计算摄影学课程设计

Image Colorization

3180105481 王泳淇

2021 年 6 月 25 日

目录

1 引言

1.1 项目介绍

图像上色（Image Colorization）指的是给黑白照片添加合理的颜色，使之变为有真实感的彩色照片。图像上色需要将像素一维的亮度值映射到多维的色彩，这是一个病态问题。通常有两种方法解决该问题：一是用户给定引导颜色上色，二是数据驱动的方式自动上色。

用户引导上色最著名的图像着色工作之一是“使用优化进行着色”[2]。用户仅对带有少量颜色涂鸦的图像进行注释，颜色会在空间中自动传播以生成完全彩色的图像。数据驱动的方法在很大程度上取决于训练数据集中的颜色先验。最近的一些工作试图使用卷积神经网络端到端地解决这个问题。

而在将黑白图像上色扩展到黑白视频上色时，如何处理单帧上色带来的不连续性，也是该问题的重要挑战之一。除了传统的使用帧与帧之间的光流进行约束外，最近也出现了一些使用卷积神经网络处理视频的时间连续性的工作 [4]。

本次课程设计中，我实现了两种传统方法进行图像上色，包括使用用户输入的颜色涂鸦引导进行上色 [2]，和将彩色图像的颜色转移到黑白图像上 [1]；前者提供了程序内鼠标绘制引导图和从外部读入引导图两种交互方式，后者提供了鼠标选取区域和从外部文件读入区域两种交互方式。我还使用了深度学习方法进行图像上色 [3]，测试了网络上在 ImageNet 数据集上训练的模型，和我自己在 MSCOCO 17 数据集上训练的模型。在讲图像上色扩展到视频上色的时候，我使用了 “Deep-Video-Prior” [4] 这一项目的开源代码，来增强上色视频的时间连续性。最后，我使用 PyQt5 编写了一个图形界面，将上述所有功能整合到一个图形化程序中。

总结起来，我实现的功能如下：

- 使用用户涂鸦引导进行上色，提供鼠标交互和外部读入文件两种交互；
- 将颜色从彩色图转移到灰度图，提供鼠标选取区域和外部读入区域文件两种交互；
- 使用卷积神经网络进行上色，测试了 ImageNet 上预训练的模型和我自己在 MSCOCO 上训练的模型
- 使用 Deep-Video-Prior 的开源代码，增强视频上色的连续性；
- 基于 PyQt5 的图形化界面；

1.2 项目环境

本次项目在 Windows 10 和 Ubuntu 16.04 环境下经过测试，在 Python 包依赖完整的情况下均能正确运行。

python 环境依赖如下：

- Python 3.8.4
- opencv-python 4.5.1
- numpy 1.19.5
- scipy 1.6.3

- scikit-learn 0.24.2
- scikit-image 0.18.1
- tensorflow 2.5.0
- pyqt5 5.15.4

要使用 deep-video-prior 进行视频时间连续性处理,请使用该目录下的 `environment.yml` 建立 conda 虚拟环境，并安装相关依赖。

2 方法概述

2.1 用户引导上色

本次项目中，我实现的用户引导上色基于"colorization using optimization"[2] 这篇文章。其方法是，用户在黑白图像上添加一些彩色的涂鸦，通过优化算法将这些涂鸦的颜色扩展到整张图象，从而获得良好的上色结果。

2.1.1 原理

该方法基于优化的方法实现，在 YUV 空间（实际代码中为 YIQ 空间）上对图像进行操作。其基本假设是，若一个像素和其邻域内像素的亮度相似，则其理应属于同一块区域，其颜色应该也是相近的。在这一假设的基础上，论文中提出的优化目标函数为：

$$J(U)=\sum_{\mathbf{r}}\left(U(\mathbf{r})-\sum_{\mathbf{s}\in N(\mathbf{r})}w_{\mathbf{rs}}U(\mathbf{s})\right)^2 \tag{1}$$

