
 vgg.build\_model() # Build the VGG19 model  
 vgg.init\_model() # Initialize the model  
 vgg.load\_model() # Load the model parameters  
 vgg.load\_image('../cat1.jpg') # Load the input image  
 pool5 = evaluate(vgg) # Evaluate the model  
 print('-------------------------------')  
 check\_pool5(pool5) # Check the pool5 layer output

## 4.2 二级目录代码

1.layers\_1.py

# Import necessary libraries  
import sys # Provides access to some variables used or maintained by the Python interpreter  
import numpy as np # Import NumPy for numerical operations  
import struct # For handling binary data (not used in this code)  
import os # Provides functions for interacting with the operating system  
import time # Import time module to measure execution time  
  
  
# Function to show matrix information (currently commented out)  
def show\_matrix(mat, name):  
 # Uncomment to print the matrix's shape, mean, and standard deviation  
 # print(name + str(mat.shape) + ' mean %f, std %f' % (mat.mean(), mat.std()))  
 pass  
  
  
# Function to show time information (currently commented out)  
def show\_time(time, name):  
 # Uncomment to print the time information  
 # print(name + str(time))  
 pass  
  
  
# Class representing a Fully Connected (FC) Layer in a neural network  
class FullyConnectedLayer(object):  
 def \_\_init\_\_(self, num\_input, num\_output): # Initialization of the FC layer  
 self.num\_input = num\_input # Number of input neurons  
 self.num\_output = num\_output # Number of output neurons  
 print('\tFully connected layer with input %d, output %d.' % (self.num\_input, self.num\_output))  
  
 def init\_param(self, std=0.01): # Parameter initialization  
 self.weight = np.random.normal(loc=0.0, scale=std, size=(  
 self.num\_input, self.num\_output)) # Initialize weights with a normal distribution  
 self.bias = np.zeros([1, self.num\_output]) # Initialize biases with zeros  
 show\_matrix(self.weight, 'fc weight ') # Display weight matrix information  
 show\_matrix(self.bias, 'fc bias ') # Display bias matrix information  
  
 def forward(self, input): # Forward propagation  
 start\_time = time.time() # Record the start time  
 self.input = input # Save the input for backpropagation  
 # Perform the forward pass: output = input \* weight + bias  
 self.output = np.dot(self.input, self.weight) + self.bias  
 return self.output # Return the output  
  
 def backward(self, top\_diff): # Backward propagation  
 # top\_diff is the gradient of the loss with respect to the output of this layer  
 # Compute gradients for weights and biases  
 self.d\_weight = np.dot(self.input.T, top\_diff) # Gradient of weights  
 self.d\_bias = np.dot(np.ones((1, self.input.shape[0])), top\_diff) # Gradient of biases  
 bottom\_diff = np.dot(top\_diff, self.weight.T) # Gradient of the input  
 return bottom\_diff # Return the gradient of the input  
  
 def get\_gradient(self):  
 return self.d\_weight, self.d\_bias # Return the gradients of weights and biases  
  
 def update\_param(self, lr): # Update parameters using gradients and learning rate  
 self.weight = self.weight - lr \* self.d\_weight # Update weights  
 self.bias = self.bias - lr \* self.d\_bias # Update biases  
  
 def load\_param(self, weight, bias): # Load parameters (weights and biases)  
 assert self.weight.shape == weight.shape # Ensure the shape matches  
 assert self.bias.shape == bias.shape # Ensure the shape matches  
 self.weight = weight # Load weights  
 self.bias = bias # Load biases  
 show\_matrix(self.weight, 'fc weight ') # Display weight matrix information  
 show\_matrix(self.bias, 'fc bias ') # Display bias matrix information  
  
 def save\_param(self): # Save parameters (weights and biases)  
 show\_matrix(self.weight, 'fc weight ') # Display weight matrix information  
 show\_matrix(self.bias, 'fc bias ') # Display bias matrix information  
 return self.weight, self.bias # Return weights and biases  
  
  
# Class representing a ReLU (Rectified Linear Unit) Layer in a neural network  
class ReLULayer(object):  
 def \_\_init\_\_(self):  
 print('\t Relu layer') # Initialization message  
  
 def forward(self, input): # Forward propagation  
 start\_time = time.time() # Record the start time  
 self.input = input # Save the input for backpropagation  
 # Perform the ReLU operation: output = max(input, 0)  
 output = np.maximum(self.input, 0)  
 return output # Return the output  
  
 def backward(self, top\_diff): # Backward propagation  
 # Compute the gradient of the ReLU operation  
 bottom\_diff = top\_diff # Gradient of the output  
 bottom\_diff[self.input < 0] = 0 # Zero out the gradients where the input was negative  
 return bottom\_diff # Return the gradient of the input  
  
