Vgg.py

# Copyright (c) 2015-2017 Anish Athalye. Released under GPLv3.

import tensorflow as tf

import numpy as np

import scipy.io

VGG19\_LAYERS = (

'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'

)

def load\_net(data\_path):

data = scipy.io.loadmat(data\_path)

if not all(i in data for i in ('layers', 'classes', 'normalization')):

raise ValueError("You're using the wrong VGG19 data. Please follow the instructions in the README to download the correct data.")

mean = data['normalization'][0][0][0]

mean\_pixel = np.mean(mean, axis=(0, 1))

weights = data['layers'][0]

return weights, mean\_pixel

def net\_preloaded(weights, input\_image, pooling):

net = {}

current = input\_image

for i, name in enumerate(VGG19\_LAYERS):

kind = name[:4]

if kind == 'conv':

kernels, bias = weights[i][0][0][0][0]

# matconvnet: weights are [width, height, in\_channels, out\_channels]

# tensorflow: weights are [height, width, in\_channels, out\_channels]

kernels = np.transpose(kernels, (1, 0, 2, 3))

bias = bias.reshape(-1)

current = \_conv\_layer(current, kernels, bias)

elif kind == 'relu':

current = tf.nn.relu(current)

elif kind == 'pool':

current = \_pool\_layer(current, pooling)

net[name] = current

assert len(net) == len(VGG19\_LAYERS)

return net

def \_conv\_layer(input, weights, bias):

conv = tf.nn.conv2d(input, tf.constant(weights), strides=(1, 1, 1, 1),

padding='SAME')

return tf.nn.bias\_add(conv, bias)

def \_pool\_layer(input, pooling):

if pooling == 'avg':

return tf.nn.avg\_pool(input, ksize=(1, 2, 2, 1), strides=(1, 2, 2, 1),

padding='SAME')

else:

return tf.nn.max\_pool(input, ksize=(1, 2, 2, 1), strides=(1, 2, 2, 1),

padding='SAME')

def preprocess(image, mean\_pixel):

return image - mean\_pixel

def unprocess(image, mean\_pixel):

return image + mean\_pixel

stylize.py

# Copyright (c) 2015-2017 Anish Athalye. Released under GPLv3.

import vgg

import tensorflow as tf

import numpy as np

from sys import stderr

from PIL import Image

CONTENT\_LAYERS = ('relu4\_2', 'relu5\_2')

STYLE\_LAYERS = ('relu1\_1', 'relu2\_1', 'relu3\_1', 'relu4\_1', 'relu5\_1')

try:

reduce

except NameError:

from functools import reduce

def stylize(network, initial, initial\_noiseblend, content, styles, preserve\_colors, iterations,

content\_weight, content\_weight\_blend, style\_weight, style\_layer\_weight\_exp, style\_blend\_weights, tv\_weight,

learning\_rate, beta1, beta2, epsilon, pooling,

print\_iterations=None, checkpoint\_iterations=None):

"""

Stylize images.

This function yields tuples (iteration, image); `iteration` is None

if this is the final image (the last iteration). Other tuples are yielded

every `checkpoint\_iterations` iterations.

:rtype: iterator[tuple[int|None,image]]

"""

shape = (1,) + content.shape

style\_shapes = [(1,) + style.shape for style in styles]

content\_features = {}

style\_features = [{} for \_ in styles]

vgg\_weights, vgg\_mean\_pixel = vgg.load\_net(network)

layer\_weight = 1.0

style\_layers\_weights = {}

for style\_layer in STYLE\_LAYERS:

style\_layers\_weights[style\_layer] = layer\_weight

layer\_weight \*= style\_layer\_weight\_exp

# normalize style layer weights

layer\_weights\_sum = 0

for style\_layer in STYLE\_LAYERS:

layer\_weights\_sum += style\_layers\_weights[style\_layer]

for style\_layer in STYLE\_LAYERS:

style\_layers\_weights[style\_layer] /= layer\_weights\_sum

# compute content features in feedforward mode

g = tf.Graph()

with g.as\_default(), g.device('/cpu:0'), tf.Session() as sess:

image = tf.placeholder('float', shape=shape)

net = vgg.net\_preloaded(vgg\_weights, image, pooling)

content\_pre = np.array([vgg.preprocess(content, vgg\_mean\_pixel)])

for layer in CONTENT\_LAYERS:

content\_features[layer] = net[layer].eval(feed\_dict={image: content\_pre})