公式 (1) 中对每个像素和其周围像素的加权平均求距离（欧式距离的平方），再将这些距离进行求和。权重的表达式为：

$$w_{\mathbf{rs}}\propto e^{-(Y(\mathbf{r})-Y(\mathbf{s}))^2/2\sigma_{\mathbf{r}}^2} \tag{2}$$

从权重的表达式可以看出，当两个像素的亮度越接近时，权重越大，在加权平均中占比越高。

令公式 (1) 对 U 求导数，可以看出，当每个像素的色彩值满足：

$$U(\mathrm{r})-\sum_{\mathrm{s}\in N(\mathbf{r})}w_{\mathrm{rs}}U(\mathrm{s})=0 \tag{3}$$

目标函数取极小值。由此，可以根据上述条件构造线性方程组。对于线性方程组的第 i 个方程 ($i\in[0,m\times n-1]$)，若像素 p_i 为未上色的区域，则第 i 个方程即为公式 (3)。反之，若 i 为用户添加引导的区域，则第 i 个方程为：

$$U\left(p_i\right)=U_i \tag{4}$$

其中，U_i 为用户在像素 p_i 处添加的引导色。对于重上色的情况，U_i 为像素 p_i 本来的颜色。

上述线性方程组是稀疏的线性方程组，我们可以调用 scipy 中的相关函数，方便地对其进行求解，解得整张图像的 U 通道色彩。用同样的方法，可以解出图像的 V 通道色彩。

2.1.2 实现流程和部分代码

根据上述基本原理，我实现该算法的流程如下：

1. 将图像从 RGB 空间转换到 YIQ 空间；
2. 构建线性方程组；

3. 使用 scipy 的 linalg 库求解上述方程组，获得图像的 U 值；使用同样的方法获得图像的 V 值；

4. 将图像转换回 RGB 空间

RGB 和 YIQ 空间之间的转换有固定的公式，只需依照公式实现函数，即可实现空间之间的转换。

```
def rgb2yiq(rgb):
    rgb = rgb / 255.0
    y = np.clip(np.dot(rgb, np.array([0.30, 0.59, 0.11])), 0, 1)
    i = 0.74 * (rgb[:, :, 0] - y) - 0.27 * (rgb[:, :, 2] - y)
    q = 0.48 * (rgb[:, :, 0] - y) + 0.41 * (rgb[:, :, 2] - y)
    yiq = rgb
    yiq[..., 0] = y
    yiq[..., 1] = i
    yiq[..., 2] = q
    return yiq

def yiq2rgb(yiq):
    r = np.clip(np.dot(yiq, np.array([1.0,  0.9468822170900693,  0.6235565819861433]))
    , 0, 1)
    g = np.clip(np.dot(yiq, np.array([1.0, -0.27478764629897834, -0.6356910791873801])
    ), 0, 1)
    b = np.clip(np.dot(yiq, np.array([1.0, -1.1085450346420322,  1.7090069284064666]))
    , 0, 1)
    rgb = yiq
    rgb[:, :, 0] = r
    rgb[:, :, 1] = g
    rgb[:, :, 2] = b
    out = np.clip(rgb, 0.0, 1.0) * 255.0
    out = out.astype(np.uint8)
    return out
```