  
# Class representing a Softmax Loss Layer in a neural network  
class SoftmaxLossLayer(object):  
 def \_\_init\_\_(self):  
 print('\tSoftmax loss layer.') # Initialization message  
  
 def forward(self, input): # Forward propagation  
 # Perform the softmax operation  
 input\_max = np.max(input, axis=1, keepdims=True) # Subtract the maximum value for numerical stability  
 input\_exp = np.exp(input - input\_max) # Compute the exponentials  
 exp\_sum = np.sum(input\_exp, axis=1, keepdims=True) # Sum of exponentials  
 self.prob = input\_exp / exp\_sum # Compute the probabilities  
 return self.prob # Return the probabilities  
  
 def get\_loss(self, label): # Compute the loss  
 self.batch\_size = self.prob.shape[0] # Number of samples in the batch  
 self.label\_onehot = np.zeros\_like(self.prob) # Initialize one-hot labels  
 self.label\_onehot[np.arange(self.batch\_size), label] = 1.0 # Create one-hot labels  
 loss = -np.sum(np.log(self.prob) \* self.label\_onehot) / self.batch\_size # Compute the loss  
 return loss # Return the loss  
  
 def backward(self): # Backward propagation  
 # Compute the gradient of the softmax loss  
 bottom\_diff = (self.prob - self.label\_onehot) / self.batch\_size # Gradient of the input  
 return bottom\_diff # Return the gradient of the input

2.layers\_2.py

import numpy as np  
import struct  
import os  
import time  
  
  
# Helper function to display matrix information  
def show\_matrix(mat, name):  
 # Uncomment the following line to print matrix details  
 # print(name + str(mat.shape) + ' mean %f, std %f' % (mat.mean(), mat.std()))  
 pass  
  
  
# Helper function to display time information  
def show\_time(time, name):  
 # Uncomment the following line to print time details  
 # print(name + str(time))  
 pass  
  
  
# Class representing a Convolutional Layer  
class ConvolutionalLayer(object):  
 def \_\_init\_\_(self, kernel\_size, channel\_in, channel\_out, padding, stride):  
 # Initialization of convolutional layer parameters  
 self.kernel\_size = kernel\_size  
 self.channel\_in = channel\_in  
 self.channel\_out = channel\_out  
 self.padding = padding  
 self.stride = stride  
 print('\tConvolutional layer with kernel size %d, input channel %d, output channel %d.' % (  
 self.kernel\_size, self.channel\_in, self.channel\_out))  
  
 def init\_param(self, std=0.01): # Parameter initialization  
 self.weight = np.random.normal(loc=0.0, scale=std,  
 size=(self.channel\_in, self.kernel\_size, self.kernel\_size, self.channel\_out))  
 self.bias = np.zeros([self.channel\_out])  
  
 def forward(self, input): # Forward propagation  
 start\_time = time.time()  
 self.input = input # Input shape: [N, C, H, W]  
  
 # Padding the input  
 height = input.shape[2] + 2 \* self.padding  
 width = input.shape[3] + 2 \* self.padding  
 self.input\_pad = np.zeros([self.input.shape[0], self.input.shape[1], height, width])  
 self.input\_pad[:, :, self.padding: self.padding + self.input.shape[2],  
 self.padding: self.padding + self.input.shape[3]] = self.input  
  
 # Calculating output dimensions  
 height\_out = (height - self.kernel\_size) // self.stride + 1  
 width\_out = (width - self.kernel\_size) // self.stride + 1  
  
 self.output = np.zeros([self.input.shape[0], self.channel\_out, height\_out, width\_out])  
  
 # Performing the convolution operation  
 for idxn in range(self.input.shape[0]):  
 for idxc in range(self.channel\_out):  
 for idxh in range(height\_out):  
 for idxw in range(width\_out):  
 self.output[idxn, idxc, idxh, idxw] = np.sum(  
 self.input\_pad[idxn, :, idxh \* self.stride: idxh \* self.stride + self.kernel\_size,  
 idxw \* self.stride: idxw \* self.stride + self.kernel\_size] \* self.weight[:, :, :, idxc]) + \  
 self.bias[idxc]  
  
 return self.output  
  
 def load\_param(self, weight, bias): # Loading parameters  
 assert self.weight.shape == weight.shape  
 assert self.bias.shape == bias.shape  
 self.weight = weight  
 self.bias = bias  
  
