数字图像处理期末项目报告

10195101447 赵胡洁

1. 项目要求

1. 一套完整的图像处理系统或app，实现数字图像处理的主要功能
2. 增加实际应用，要求场景明确
3. 实现以下功能之一：
    1. 图像风格迁移；
    2. 图像动作驱动；
    3. ……

    这里选择的第一个附加功能：图像风格迁移

2. 功能概述与过程分析

2.0. 环境和依赖

基本功能和风格迁移都是使用Python语言。使用numpy模块、PIL模块、matplotlib模块及tkinter模块实现了一个完整的GUI程序。

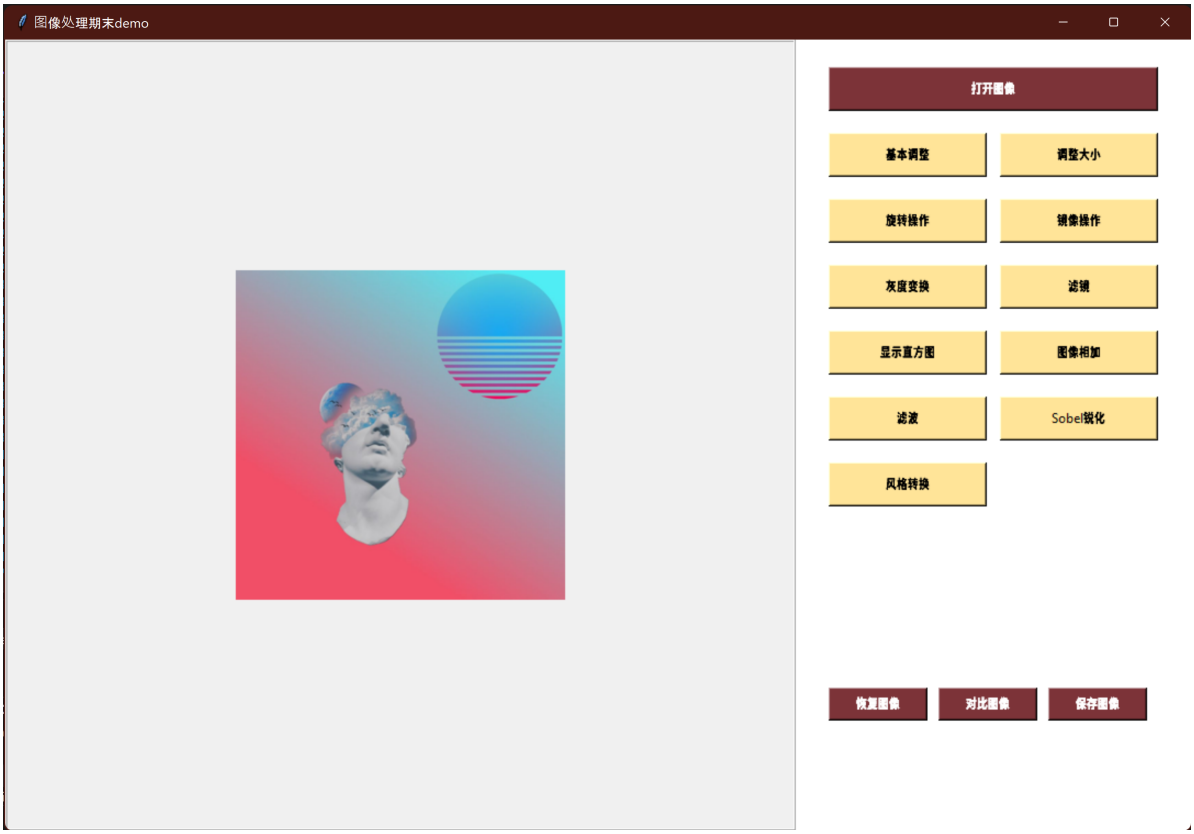
额外依赖: numpy, Pillow, torch, torchvision, opencv-python, matplotlib