构建线性方程组的过程，首先根据像素及其邻域内像素的亮度计算出权重，再根据权重计算出方程组的系数矩阵 W。再根据用户选择上色的区域和重上色的区域，修改对应位置的方程组系数。最后进行求解。整个这个过程的代码如下：

```
def get_neighbors(img, p, d=2):
    y = p[0]
```

```
    x = p[1]
    left = max(0, x - d)
    right = min(img.shape[1] - 1, x + d)
    top = max(0, y - d)
    bottom = min(img.shape[0] - 1, y + d)
    neighbors = []
    for i in range(top, bottom + 1):
        for j in range(left, right + 1):
            if not (i == y and j == x):
                neighbors.append((i, j))

    return neighbors

img_lum = np.array(img[:, :, 0], dtype=np.float64)

def get_weights(img, p):
    neighbors = get_neighbors(img, p)

    neighbor_idx = [cal_index(img, p) for p in neighbors]
    window_lum = [img_lum[p[0], p[1]] for p in neighbors]
    window_lum.append(img_lum[p[0], p[1]])
    neighbor_std = np.std(window_lum)

    sigma = (neighbor_std ** 2) * 0.6
    mgv = min((window_lum - img_lum[p[0], p[1]]) ** 2)
    sigma = max(-mgv / np.log(0.01), sigma)
    sigma = max(0.00002, sigma)

    window_size = len(window_lum)
    window_lum = window_lum[0:window_size - 1]
    window_vals = [np.exp(-1 * (np.square(pixel - img_lum[p[0], p[1]])) / sigma) for
    pixel in window_lum]

    return window_vals, neighbor_idx

```

```
def get_weight_matrix(img):
    n, m = img.shape[0], img.shape[1]
    weight_matrix = sparse.lil_matrix((n * m, n * m))

    for i in range(n):
        for j in range(m):
            # if (colored_dict.get(cal_index(img, (i, j)), False)):
            neighbor_weights, neighbor_idx = get_weights(img, [i, j])
            weight_matrix[cal_index(img, [i, j]), neighbor_idx] = -1 * np.asarray(
    neighbor_weights)

    weight_matrix = normalize(weight_matrix, norm='l1', axis=1).tolil()
    weight_matrix[np.arange(n * m), np.arange(n * m)] = 1.
    return weight_matrix

weight_matrix = get_weight_matrix(img)
print("finish calculating matrix")

weight_matrix = weight_matrix.tocsc()
start = time.time()

for p in list(colored_indices):
    weight_matrix[p] = sparse.csr_matrix(([1.0], ([0], [p])), shape=(1, n*m))
for p in list(white_indices):
    weight_matrix[p] = sparse.csr_matrix(([1.0], ([0], [p])), shape=(1, n*m))

b1 = np.zeros(m * n)
b2 = np.zeros(m * n)
b1[colored_indices] = sketch[:, :, 1].flatten()[colored_indices]
b2[colored_indices] = sketch[:, :, 2].flatten()[colored_indices]
b1[white_indices] = img[:, :, 1].flatten()[white_indices]
b2[white_indices] = img[:, :, 2].flatten()[white_indices]

x1 = sparse.linalg.spsolve(weight_matrix, b1)
x2 = sparse.linalg.spsolve(weight_matrix, b2)
```

2.2 转移上色

本次项目中，我实现的转移上色基于”transferring color to grayscale image”[1] 这篇文章。该方法使用一张用于参考的彩色图像和一张待上色的黑白图像，通过选定一些区域，将参考图像区域中的颜色转移到黑白图像的对应区域中，再将黑白图像中的颜色扩展到整张图像上。

2.2.1 原理

该方法基于的基本假设是，参考图像和待上色图像中一组对应的区域应该属于类似的物体，因此在颜色转移的过程中，对于待上色区域中的每个像素，在参考区域中，找到亮度和邻域信息最为接近的像素，将其色彩拷贝过来。

而对于待上色图像中已上色区域和剩余的区域，该方法假设具有相似纹理（即区域内亮度的距离）的部分应该属于相同的颜色，因此在颜色扩展的时候将待上色部分的纹理和已上色部分的纹理进行比较，选取纹理最接近的部分，并拷贝其颜色。

2.2.2 实现流程和部分代码

根据原理实现具体流程，主要包括区域颜色转移和颜色扩展两部分。

1. 对于一组选定的区域，首先将参考图像区域的亮度（LAB 空间）进行线性变换，变换到待上色区域的亮度区间。然后，对于待上色区域的每个像素，使用 jittered sampling 的方法在参考区域内选择亮度和纹理（权重各为 0.5）最接近的像素，将其颜色值拷贝过来；