  
# Class representing a Max Pooling Layer  
class MaxPoolingLayer(object):  
 def \_\_init\_\_(self, kernel\_size, stride): # Initialization of max pooling layer parameters  
 self.kernel\_size = kernel\_size  
 self.stride = stride  
 print('\tMax pooling layer with kernel size %d, stride %d.' % (self.kernel\_size, self.stride))  
  
 def forward(self, input): # Forward propagation  
 start\_time = time.time()  
 self.input = input # Input shape: [N, C, H, W]  
 self.max\_index = np.zeros(self.input.shape)  
  
 # Calculating output dimensions  
 height\_out = (self.input.shape[2] - self.kernel\_size) // self.stride + 1  
 width\_out = (self.input.shape[3] - self.kernel\_size) // self.stride + 1  
  
 self.output = np.zeros([self.input.shape[0], self.input.shape[1], height\_out, width\_out])  
  
 # Performing the max pooling operation  
 for idxn in range(self.input.shape[0]):  
 for idxc in range(self.input.shape[1]):  
 for idxh in range(height\_out):  
 for idxw in range(width\_out):  
 self.output[idxn, idxc, idxh, idxw] = np.max(  
 self.input[idxn, idxc, idxh \* self.stride: idxh \* self.stride + self.kernel\_size,  
 idxw \* self.stride: idxw \* self.stride + self.kernel\_size])  
  
 return self.output  
  
  
# Class representing a Flatten Layer  
class FlattenLayer(object):  
 def \_\_init\_\_(self, input\_shape, output\_shape): # Initialization of flatten layer parameters  
 self.input\_shape = input\_shape  
 self.output\_shape = output\_shape  
 assert np.prod(self.input\_shape) == np.prod(self.output\_shape)  
 print('\tFlatten layer with input shape %s, output shape %s.' % (str(self.input\_shape), str(self.output\_shape)))  
  
 def forward(self, input): # Forward propagation  
 assert list(input.shape[1:]) == list(self.input\_shape)  
  
 # Transposing the input to match required dimensions  
 self.input = np.transpose(input, [0, 2, 3, 1])  
  
 # Reshaping the input to the output shape  
 self.output = self.input.reshape([self.input.shape[0]] + list(self.output\_shape))  
 show\_matrix(self.output, 'flatten out ')  
 return self.output

3.vgg\_cpu.py

import numpy as np  
import struct  
import os  
import scipy.io  
import time  
import sys  
  
# 添加当前文件所在目录到Python路径中  
sys.path.append(os.path.dirname(os.path.abspath(\_\_file\_\_)))  
  
# 导入自定义的神经网络层  
from layers\_1 import FullyConnectedLayer, ReLULayer, SoftmaxLossLayer  
from layers\_2 import ConvolutionalLayer, MaxPoolingLayer, FlattenLayer  
  
# 定义一个函数用于显示矩阵信息（目前未使用）  
def show\_matrix(mat, name):  
 #print(name + str(mat.shape) + ' mean %f, std %f' % (mat.mean(), mat.std()))  
 pass  
  
class VGG19(object):  
 def \_\_init\_\_(self, param\_path='../../imagenet-vgg-verydeep-19.mat'):  
 self.param\_path = param\_path  
 self.param\_layer\_name = (  
 'conv1\_1', 'relu1\_1', 'conv1\_2', 'relu1\_2', 'pool1',  
 'conv2\_1', 'relu2\_1', 'conv2\_2', 'relu2\_2', 'pool2',  
 'conv3\_1', 'relu3\_1', 'conv3\_2', 'relu3\_2', 'conv3\_3', 'relu3\_3', 'conv3\_4', 'relu3\_4', 'pool3',  
 'conv4\_1', 'relu4\_1', 'conv4\_2', 'relu4\_2', 'conv4\_3', 'relu4\_3', 'conv4\_4', 'relu4\_4', 'pool4',  
 'conv5\_1', 'relu5\_1', 'conv5\_2', 'relu5\_2', 'conv5\_3', 'relu5\_3', 'conv5\_4', 'relu5\_4', 'pool5',  
 'flatten', 'fc6', 'relu6', 'fc7', 'relu7', 'fc8', 'softmax'  
 )  
  
 def build\_model(self):  
 # 定义VGG19的网络结构  
 print('Building vgg-19 model...')  
  