2.1. 系统功能概述

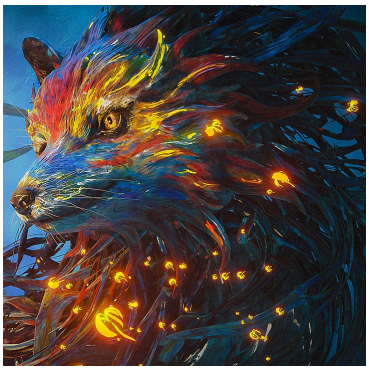
对数字图像进行基本调整(亮度，色彩度，对比度，锐度)、大小缩放调整、旋转操作、镜像操作、灰度变换、各种滤镜、直方图绘制、图像相加、图像滤波、锐化操作以及风格迁移的处理。详见readme

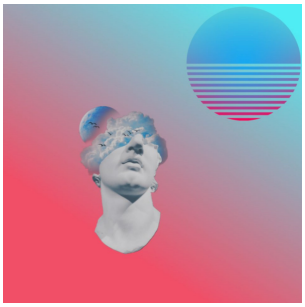
2.2. 程序运行界面截图

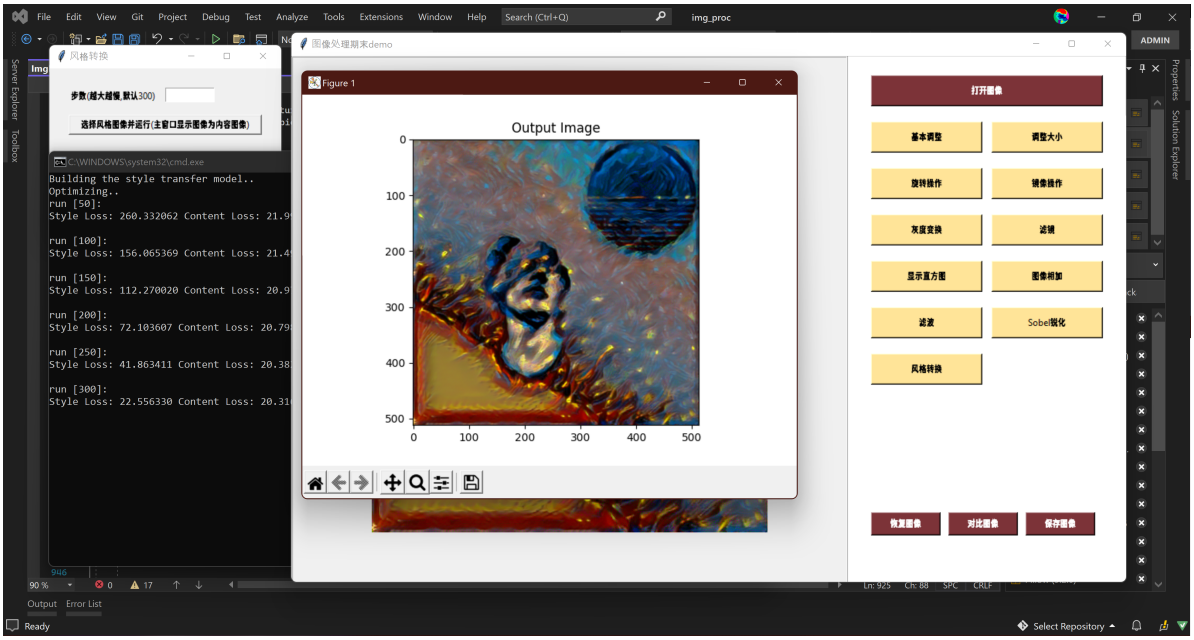
基本功能：



风格转换功能正常(优化步数300需要等待3分钟左右(device=cpu)，默认300):







2.3. 系统结构框架

主要包括Win()类，实现窗体用户界面的展示及直接对图像对象处理的一些方法；modules()类，实现对数字图像列表化以后的计算方法；picture()类，实现图像的压缩展示以及基本处理等功能。

文件结构

```
image_proc\
    build\              #pyinstaller生成
    dist\               #pyinstaller生成win可执行文件(包含环境)
    env\                #主文件执行环境
    ImgProc.py          #主文件
    ImgProc.spec        #pyinstaller生成配置
    readme.txt
```

2.4. 主要功能的实现

增强调整(亮度，色彩度，对比度，锐度)

使用Pillow的ImageEnhance模块

尺寸缩放操作

使用Pillow中的open将图片按RGBA模式打开，使用图片类型的.resize()成员函数直接进行修改

```
# ---Image.pyi---
    def resize(
        self,
        size: tuple[int, int],
        resample: _Resample | None = ...,
        box: tuple[float, float, float, float] | None = ...,
        reducing_gap: float | None = ...,
    ) -> Image: ...
```