2. 对于其他区域，以 5×5 大小的 patch 为单位，比较 patch 和已上色区域的亮度的欧氏距离（即纹理的相似性），选取纹理最接近的区域，拷贝其颜色。

单个区域上色部分代码如下。其中，亮度信息和邻域信息的权重各为 0.5。

```
def colorize_swatch(img, ref):
    ref = cv2.cvtColor(ref, cv2.COLOR_BGR2Lab)

    ref = ref.astype(np.float64)
    ref_lum = ref[:, :, 0]

    # luminance remapping
    ref_lum_mean = np.mean(ref_lum)
    img_mean = np.mean(img)
```

```
    ref_lum_std = np.std(ref_lum)
    img_std = np.std(img)

    ref_lum = (ref_lum - ref_lum_mean) * img_std / ref_lum_std + img_mean

    # calculate neighborhood statistics
    neighborhood_size = 5
    ref_stat = generic_filter(ref_lum, np.std, neighborhood_size)
    img_stat = generic_filter(img, np.std, neighborhood_size)

    # jittered sample
    window_size = 5
    grid_num_x = int(ref.shape[1] / window_size)
    grid_num_y = int(ref.shape[0] / window_size)

    sampled_pos = []
    sampled_pixel = []
    sampled_neighbor_stat = []

    for i in range(grid_num_y):
        for j in range(grid_num_x):
            pos_y = randint(i * window_size, (i + 1) * window_size - 1)
            pos_x = randint(j * window_size, (j + 1) * window_size - 1)
            sampled_pos.append((pos_y, pos_x))
            sampled_pixel.append(ref_lum[pos_y][pos_x])
            sampled_neighbor_stat.append(ref_stat[pos_y][pos_x])

    # find best match
    output = np.full((img.shape[0], img.shape[1], 3), 128, dtype=np.float64)
    output[:, :, 0] = img

    weight_lum = 0.5
    weight_stat = 1. - weight_lum
    for i in range(output.shape[0]):
        for j in range(output.shape[1]):
```

```
            weighted_sum = (weight_lum * np.square(sampled_pixel - img[i][j])) + \
                           (weight_stat * np.square(sampled_neighbor_stat - img_stat[i][j]))
            match_index = np.argmin(weighted_sum)
            [ref_y, ref_x] = sampled_pos[match_index]
            output[i][j][1] = ref[ref_y][ref_x][1]
            output[i][j][2] = ref[ref_y][ref_x][2]

    output = output.astype(np.uint8)
    output_bgr = cv2.cvtColor(output, cv2.COLOR_LAB2BGR)

    return output, output_bgr
```