# compute style features in feedforward mode

for i in range(len(styles)):

g = tf.Graph()

with g.as\_default(), g.device('/cpu:0'), tf.Session() as sess:

image = tf.placeholder('float', shape=style\_shapes[i])

net = vgg.net\_preloaded(vgg\_weights, image, pooling)

style\_pre = np.array([vgg.preprocess(styles[i], vgg\_mean\_pixel)])

for layer in STYLE\_LAYERS:

features = net[layer].eval(feed\_dict={image: style\_pre})

features = np.reshape(features, (-1, features.shape[3]))

gram = np.matmul(features.T, features) / features.size

style\_features[i][layer] = gram

initial\_content\_noise\_coeff = 1.0 - initial\_noiseblend

# make stylized image using backpropogation

with tf.Graph().as\_default():

if initial is None:

noise = np.random.normal(size=shape, scale=np.std(content) \* 0.1)

initial = tf.random\_normal(shape) \* 0.256

else:

initial = np.array([vgg.preprocess(initial, vgg\_mean\_pixel)])

initial = initial.astype('float32')

noise = np.random.normal(size=shape, scale=np.std(content) \* 0.1)

initial = (initial) \* initial\_content\_noise\_coeff + (tf.random\_normal(shape) \* 0.256) \* (1.0 - initial\_content\_noise\_coeff)

image = tf.Variable(initial)

net = vgg.net\_preloaded(vgg\_weights, image, pooling)

# content loss

content\_layers\_weights = {}

content\_layers\_weights['relu4\_2'] = content\_weight\_blend

content\_layers\_weights['relu5\_2'] = 1.0 - content\_weight\_blend

content\_loss = 0

content\_losses = []

for content\_layer in CONTENT\_LAYERS:

content\_losses.append(content\_layers\_weights[content\_layer] \* content\_weight \* (2 \* tf.nn.l2\_loss(

net[content\_layer] - content\_features[content\_layer]) /

content\_features[content\_layer].size))

content\_loss += reduce(tf.add, content\_losses)

# style loss

style\_loss = 0

for i in range(len(styles)):

style\_losses = []

for style\_layer in STYLE\_LAYERS:

layer = net[style\_layer]

\_, height, width, number = map(lambda i: i.value, layer.get\_shape())

size = height \* width \* number

feats = tf.reshape(layer, (-1, number))

gram = tf.matmul(tf.transpose(feats), feats) / size

style\_gram = style\_features[i][style\_layer]

style\_losses.append(style\_layers\_weights[style\_layer] \* 2 \* tf.nn.l2\_loss(gram - style\_gram) / style\_gram.size)

style\_loss += style\_weight \* style\_blend\_weights[i] \* reduce(tf.add, style\_losses)

# total variation denoising

tv\_y\_size = \_tensor\_size(image[:,1:,:,:])

tv\_x\_size = \_tensor\_size(image[:,:,1:,:])

tv\_loss = tv\_weight \* 2 \* (

(tf.nn.l2\_loss(image[:,1:,:,:] - image[:,:shape[1]-1,:,:]) /

tv\_y\_size) +

(tf.nn.l2\_loss(image[:,:,1:,:] - image[:,:,:shape[2]-1,:]) /

tv\_x\_size))