图像旋转操作

类似地调取成员函数.rotate()，传递角度参数

图像镜像操作

类似地调取成员函数.rotate()

```
pic_mir_lr.transpose(Image.Transpose.FLIP_LEFT_RIGHT)# 镜像左右
pic_mir_tp.transpose(Image.Transpose.FLIP_LEFT_RIGHT)# 镜像上下
```

图像灰度变换

包含了3种不同的灰度变换方法：

1. n值化

```
#n值灰度转换
def tonpic(Imf, n):
    Im16=np.array(Imf)
    r,g,b = Im16[:,:,0], Im16[:,:,1], Im16[:,:,2]
    Im16 = 0.2989*r + 0.5870*g + 0.1140*b
    Im16=float(n)*Im16
    Im16[Im16>255]=255 #进行数据截断，大于255的值要截断为255
    Im16=np.round(Im16) #数据类型转化
    Im16=Im16.astype(np.uint8)
    return Image.fromarray(Im16)
```

首先对图像进行数组化，分离rgb channel，根据公式转换成灰度图。再进行乘n操作，并进行阶段操作。

2. 线性化

```
#线性灰度转换
def linearization(Imf, a, c):
    Im12=np.array(Imf)
    r,g,b = Im12[:,:,0], Im12[:,:,1], Im12[:,:,2]
    Im12 = 0.2989*r + 0.5870*g + 0.1140*b
    Im12=float(a)*Im12+float(c) #对矩阵类型计算,a是对比度，c是亮度，由k和b传入
    #进行数据截断，大于255的值要截断为255
    Im12[Im12>255]=255
    #数据类型转化
    Im12=np.round(Im12)
    Im12=Im12.astype(np.uint8)
    return Image.fromarray(Im12)
```

类似操作，首先对图像进行数组化，分离rgb channel，根据公式转换成灰度图。

再进行线性求值操作，得出最后结果。

3. 非线性化对数

```
#非线性log灰度转换
def tologpic(Imf, c):
    Im14=np.array(Imf)
    r,g,b = Im14[:,:,0], Im14[:,:,1], Im14[:,:,2]
    Im14 = 0.2989*r + 0.5870*g + 0.1140*b
    Im14 = c * np.log(1.0 + Im14) #对数运算
    Im14[Im14>255]=255 #
    Im14 = np.uint8(Im14 + 0.5)
    return Image.fromarray(Im14)
```

类似，首先分离rgb，转化成灰度图，根据参数c值进行对应的对数运算处理

图像滤镜功能

使用Pillow图像的成员函数.filter()，使用ImageFilter提供的参数

```
Im2 = Imf.filter(ImageFilter.BLUR)  # 图像模糊
Im4 = Imf.filter(ImageFilter.EDGE_ENHANCE)  # 边界增强
Im6 = Imf.filter(ImageFilter.GaussianBlur)  # 高斯模糊
Im8 = Imf.filter(ImageFilter.EMBOSS)  # 浮雕滤镜，
```

对于一些filter能够传入radius参数值来进行更细致的变更，这里没有进一步实现，待后续

彩色图像三通道直方图

通过调用成员函数.ravel()，将图片转化成一维数组，再使用collection.Counter()进行元素重复个数计数，统计得到每个bin值对应出现的次数。

```
def calc_hist(gray):
    # 计算彩色图单通道的直方图
    hist_new = []
    num = []
    hist_result = []
    hist_key = []
    gray1 = list(gray.ravel()) # 将读取出来的数组转化为一维列表方便循环遍历
    obj = dict(collections.Counter(gray1)) # 计算每个灰度级出现的次数
    obj = sorted(obj.items(),key=lambda item:item[0])
    # 初始化hist数组
    for each in obj:
        hist1 = []
        key = list(each)[0]
        cnt = list(each)[1]
        hist_key.append(key)
        hist1.append(cnt)
        hist_new.append(hist1)
    # 检查从0-255每个通道是否都有个数，没有的话添加并将值设为0
    for i in range (0, 256):
        if i in hist_key:
            num = hist_key.index(i)
            hist_result.append(hist_new[num])
        else:
            hist_result.append([0])
    hist_result = np.array(hist_result)
    return hist_result
```