整体上色代码如下:

```
def colorize(img, ref, img_swatches: list, ref_swatches: list):
    assert len(img_swatches) == len(ref_swatches)
    mask = np.zeros(img.shape, dtype=np.int)
    swatch_num = len(img_swatches)
    output = np.full((img.shape[0], img.shape[1], 3), 128, dtype=np.uint8)
    output[:, :, 0] = img

    d = 2
    colorized_swatches = []

    for idx in range(0, swatch_num):
        img_region = img_swatches[idx]
        ref_region = ref_swatches[idx]
        # assert img_region is RectRegion and ref_region is RectRegion
        img_swatch = img[img_region.y_start:img_region.y_end + 1, img_region.x_start:img_region.x_end + 1]
        ref_swatch = ref[ref_region.y_start:ref_region.y_end + 1, ref_region.x_start:ref_region.x_end + 1]
        output_swatch = output[img_region.y_start:img_region.y_end + 1, img_region.x_start:img_region.x_end + 1]
        output_swatch[:, :, :], _ = colorize_swatch(img_swatch, ref_swatch)
```

```
        mask[img_region.y_start:img_region.y_end + 1, img_region.x_start:img_region.
    x_end + 1] = 1
        colorized_swatches.append(cv2.copyMakeBorder(output_swatch, d * 2, d * 2,
                                                     d * 2, d * 2, cv2.BORDER_REFLECT)
    )

    # expand color to the rest part
    output = cv2.copyMakeBorder(output, d * 2, d * 2,
                                d * 2, d * 2, cv2.BORDER_REFLECT)
    mask = cv2.copyMakeBorder(output, d * 2, d * 2,
                              d * 2, d * 2, cv2.BORDER_REFLECT)

    for i in range(2 * d, output.shape[0], 2 * d):
        if i >= output.shape[0] - d:
            continue
        for j in range(2 * d, output.shape[1], 2 * d):
            if j >= output.shape[1] - d:
                continue
            if np.sum(mask[i-d:i+d, j-d:j+d]) == np.square(2 * d + 1):
                continue
            patch = np.copy(output[i-d:i+d, j-d:j+d])
            patch = patch.astype(np.float64)
            error = 1e8

            for swatch in colorized_swatches:
                for m in range(2 * d, swatch.shape[0], 2 * d):
                    if m >= swatch.shape[0] - d:
                        continue
                    for n in range(2 * d, swatch.shape[1], 2 * d):
                        if n >= swatch.shape[1] - d:
                            continue
                        swatch_patch = np.copy(swatch[m-d:m+d, n-d:n+d])
                        swatch_patch = swatch_patch.astype(np.float64)
                        distance = np.sum(np.square(swatch_patch - patch))
                        if (distance < error):
```

```
                            error = distance
                            output[i-d:i+d, j-d:j+d, 1:] = swatch[m-d:m+d, n-d:n+d,
    1:]

    output = output[2*d:output.shape[0]-2*d, 2*d:output.shape[1]-2*d]
    output = cv2.cvtColor(output, cv2.COLOR_LAB2BGR)

    return output
```

2.3 深度学习上色

本次设计中，我使用 tensorflow v1 实现了"colorful image colorization"[3] 这篇代码中的卷积神经网络。其中，数据集的处理和模型训练的部分我使用了 Github 上的公开代码（https://github.com/nilboy/colorization-tf），而用于测试部分的网络结构及相关代码则是我自己依照参数进行实现，并加以修改以便封装到图形界面中。

我使用了该代码作者在 ImageNet 上训好的模型，并自己在 MSCOCO 17 数据集上自己训练了一个模型，并对这两个模型进行了测试。

2.3.1 原理

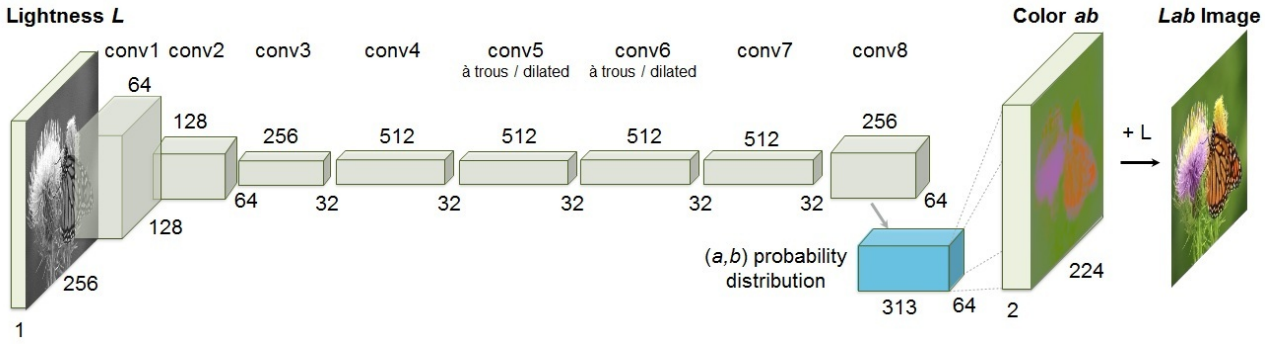

图 1: 网络结构

该网络的结构如图 1 所示，是一个全卷积的网络。网络处理 Lab 空间下的图像，输入是图像的 L 通道，输出图像的 a 和 b 通道。

这篇文章中，作者将 Lab 颜色空间的 ab 空间离散化为 313 个子区域，进而使用处理分类问题的方法来处理图像上色问题，使用分类问题常用的交叉熵 Loss 来训练这个网络。