# overall loss

loss = content\_loss + style\_loss + tv\_loss

# optimizer setup

train\_step = tf.train.AdamOptimizer(learning\_rate, beta1, beta2, epsilon).minimize(loss)

def print\_progress():

stderr.write(' content loss: %g\n' % content\_loss.eval())

stderr.write(' style loss: %g\n' % style\_loss.eval())

stderr.write(' tv loss: %g\n' % tv\_loss.eval())

stderr.write(' total loss: %g\n' % loss.eval())

# optimization

best\_loss = float('inf')

best = None

with tf.Session() as sess:

sess.run(tf.global\_variables\_initializer())

stderr.write('Optimization started...\n')

if (print\_iterations and print\_iterations != 0):

print\_progress()

for i in range(iterations):

stderr.write('Iteration %4d/%4d\n' % (i + 1, iterations))

train\_step.run()

last\_step = (i == iterations - 1)

if last\_step or (print\_iterations and i % print\_iterations == 0):

print\_progress()

if (checkpoint\_iterations and i % checkpoint\_iterations == 0) or last\_step:

this\_loss = loss.eval()

if this\_loss < best\_loss:

best\_loss = this\_loss

best = image.eval()

img\_out = vgg.unprocess(best.reshape(shape[1:]), vgg\_mean\_pixel)

if preserve\_colors and preserve\_colors == True:

original\_image = np.clip(content, 0, 255)

styled\_image = np.clip(img\_out, 0, 255)

# Luminosity transfer steps:

# 1. Convert stylized RGB->grayscale accoriding to Rec.601 luma (0.299, 0.587, 0.114)

# 2. Convert stylized grayscale into YUV (YCbCr)

# 3. Convert original image into YUV (YCbCr)

# 4. Recombine (stylizedYUV.Y, originalYUV.U, originalYUV.V)

# 5. Convert recombined image from YUV back to RGB

# 1

styled\_grayscale = rgb2gray(styled\_image)

styled\_grayscale\_rgb = gray2rgb(styled\_grayscale)

# 2

styled\_grayscale\_yuv = np.array(Image.fromarray(styled\_grayscale\_rgb.astype(np.uint8)).convert('YCbCr'))

# 3

original\_yuv = np.array(Image.fromarray(original\_image.astype(np.uint8)).convert('YCbCr'))

# 4

w, h, \_ = original\_image.shape

combined\_yuv = np.empty((w, h, 3), dtype=np.uint8)

combined\_yuv[..., 0] = styled\_grayscale\_yuv[..., 0]

combined\_yuv[..., 1] = original\_yuv[..., 1]

combined\_yuv[..., 2] = original\_yuv[..., 2]

# 5

img\_out = np.array(Image.fromarray(combined\_yuv, 'YCbCr').convert('RGB'))

yield (

(None if last\_step else i),

img\_out

)

def \_tensor\_size(tensor):

from operator import mul

return reduce(mul, (d.value for d in tensor.get\_shape()), 1)

def rgb2gray(rgb):

return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])

def gray2rgb(gray):

w, h = gray.shape

rgb = np.empty((w, h, 3), dtype=np.float32)

rgb[:, :, 2] = rgb[:, :, 1] = rgb[:, :, 0] = gray

return rgb

neural.py

# Copyright (c) 2015-2017 Anish Athalye. Released under GPLv3.

import os

import numpy as np

import scipy.misc

from stylize import stylize

import math

from argparse import ArgumentParser

from PIL import Image

# default arguments

CONTENT\_WEIGHT = 5e0

CONTENT\_WEIGHT\_BLEND = 1

STYLE\_WEIGHT = 5e2

TV\_WEIGHT = 1e2

STYLE\_LAYER\_WEIGHT\_EXP = 1

LEARNING\_RATE = 2e1

BETA1 = 0.9

BETA2 = 0.999

EPSILON = 1e-08

STYLE\_SCALE = 1.0

ITERATIONS = 100

VGG\_PATH = 'imagenet-vgg-verydeep-19.mat'