 self.layers = {}  
 # 第1层卷积和激活  
 self.layers['conv1\_1'] = ConvolutionalLayer(3, 3, 64, 1, 1)  
 self.layers['relu1\_1'] = ReLULayer()  
  
 # 第2层卷积、激活和池化  
 self.layers['conv1\_2'] = ConvolutionalLayer(3, 64, 64, 1, 1)  
 self.layers['relu1\_2'] = ReLULayer()  
 self.layers['pool1'] = MaxPoolingLayer(2, 2)  
  
 # 第3层卷积、激活和池化  
 self.layers['conv2\_1'] = ConvolutionalLayer(3, 64, 128, 1, 1)  
 self.layers['relu2\_1'] = ReLULayer()  
 self.layers['conv2\_2'] = ConvolutionalLayer(3, 128, 128, 1, 1)  
 self.layers['relu2\_2'] = ReLULayer()  
 self.layers['pool2'] = MaxPoolingLayer(2, 2)  
  
 # 第4层卷积、激活和池化  
 self.layers['conv3\_1'] = ConvolutionalLayer(3, 128, 256, 1, 1)  
 self.layers['relu3\_1'] = ReLULayer()  
 self.layers['conv3\_2'] = ConvolutionalLayer(3, 256, 256, 1, 1)  
 self.layers['relu3\_2'] = ReLULayer()  
 self.layers['conv3\_3'] = ConvolutionalLayer(3, 256, 256, 1, 1)  
 self.layers['relu3\_3'] = ReLULayer()  
 self.layers['conv3\_4'] = ConvolutionalLayer(3, 256, 256, 1, 1)  
 self.layers['relu3\_4'] = ReLULayer()  
 self.layers['pool3'] = MaxPoolingLayer(2, 2)  
  
 # 第5层卷积、激活和池化  
 self.layers['conv4\_1'] = ConvolutionalLayer(3, 256, 512, 1, 1)  
 self.layers['relu4\_1'] = ReLULayer()  
 self.layers['conv4\_2'] = ConvolutionalLayer(3, 512, 512, 1, 1)  
 self.layers['relu4\_2'] = ReLULayer()  
 self.layers['conv4\_3'] = ConvolutionalLayer(3, 512, 512, 1, 1)  
 self.layers['relu4\_3'] = ReLULayer()  
 self.layers['conv4\_4'] = ConvolutionalLayer(3, 512, 512, 1, 1)  
 self.layers['relu4\_4'] = ReLULayer()  
 self.layers['pool4'] = MaxPoolingLayer(2, 2)  
  
 # 第6层卷积、激活和池化  
 self.layers['conv5\_1'] = ConvolutionalLayer(3, 512, 512, 1, 1)  
 self.layers['relu5\_1'] = ReLULayer()  
 self.layers['conv5\_2'] = ConvolutionalLayer(3, 512, 512, 1, 1)  
 self.layers['relu5\_2'] = ReLULayer()  
 self.layers['conv5\_3'] = ConvolutionalLayer(3, 512, 512, 1, 1)  
 self.layers['relu5\_3'] = ReLULayer()  
 self.layers['conv5\_4'] = ConvolutionalLayer(3, 512, 512, 1, 1)  
 self.layers['relu5\_4'] = ReLULayer()  
 self.layers['pool5'] = MaxPoolingLayer(2, 2)  
  
 # 将多维特征图展平  
 self.layers['flatten'] = FlattenLayer([512, 7, 7], [512 \* 7 \* 7])  
  
 # 全连接层和激活层  
 self.layers['fc6'] = FullyConnectedLayer(25088, 4096)  
 self.layers['relu6'] = ReLULayer()  
 self.layers['fc7'] = FullyConnectedLayer(4096, 4096)  
 self.layers['relu7'] = ReLULayer()  
  
 # 最后一层全连接层和Softmax层  
 self.layers['fc8'] = FullyConnectedLayer(4096, 1000)  
 self.layers['softmax'] = SoftmaxLossLayer()  
  