外层调用calc_hist()函数分别对rgb channel进行直方图计算，最后整合到一张图中。

```
# 计算直方图
image= np.array(image)
r,g,b = image[:,:,0], image[:,:,1], image[:,:,2]
hist_new_b = modules.calc_hist(b)
hist_new_g = modules.calc_hist(g)
hist_new_r = modules.calc_hist(r)
```

两幅图像相加操作

以程序中当前显示的图片为标准，对之后选择的第二张图片的大小进行放缩与第一张图片相同，然后进行像素值相加，各图取一半的值，直接相加作为新图对应像素值。

```
# 图像相加函数
def IMG_PLUS(img1, img2):
    # 先修改img1尺寸和img2相同
    img1 = cv.resize(img1, (img2.shape[1], img2.shape[0]))
    # 矩阵相加
    newimg = img1*0.5 + img2*0.5
    newimg = newimg.astype(np.uint8)
    return newimg
```

滤波操作

1. 均值滤波

对图像每个像素取临近像素的平均值，默认filter大小3x3

首先对图像进行边界填充，遍历范围(padnum, side_len - padnum)

```
# 均值滤波处理函数
def mean_filter(img, b=3):
    padnum = (b-1)//2# 填充数量
    pad = ((padnum, padnum), (padnum, padnum), (0,0))# 填充格式
    Filter = np.ones((b, b, img.shape[2]), img.dtype)# 方阵滤波器
    padnumImg= np.pad(img, pad, 'constant', constant_values=(0, 0))
    # 用滤波器对图像中像素依次计算取均值
    for i in range(padnum, padnumImg.shape[0] - padnum):
        for j in range(padnum, padnumImg.shape[1] - padnum):
            padnumImg[i][j]=(Filter*padnumImg[i-padnum:i+padnum+1, \
                                              j-
padnum:j+padnum+1]).sum(axis=0).sum(axis=0)//(b ** 2)
    newimg=padnumImg[padnum:padnumImg.shape[0]-
padnum,padnum:padnumImg.shape[1]-padnum]  # 剪切使尺寸一样
    return newimg
```

2. 中值滤波

类似地，遍历每个像素，对相邻像素选择中位数作为此像素的新值，放在新图中

同样首先边界填充，然后进行遍历中值计算。

```
# 中值滤波处理函数
def median_filter(img, b=3):
    padnum = (b-1)//2# 填充数量
    pad = ((padnum, padnum), (padnum, padnum), (0,0))# 填充格式
    padImg= np.pad(img, pad, 'constant', constant_values=(0, 0))# 方阵滤波器
    # 按通道计算中值函数
    def DimensionAdd(img):
        blank = np.zeros((img.shape[2]))
```

```
        for i in range(img.shape[2]):
            blank[i] = np.median(img[:,:,i])
        return blank
    # 用滤波器对图像中像素依次计算中值
    for i in range(padnum, padImg.shape[0] - padnum):
        for j in range(padnum, padImg.shape[1] - padnum):
            padImg[i][j] = DimensionAdd(padImg[i-padnum:i+padnum+1, j-padnum:j+padnum+1])
    # 把操作完多余的0去除，保证尺寸一样大
    newimg = padImg[padnum:padImg.shape[0] - padnum, padnum:padImg.shape[1] - padnum]

    return newimg
```

Sobel算子锐化图像处理

首先将图片数组化，然后进行灰度处理转化为灰度图。之后使用sobel算子进行锐化操作

```
# sobel锐化
def sharpen(img):
    img = np.array(img)
    r,g,b = img[:,:,0], img[:,:,1], img[:,:,2]
    img = 0.2989*r + 0.5870*g + 0.1140*b
    # sobel算子
    G_x = np.array([[-1, 0, 1],[-2, 0, 2],[-1, 0, 1]])
    G_y = np.array([[-1, -2, -1],[0, 0, 0],[1, 2, 1]])
    rows = np.size(img, 0)
    columns = np.size(img, 1)
    mag = np.zeros(img.shape)
    # 分别检测水平和垂直，在计算每个pixel的时候，将水平和垂直的值作一次平方和的处理
    for i in range(0, rows - 2):
        for j in range(0, columns - 2):
            v = sum(sum(G_x * img[i:i+3, j:j+3]))  # vertical
            h = sum(sum(G_y * img[i:i+3, j:j+3]))  # horizon
            mag[i+1, j+1] = np.sqrt((v ** 2) + (h ** 2))
    # 设置阈值
    threshold=120
    mag[mag<threshold] = 0

    mag = mag.astype(np.uint8)
    return Image.fromarray(mag)
```

