2017秋季零基础学Python第四组作品报告

一、作品名称：图像风格转移

二、成员介绍：

组长 孔庆哲

数学与统计学院

信息与计算科学专业

20151910026

组员 周楠

信息学院

计算机科学与技术

20161060167

组员 张兆阳

经济学院

财政学

组员 吴晓天

数学与统计学院

信息与计算科学

20161910052

组员 杨会

生命科学学院

生物技术

20161070136

三、作品简介：将一张图片的背景风格应用到另一张图片上，从而形成新的图片。可应用到媒体，后期等多种领域上，实现不同风格的照片。

四、总体设计

a）基本思路

将所需的两张图片通过梯度金字塔进行塔形分解，然后将其

融合形成新的梯度金字塔，再通过逆变换生成图片。之后利用人工智能深度学习对生成的图片不断优化。

b）主要技术难点和解决方案

图片的分解（通过塔形金字塔进行分解）

图片的逆合成（利用神经网络框架进行逆合成）

人工智能深度学习（利用vgg16卷积神经网络进行学习）

五、特色和创新点

通过对不同代码框架的利用和组装来达到我们所需要的目的，并且利用卷积神经网络和人工智能深度学习来实现图片的合成和优化。

六、运行截图

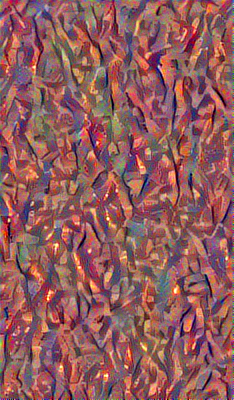
原图：



背景：



结果：



七、项目组成员的工作心得

成员一：孔庆哲，2015级，项目负责人

在本次小组任务，我很有幸能够担任组长一职，负责对组员进行协调以及将强小组的凝聚力。其次，通过本次学习，更加深刻的了解了python这门语言的适用范围之广，也加强了我对python编程的理解和对面对对象的理解。但是，在学习过程中，也发现了自己在独立思考和发现问题上有所欠缺，需要去进一步练习。

成员二：周楠，2016级，项目组员

这次小组任务中，主要负责程序的实现。对“深度学习”的兴趣也是我对python选择的理由，通过自学和对前辈成果的借鉴对图像处理有了一定的了解。在程序调试和运行的过程中，对每一个过程在matlab上进行输出。

对于小组合作，学会如何和组员分享想法，不再像数学建模那样小组里只有我一个人能理解我的思想，分工合作很重要，一个人的力量终究有限。

综上，学习python的过程很愉快，和组员的配合也不错，nice。

成员三：张兆阳，2016级，项目组员

学习能力良好，能够在平时学习和小组讨论中较好地掌握基本编程知识，了解项目工作原理。

在学习中能够做到独立思考并解决问题，但当遇到无法解决的问题时，能主动请教老师和同学，共同进步。

在作业和小组活动中能通过自身的实践和摸索发现问题和漏洞，并提出问题和大家讨论解决。

在设计选题及讨论方法时，能发表自身的意见并参与讨论，取长补短，完善自己，与小组成员共同成长进步。

成员四：吴晓天，2016级，项目组员

学习能力：学习能力一般，学习到较深刻的内容会有些吃力。

独立思考能力：思考能力较强，能想到一些ideas。

发现问题能力：编程基础较弱，对出现的问题不够敏感，较为薄弱。

其他方面着手：具有一定的整理资料的能力，能迅速筛选信息。

成员五：杨会，2016级，项目组员

学习能力：不强，需要其他组员协助。

独立思考能力：缺乏独立思考能力。

发现问题能力：有待提高，知识掌握不强。

其他方面着手：考虑多，团队合作可以。

八、存在的问题、建议及其他需要说明的情况

小组组员之间还存在这分工不明确的问题，同时对较高级的程序理念的理解有些困难。

九、附件：代码

from \_\_future\_\_ import print\_function

from keras.preprocessing.image import load\_img, img\_to\_array

from scipy.misc import imsave

import numpy as np

from scipy.optimize import fmin\_l\_bfgs\_b

import time

import argparse

from keras.applications import vgg16

from keras import backend as K

parser = argparse.ArgumentParser(description='Neural style transfer with Keras.')