 # 保存需要更新参数的层  
 self.update\_layer\_list = []  
 for layer\_name in self.layers.keys():  
 if 'conv' in layer\_name or 'fc' in layer\_name:  
 self.update\_layer\_list.append(layer\_name)  
  
 def init\_model(self):  
 # 初始化VGG-19的每一层的参数  
 print('Initializing parameters of each layer in vgg-19...')  
 for layer\_name in self.update\_layer\_list:  
 self.layers[layer\_name].init\_param()  
  
 def load\_model(self):  
 # 从文件中加载预训练的参数  
 print('Loading parameters from file ' + self.param\_path)  
 params = scipy.io.loadmat(self.param\_path)  
 self.image\_mean = params['normalization'][0][0][0]  
 self.image\_mean = np.mean(self.image\_mean, axis=(0, 1))  
 print('Get image mean: ' + str(self.image\_mean))  
  
 for idx in range(43):  
 if 'conv' in self.param\_layer\_name[idx]:  
 weight, bias = params['layers'][0][idx][0][0][0][0]  
 # matconvnet: weights dim [height, width, in\_channel, out\_channel]  
 # ours: weights dim [in\_channel, height, width, out\_channel]  
 # 调整参数的形状  
 weight = np.transpose(weight, [2, 0, 1, 3])  
 bias = bias.reshape(-1)  
 self.layers[self.param\_layer\_name[idx]].load\_param(weight, bias)  
 if idx >= 37 and 'fc' in self.param\_layer\_name[idx]:  
 weight, bias = params['layers'][0][idx-1][0][0][0][0]  
 weight = weight.reshape([weight.shape[0] \* weight.shape[1] \* weight.shape[2], weight.shape[3]])  
 self.layers[self.param\_layer\_name[idx]].load\_param(weight, bias)  
  
 def load\_image(self, image\_dir):  
 # 加载并预处理图像  
 print('Loading and preprocessing image from ' + image\_dir)  
 self.input\_image = scipy.misc.imread(image\_dir) # 读取图像文件  
 self.input\_image = scipy.misc.imresize(self.input\_image, [224, 224, 3]) # 调整图像大小为 224x224 像素，3个通道  
 self.input\_image = np.array(self.input\_image).astype(np.float32) # 转换图像数据类型为 float32  
 self.input\_image -= self.image\_mean # 减去图像均值  
 self.input\_image = np.reshape(self.input\_image,  
 [1] + list(self.input\_image.shape)) # 调整图像维度为 [1, channel, height, width]  
  
 # 调整图片维度顺序  
 # input dim [N, channel, height, width]  
 # *TODO：调整图片维度顺序* ## *Begin* self.input\_image = np.transpose(self.input\_image, [0, 3, 1, 2]) # 调整维度顺序为 [N, height, width, channel]  
  
 ## End  
  
 def forward(self): # *TODO：神经网络的前向传播* print('Inferencing...')  
 start\_time = time.time()  
 current = self.input\_image # 获取输入图像数据  
 for idx in range(len(self.param\_layer\_name)):  
 print('Inferencing layer: ' + self.param\_layer\_name[idx])  
 ## Begin  
 current = self.layers[self.param\_layer\_name[idx]].forward(current) # 依次进行每一层的前向传播  
 ## End  
 print('Inference time: %f' % (time.time() - start\_time))  
 return current  
  
 def evaluate(self):  
 # *TODO：获取神经网络前向传播的结果* ## *Begin* prob = self.forward() # 获取前向传播的输出结果  
 ## End  
 top1 = np.argmax(prob[0]) # 获取最大概率对应的类别  
 print('Classification result: id = %d, prob = %f' % (top1, prob[0, top1])) # 打印分类结果及对应的概率  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 vgg = VGG19()  
 vgg.build\_model()  
 vgg.init\_model()  
 vgg.load\_model()  
 vgg.load\_image('../../cat1.jpg')  
 prob = vgg.evaluate()

# 五、结果分析

## 5.1实验评估标准

本实验的评估标准设定如下:

60分标准:给定卷积层和池化层的前向传播输入矩阵和参数值，可以得到正确的前向

传播输出矩阵。

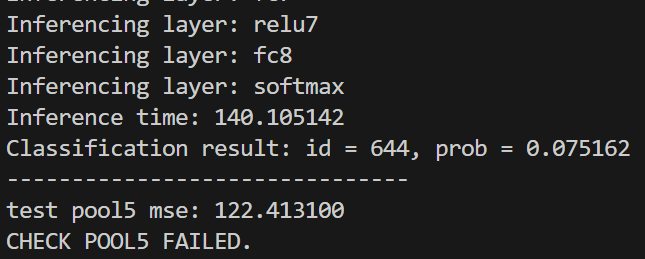
80分标准:建立VGG19网络后，给定VGG19的网络参数值和输入图像，可以得到