图片风格转换

步骤:

1. 打开第二张图作为风格图，为后续准备

2. 建立损失函数

内容损失(Content Loss):

内容损失是加权版本的函数单个层的内容距离。在输入X下该函数接收L层的特征图F_{XL}(即通过不同

filter进行卷积后得到的就是特征图feature maps)并返回加权版本的图像X与内容图像C的内容距离: $w_{CL} \cdot D_C^L(X, C)$。

内容图像的特征图F_{CL}必须被函数所知，以计算内容距离。将函数作为torch模块写，带有构造函数将作F_{CL}

为输入。距离$||F_{XL} - F_{CL}||^2$是两组特征映射之间的均方误差MSE，可以用'nn.MSELoss'来计算。

直接添加这个内容损失模块到计算内容距离的卷积层后面。这样每次向网络输入图像时，将在所需的层

上计算内容损失，且由于自动梯度，所有的梯度都将被计算。定义一个'forward'方法来计算内容损失，

然后返回这层的输入。计算的损耗被保存为模块的一个参数。

```
# 风格转换相关
class ContentLoss(nn.Module):
    def __init__(self, target,):
        super(ContentLoss, self).__init__()
        # 将目标内容从用于动态计算梯度的树中“分离”出来
        self.target = target.detach()

    def forward(self, input):
        self.loss = F.mse_loss(input, self.target)
        return input
```

风格损失(Style Loss):

风格损失模块的实现与内容损失模块类似。它将作为一个网络中的透明的层来计算该层的风格损失。为了计算风格损失，需要计算格拉姆矩阵G_{XL}。(格拉姆矩阵是一个给定的矩阵乘以它的转置矩阵的结 果)。在这个里，给定的矩阵是层L的特征图F_{XL}的变化版本。F_{XL}重构为$\hat{F}_{XL}$，一个$K \times N$矩阵，其中K是层L的特征图的个数，N是任意向量化的特征图F_{XL}^K的长度。例如，$\hat{F}_{XL}$的第一行对应于第一个向量化的特征图F_{XL}^1。最后，格拉姆矩阵必须标准化，即每个元素除以矩阵中元素的总数。这种归一化 是为了缓解$\hat{F}_{XL}$矩阵具有的较大的N维会在Gram矩阵中产生较大的值。这些较大的值将导致第一个层 (在池化层之前)在梯度下降时对结果影响偏大。风格特征就会往往位于网络的更深层，所以需要这一步规范化步骤。

```
def gram_matrix(input):
    a, b, c, d = input.size()
    features = input.view(a * b, c * d)
    G = torch.mm(features, features.t())  # 计算格拉姆矩阵
    # 矩阵标准化
    return G.div(a * b * c * d)

# 风格损失
class StyleLoss(nn.Module):

    def __init__(self, target_feature):
        super(StyleLoss, self).__init__()
        self.target = gram_matrix(target_feature).detach()

```

```
    def forward(self, input):
        G = gram_matrix(input)
        self.loss = F.mse_loss(G, self.target)
        return input
```

3. 模型引用

现在需要导入一个预先训练的神经网络。将使用文中使用的19层VGG网络。
PyTorch对VGG的实现是个分为两个子sequential模块:

- features (包含卷积和池化层)
- classifier (包含全连接层)