POOLING = 'max'

def build\_parser():

parser = ArgumentParser()

parser.add\_argument('--content',

dest='content', help='content image',

metavar='CONTENT', required=True)

parser.add\_argument('--styles',

dest='styles',

nargs='+', help='one or more style images',

metavar='STYLE', required=True)

parser.add\_argument('--output',

dest='output', help='output path',

metavar='OUTPUT', required=True)

parser.add\_argument('--iterations', type=int,

dest='iterations', help='iterations (default %(default)s)',

metavar='ITERATIONS', default=ITERATIONS)

parser.add\_argument('--print-iterations', type=int,

dest='print\_iterations', help='statistics printing frequency',

metavar='PRINT\_ITERATIONS')

parser.add\_argument('--checkpoint-output',

dest='checkpoint\_output', help='checkpoint output format, e.g. output%%s.jpg',

metavar='OUTPUT')

parser.add\_argument('--checkpoint-iterations', type=int,

dest='checkpoint\_iterations', help='checkpoint frequency',

metavar='CHECKPOINT\_ITERATIONS')

parser.add\_argument('--width', type=int,

dest='width', help='output width',

metavar='WIDTH')

parser.add\_argument('--style-scales', type=float,

dest='style\_scales',

nargs='+', help='one or more style scales',

metavar='STYLE\_SCALE')

parser.add\_argument('--network',

dest='network', help='path to network parameters (default %(default)s)',

metavar='VGG\_PATH', default=VGG\_PATH)

parser.add\_argument('--content-weight-blend', type=float,

dest='content\_weight\_blend', help='content weight blend, conv4\_2 \* blend + conv5\_2 \* (1-blend) (default %(default)s)',

metavar='CONTENT\_WEIGHT\_BLEND', default=CONTENT\_WEIGHT\_BLEND)

parser.add\_argument('--content-weight', type=float,

dest='content\_weight', help='content weight (default %(default)s)',

metavar='CONTENT\_WEIGHT', default=CONTENT\_WEIGHT)

parser.add\_argument('--style-weight', type=float,

dest='style\_weight', help='style weight (default %(default)s)',

metavar='STYLE\_WEIGHT', default=STYLE\_WEIGHT)

parser.add\_argument('--style-layer-weight-exp', type=float,

dest='style\_layer\_weight\_exp', help='style layer weight exponentional increase - weight(layer<n+1>) = weight\_exp\*weight(layer<n>) (default %(default)s)',

metavar='STYLE\_LAYER\_WEIGHT\_EXP', default=STYLE\_LAYER\_WEIGHT\_EXP)

parser.add\_argument('--style-blend-weights', type=float,

dest='style\_blend\_weights', help='style blending weights',

nargs='+', metavar='STYLE\_BLEND\_WEIGHT')

parser.add\_argument('--tv-weight', type=float,

dest='tv\_weight', help='total variation regularization weight (default %(default)s)',

metavar='TV\_WEIGHT', default=TV\_WEIGHT)

parser.add\_argument('--learning-rate', type=float,

dest='learning\_rate', help='learning rate (default %(default)s)',

metavar='LEARNING\_RATE', default=LEARNING\_RATE)

parser.add\_argument('--beta1', type=float,

dest='beta1', help='Adam: beta1 parameter (default %(default)s)',

metavar='BETA1', default=BETA1)

parser.add\_argument('--beta2', type=float,

dest='beta2', help='Adam: beta2 parameter (default %(default)s)',

metavar='BETA2', default=BETA2)

parser.add\_argument('--eps', type=float,

dest='epsilon', help='Adam: epsilon parameter (default %(default)s)',

metavar='EPSILON', default=EPSILON)

parser.add\_argument('--initial',

dest='initial', help='initial image',

metavar='INITIAL')

parser.add\_argument('--initial-noiseblend', type=float,

dest='initial\_noiseblend', help='ratio of blending initial image with normalized noise (if no initial image specified, content image is used) (default %(default)s)',