正确的 pool5 层输出结果。

90分标准:建立VGG19网络后，给定VGG19的网络参数值和输入图像，可以得到

正确的 Softmax层输出结果和正确的图像分类结果。

## 5.2 实验结果分析



为验证实验的正确性，选择猫咪的图像进行分类测试。该猫咪图像的真实类别为 tabby cat，对应 ImageNet 数据集类别编号的 644。实验结果将该图像的类别编号判断为 644。通过査询ImageNet数据集类别编号对应的具体类别，编号644对应tabbycat，说明利用 VGG19网络判断得到了正确的图像类别。

# 六、实验思考

选择题目2

2.（1）在实现深度神经网络后，如何确保整个网络的实现是正确的?

单元测试

针对每一层单独编写单元测试，通过给定已知输入和预期输出，确保每一层单独运行时的行为正确。还可以测试一些基础操作如矩阵乘法、激活函数等，确保它们在不同情况下都能正确工作。

集成测试，

在网络的不同子模块间进行集成测试，确保这些模块能正确协同工作。通过使用一些简单的已知输入和输出，测试整个网络的输出是否符合预期。

梯度检查

对于训练阶段，可以实施梯度检查，通过数值梯度和解析梯度的比较，确保反向传播的实现是正确的。还可以与已知正确实现的神经网络进行对比，在相同的输入下，检查输出是否一致。

观察损失函数的输出

通过验证损失函数的输出是否合理，来确保网络的实现正确。例如，对于一个分类问题，确保在训练初期的损失函数值较高，并随着训练逐渐降低。

2.（2）如果是网络中的某个层计算有误，如何快速定位到有错误的层?

在定位有错误的层时，可以逐层检查网络的输出。通过固定输入并检查每一层的输出，定位输出异常的层。可以在每一层的输出之后插入断点，打印中间结果或使用断点调试，检查数据在层间的传递过程。通过检查反向传播过程中梯度的变化，若某层的梯度异常（如梯度爆炸或消失），可能该层存在问题。使用可视化工具（如TensorBoard）可视化中间层的输出、权重和梯度，通过图形化界面，更直观地发现问题所在。针对可疑层，可以用已知的输入输出对其进行单独测试，验证其计算逻辑是否正确。可以从一个简单的模型开始构建，逐步增加复杂性，随时测试各层的正确性，这样可以迅速定位在增加新层或修改网络结构时引入的错误。

# 七、实验总结

本次实验通过实现VGG19网络来进行图像分类，旨在加深对卷积神经网络的理解和应用。通过搭建VGG19网络结构并使用预训练模型参数对给定图像进行分类，我们成功地验证了模型的有效性，并达到了预期的实验目标。

心得体会

理论与实践的结合：在实验过程中，通过实践加深了对卷积层、池化层等基本单元的理解。从最初的理论学习到实际编写代码，实现VGG19网络，使我们更加深刻地理解了深度学习模型的内部机制和工作原理。

编程技能的提升：在编写和调试代码的过程中，我们不仅提高了Python编程能力，还掌握了使用相关扩展库（如Pillow、SciPy、NumPy）进行数据处理和模型构建的方法。特别是对VGG19网络的前向传播计算过程有了更深入的了解，为后续的复杂实验打下了坚实的基础。

团队合作的重要性：在项目的不同阶段，我们小组成员分工明确，密切合作。从数据加载、网络结构的搭建到最终的模型推断，每个环节都体现了团队合作的力量。在遇到问题时，通过集思广益和相互支持，能够更快速地解决问题，确保项目的顺利进行。

问题解决能力的提高：在实验过程中，我们也遇到了一些挑战，例如如何确保网络实现的正确性、如何调试和优化模型等。通过编写单元测试、集成测试和梯度检查等方法，我们有效地验证了网络的正确性，并提升了问题定位和解决能力。这些经验将对我们未来的研究和工作大有裨益。

本次实验不仅让我们在技术上有所提升，更培养了我们解决问题和团队合作的能力，为将来从事相关领域的研究和工作打下了良好的基础

# 八、成员分工

陈：代码填写，文档书写

刘：代码注释，结果分析

柯：代码调试，文档书写