这里将使用"feature"模块，因为需要获取各个卷积层的输出来计算内容损失和风格损失。
一些层在训练和评估过程中有不同的行为，使用' .eval() '将层设置为评估模式。

```
cnn=models.vgg19(weights='VGG19_Weights.DEFAULT').features.to(solver.device).eval()
cnn_normalization_mean = torch.tensor([0.485, 0.456, 0.406]).to(solver.device)
cnn_normalization_std = torch.tensor([0.229, 0.224, 0.225]).to(solver.device)
```

```
# 创建一个模块来标准化输入的图像，使其能被放在nn.Sequential中
class Normalization(nn.Module):
    def __init__(self, mean, std):
        super(Normalization, self).__init__()
        # .view the mean and std to make them [C x 1 x 1] so that they can
        # directly work with image Tensor of shape [B x C x H x W].
        # B is batch size. C is number of channels. H is height and W is width.
        self.mean = mean.clone().detach().view(-1, 1, 1)
        self.std = std.clone().detach().view(-1, 1, 1)

    def forward(self, img):
        # normalize img
        return (img - self.mean) / self.std
```

'Sequential'模块包含子模块的有序列表。例如，'vgg19.features'包含sequence(Conv2d, ReLU, MaxPool2d, Conv2d, ReLU...)，按照正确的深度顺序。而如果需要在卷积层之后立即添加内容损失和样
式损失层，就必须创建一个新的'Sequential'模块，将内容损失和样式损失模块插入到合适位置。

```
# 使用的是nn.Sequential，则建立一个新的nn.Sequential
# 放入模块，会按顺序激活
model = nn.Sequential(normalization)

i = 0  # 每次看到一个conv层就增加
for layer in cnn.children():
    if isinstance(layer, nn.Conv2d):
        i += 1
        name = 'conv_{}'.format(i)
```

```
    elif isinstance(layer, nn.ReLU):
        name = 'relu_{}'.format(i)
        layer = nn.ReLU(inplace=False)
    elif isinstance(layer, nn.MaxPool2d):
        name = 'pool_{}'.format(i)
    elif isinstance(layer, nn.BatchNorm2d):
        name = 'bn_{}'.format(i)
    else:
        raise RuntimeError('Unrecognized layer
{}'.format(layer.__class__.__name__))
        model.add_module(name, layer)

    if name in content_layers:
        # 加入内容损失:
        target = model(content_img).detach()
        content_loss = ContentLoss(target)
        model.add_module("content_loss_{}".format(i), content_loss)
        content_losses.append(content_loss)

    if name in style_layers:
        # 加入风格损失:
        target_feature = model(style_img).detach()
        style_loss = StyleLoss(target_feature)
        model.add_module("style_loss_{}".format(i), style_loss)
        style_losses.append(style_loss)

for i in range(len(model) - 1, -1, -1):
    if isinstance(model[i], ContentLoss) or isinstance(model[i],
StyleLoss):
        break

model = model[:(i + 1)]
return model, style_losses, content_losses
```

4. 输入图像
5. 梯度下降

   使用L-BFGS算法进行梯度下降。与训练一个网络不同，此处希望训练输入图像，以尽量减少内容/风格
   的损失。
   创建一个PyTorch L-BFGS优化器。并将图像作为tensor传递给它进行优化。

```
def get_input_optimizer(input_img):
    optimizer = optim.LBFGS([input_img])
    return optimizer
```

6. 最后，定义一个执行neural transfer的函数。对于网络的每一次迭代，都会对其输入一个更新的输入并
   计算新的损失。将运行每个损失模块的backward方法来动态计算它们的梯度。优化器需要一个“闭包”函
   数来重新评估模块并返回损失。

```
def run_style_transfer(cnn, normalization_mean, normalization_std,
content_img, style_img, input_img, num_steps=300,
style_weight=1000000, content_weight=1)
# 详见ImgProc.py
```

3. 总结

在程序开发过程中遇到了很多问题，包括调整图像算法的各项参数，通过不断的调试，终于获得了一个较为满意的交互效果，通过对各种类的设计，对程序设计面向对象编程的理念有了更深的理解。之后可以加入更多的功能，对图像处理软件进行进一步的完善。

对于风格传递功能的实现上，总结了如下：

风格传递是基于深度神经网络生成艺术图像的系统。这种方法使用两个图像，内容图像和样式图像。它从内容图像中提取结构特征，而从风格图像中提取风格特征。为了获得这些图像的内容特征，可以使用网络中较深层的特征图。对于风格特征，在每个网络层的过滤器的回应上建立一个特征空间，其包含了不同过滤器回应之间的关联。

参考文献：

1. A Neural Algorithm of Artistic Style
2. Image Style Transfer Using Convolutional Neural Networks