parser.add\_argument('base\_image\_path', metavar='base', type=str,

help='Path to the image to transform.')

parser.add\_argument('style\_reference\_image\_path', metavar='ref', type=str,

help='Path to the style reference image.')

parser.add\_argument('result\_prefix', metavar='res\_prefix', type=str,

help='Prefix for the saved results.')

parser.add\_argument('--iter', type=int, default=10, required=False,

help='Number of iterations to run.')

parser.add\_argument('--content\_weight', type=float, default=0.025, required=False,

help='Content weight.')

parser.add\_argument('--style\_weight', type=float, default=1.0, required=False,

help='Style weight.')

parser.add\_argument('--tv\_weight', type=float, default=1.0, required=False,

help='Total Variation weight.')

args = parser.parse\_args()

base\_image\_path = args.base\_image\_path

style\_reference\_image\_path = args.style\_reference\_image\_path

result\_prefix = args.result\_prefix

iterations = args.iter

# these are the weights of the different loss components

total\_variation\_weight = args.tv\_weight

style\_weight = args.style\_weight

content\_weight = args.content\_weight

# dimensions of the generated picture.

width, height = load\_img(base\_image\_path).size

img\_nrows = 400

img\_ncols = int(width \* img\_nrows / height)

# util function to open, resize and format pictures into appropriate tensors

def preprocess\_image(image\_path):

img = load\_img(image\_path, target\_size=(img\_nrows, img\_ncols))

img = img\_to\_array(img)

img = np.expand\_dims(img, axis=0)

img = vgg16.preprocess\_input(img)

return img

# util function to convert a tensor into a valid image

def deprocess\_image(x):

if K.image\_data\_format() == 'channels\_first':

x = x.reshape((3, img\_nrows, img\_ncols))

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

else:

x = x.reshape((img\_nrows, img\_ncols, 3))

# Remove zero-center by mean pixel

x[:, :, 0] += 103.939

x[:, :, 1] += 116.779

x[:, :, 2] += 123.68

# 'BGR'->'RGB'

x = x[:, :, ::-1]

x = np.clip(x, 0, 255).astype('uint8')

return x

# get tensor representations of our images

base\_image = K.variable(preprocess\_image(base\_image\_path))

style\_reference\_image = K.variable(preprocess\_image(style\_reference\_image\_path))

# this will contain our generated image

if K.image\_data\_format() == 'channels\_first':

combination\_image = K.placeholder((1, 3, img\_nrows, img\_ncols))

else:

combination\_image = K.placeholder((1, img\_nrows, img\_ncols, 3))

# combine the 3 images into a single Keras tensor

input\_tensor = K.concatenate([base\_image,

style\_reference\_image,

combination\_image], axis=0)

# build the VGG16 network with our 3 images as input

# the model will be loaded with pre-trained ImageNet weights

model = vgg16.VGG16(input\_tensor=input\_tensor,

weights='imagenet', include\_top=False)

print('Model loaded.')

# get the symbolic outputs of each "key" layer (we gave them unique names).

outputs\_dict = dict([(layer.name, layer.output) for layer in model.layers])

# compute the neural style loss

# first we need to define 4 util functions

# the gram matrix of an image tensor (feature-wise outer product)

def gram\_matrix(x):

assert K.ndim(x) == 3

if K.image\_data\_format() == 'channels\_first':

features = K.batch\_flatten(x)

else:

features = K.batch\_flatten(K.permute\_dimensions(x, (2, 0, 1)))

gram = K.dot(features, K.transpose(features))

return gram

# the "style loss" is designed to maintain

# the style of the reference image in the generated image.

# It is based on the gram matrices (which capture style) of

# feature maps from the style reference image

# and from the generated image

def style\_loss(style, combination):

assert K.ndim(style) == 3

assert K.ndim(combination) == 3

S = gram\_matrix(style)

C = gram\_matrix(combination)

channels = 3

size = img\_nrows \* img\_ncols

return K.sum(K.square(S - C)) / (4. \* (channels \*\* 2) \* (size \*\* 2))

# an auxiliary loss function

# designed to maintain the "content" of the

# base image in the generated image

def content\_loss(base, combination):

return K.sum(K.square(combination - base))