metavar='INITIAL\_NOISEBLEND')

parser.add\_argument('--preserve-colors', action='store\_true',

dest='preserve\_colors', help='style-only transfer (preserving colors) - if color transfer is not needed')

parser.add\_argument('--pooling',

dest='pooling', help='pooling layer configuration: max or avg (default %(default)s)',

metavar='POOLING', default=POOLING)

return parser

def main():

parser = build\_parser()

options = parser.parse\_args()

if not os.path.isfile(options.network):

parser.error("Network %s does not exist. (Did you forget to download it?)" % options.network)

content\_image = imread(options.content)

style\_images = [imread(style) for style in options.styles]

width = options.width

if width is not None:

new\_shape = (int(math.floor(float(content\_image.shape[0]) /

content\_image.shape[1] \* width)), width)

content\_image = scipy.misc.imresize(content\_image, new\_shape)

target\_shape = content\_image.shape

for i in range(len(style\_images)):

style\_scale = STYLE\_SCALE

if options.style\_scales is not None:

style\_scale = options.style\_scales[i]

style\_images[i] = scipy.misc.imresize(style\_images[i], style\_scale \*

target\_shape[1] / style\_images[i].shape[1])

style\_blend\_weights = options.style\_blend\_weights

if style\_blend\_weights is None:

# default is equal weights

style\_blend\_weights = [1.0/len(style\_images) for \_ in style\_images]

else:

total\_blend\_weight = sum(style\_blend\_weights)

style\_blend\_weights = [weight/total\_blend\_weight

for weight in style\_blend\_weights]

initial = options.initial

if initial is not None:

initial = scipy.misc.imresize(imread(initial), content\_image.shape[:2])

# Initial guess is specified, but not noiseblend - no noise should be blended

if options.initial\_noiseblend is None:

options.initial\_noiseblend = 0.0

else:

# Neither inital, nor noiseblend is provided, falling back to random generated initial guess

if options.initial\_noiseblend is None:

options.initial\_noiseblend = 1.0

if options.initial\_noiseblend < 1.0:

initial = content\_image

if options.checkpoint\_output and "%s" not in options.checkpoint\_output:

parser.error("To save intermediate images, the checkpoint output "

"parameter must contain `%s` (e.g. `foo%s.jpg`)")

for iteration, image in stylize(

network=options.network,

initial=initial,

initial\_noiseblend=options.initial\_noiseblend,

content=content\_image,

styles=style\_images,

preserve\_colors=options.preserve\_colors,

iterations=options.iterations,

content\_weight=options.content\_weight,

content\_weight\_blend=options.content\_weight\_blend,

style\_weight=options.style\_weight,

style\_layer\_weight\_exp=options.style\_layer\_weight\_exp,

style\_blend\_weights=style\_blend\_weights,

tv\_weight=options.tv\_weight,

learning\_rate=options.learning\_rate,

beta1=options.beta1,

beta2=options.beta2,

epsilon=options.epsilon,

pooling=options.pooling,

print\_iterations=options.print\_iterations,

checkpoint\_iterations=options.checkpoint\_iterations

):

output\_file = None

combined\_rgb = image

if iteration is not None:

if options.checkpoint\_output:

output\_file = options.checkpoint\_output % iteration

else:

output\_file = options.output

if output\_file:

imsave(output\_file, combined\_rgb)

def imread(path):

img = scipy.misc.imread(path).astype(np.float)

if len(img.shape) == 2:

# grayscale

img = np.dstack((img,img,img))

elif img.shape[2] == 4:

# PNG with alpha channel

img = img[:,:,:3]

return img

def imsave(path, img):

img = np.clip(img, 0, 255).astype(np.uint8)

Image.fromarray(img).save(path, quality=95)

if \_\_name\_\_ == '\_\_main\_\_':

main()