# the 3rd loss function, total variation loss,

# designed to keep the generated image locally coherent

def total\_variation\_loss(x):

assert K.ndim(x) == 4

if K.image\_data\_format() == 'channels\_first':

a = K.square(x[:, :, :img\_nrows - 1, :img\_ncols - 1] - x[:, :, 1:, :img\_ncols - 1])

b = K.square(x[:, :, :img\_nrows - 1, :img\_ncols - 1] - x[:, :, :img\_nrows - 1, 1:])

else:

a = K.square(x[:, :img\_nrows - 1, :img\_ncols - 1, :] - x[:, 1:, :img\_ncols - 1, :])

b = K.square(x[:, :img\_nrows - 1, :img\_ncols - 1, :] - x[:, :img\_nrows - 1, 1:, :])

return K.sum(K.pow(a + b, 1.25))

# combine these loss functions into a single scalar

loss = K.variable(0.)

layer\_features = outputs\_dict['block4\_conv2']

base\_image\_features = layer\_features[0, :, :, :]

combination\_features = layer\_features[2, :, :, :]

loss += content\_weight \* content\_loss(base\_image\_features,

combination\_features)

feature\_layers = ['block1\_conv1', 'block2\_conv1',

'block3\_conv1', 'block4\_conv1',

'block5\_conv1']

for layer\_name in feature\_layers:

layer\_features = outputs\_dict[layer\_name]

style\_reference\_features = layer\_features[1, :, :, :]

combination\_features = layer\_features[2, :, :, :]

sl = style\_loss(style\_reference\_features, combination\_features)

loss += (style\_weight / len(feature\_layers)) \* sl

loss += total\_variation\_weight \* total\_variation\_loss(combination\_image)

# get the gradients of the generated image wrt the loss

grads = K.gradients(loss, combination\_image)

outputs = [loss]

if isinstance(grads, (list, tuple)):

outputs += grads

else:

outputs.append(grads)

f\_outputs = K.function([combination\_image], outputs)

def eval\_loss\_and\_grads(x):

if K.image\_data\_format() == 'channels\_first':

x = x.reshape((1, 3, img\_nrows, img\_ncols))

else:

x = x.reshape((1, img\_nrows, img\_ncols, 3))

outs = f\_outputs([x])

loss\_value = outs[0]

if len(outs[1:]) == 1:

grad\_values = outs[1].flatten().astype('float64')

else:

grad\_values = np.array(outs[1:]).flatten().astype('float64')

return loss\_value, grad\_values

# this Evaluator class makes it possible

# to compute loss and gradients in one pass

# while retrieving them via two separate functions,

# "loss" and "grads". This is done because scipy.optimize

# requires separate functions for loss and gradients,

# but computing them separately would be inefficient.

class Evaluator(object):

def \_\_init\_\_(self):

self.loss\_value = None

self.grads\_values = None

def loss(self, x):

assert self.loss\_value is None

loss\_value, grad\_values = eval\_loss\_and\_grads(x)

self.loss\_value = loss\_value

self.grad\_values = grad\_values

return self.loss\_value

def grads(self, x):

assert self.loss\_value is not None

grad\_values = np.copy(self.grad\_values)

self.loss\_value = None

self.grad\_values = None

return grad\_values

evaluator = Evaluator()

# run scipy-based optimization (L-BFGS) over the pixels of the generated image

# so as to minimize the neural style loss

if K.image\_data\_format() == 'channels\_first':

x = np.random.uniform(0, 255, (1, 3, img\_nrows, img\_ncols)) - 128.

else:

x = np.random.uniform(0, 255, (1, img\_nrows, img\_ncols, 3)) - 128.

for i in range(iterations):

print('Start of iteration', i)

start\_time = time.time()

x, min\_val, info = fmin\_l\_bfgs\_b(evaluator.loss, x.flatten(),

fprime=evaluator.grads, maxfun=20)

print('Current loss value:', min\_val)

# save current generated image

img = deprocess\_image(x.copy())

fname = result\_prefix + '\_at\_iteration\_%d.png' % i

imsave(fname, img)

end\_time = time.time()

print('Image saved as', fname)

print('Iteration %d completed in %ds' % (i, end\_time - start\_time))