由于训练集中低饱和度的图像数量很多，作者提出通过权重的方法使得不同类的像素的比例尽可能均衡。权重的计算方法是，统计训练图片 a，b 量化后的分布（只取距离最近的），对分布使用高斯函数进行平滑，平滑后的分布的倒数即可作为权重。这里作者在加权重和不加权重上做了个折中，可以理解为最后的损失有 2 部分，一部分是没有乘权重的，另一部分是乘了权重的。论文通过在计算的权重上加一个均匀分布来实现这种折中。

模型对于每个位置输出 313 个可能颜色的概率，用所得的概率对 313 个区间的中心加权求和可得到最终的颜色。论文中通过实验发现，如果直接用模型输出的概率计算颜色，颜色饱和度会比较低；如果只选取最大的概率的区间所表示的颜色，整个图片会不连续。作者通过下面这个公式来调整模型输出的概率，然后用改变后的概率对区间中心加权求和得到最终的颜色：

$$\frac{\exp(\log(\mathbf{z})/T)}{\sum_q \exp\left(\log\left(\mathbf{z}_q\right)/T\right)} \tag{5}$$

2.3.2 网络结构代码

本次设计中，用于训练模型的代码使用开源代码实现，我没有放在封装后的程序中。封装后的程序仅包含网络结构和测试用的代码。这里仅张贴这部分代码。

```
class Net(object):
    def __init__(self):
        tf.reset_default_graph()
        pass

    def conv2d(self, scope, input, kernel_size, stride=1, dilation=1, relu=True, wd
    =0.0):
        with tf.variable_scope(scope) as scope:
            kernel = _variable_with_weight_decay('weights', shape=kernel_size, stddev
    =5e-2, wd=wd)
            if (dilation == 1):
                conv = tf.nn.conv2d(input, kernel, [1, stride, stride, 1], padding='
    SAME')
            else:
                conv = tf.nn.atrous_conv2d(input, kernel, dilation, padding='SAME')
            biases = _variable('biases', kernel_size[3:], tf.constant_initializer(0.))
            bias = tf.nn.bias_add(conv,biases)
            if relu:
                conv1 = tf.nn.relu(bias)
            else:
```

```
                conv1 = bias
        return conv1

    def deconv2d(self, scope, input, kernel_size, stride=1, wd=0.0):
        pad_size = int((kernel_size[0] - 1) / 2)
        batch_size, height, width, in_channel = [int(i) for i in input.get_shape()]
        out_channel = kernel_size[3]
        kernel_size = [kernel_size[0], kernel_size[1], kernel_size[3], kernel_size[2]]
        output_shape = [batch_size, height * stride, width * stride, out_channel]
        with tf.variable_scope(scope) as scope:
            kernel = _variable_with_weight_decay('weights', shape=kernel_size, stddev
=5e-2, wd=wd)
            deconv = tf.nn.conv2d_transpose(input, kernel, output_shape, [1, stride,
stride, 1], padding='SAME')
            biases = _variable('biases', (out_channel), tf.constant_initializer(0.0))
            bias = tf.nn.bias_add(deconv, biases)
            deconv1 = tf.nn.relu(bias)
        return deconv1

    def batch_norm(self, scope, x):
        return tf.keras.layers.BatchNormalization(name=scope, center=True, scale=True,
 trainable=False)(x)

    def inference(self, input):
        # model 1
        temp_conv = self.conv2d('conv1', input, [3, 3, 1, 64], stride=1)
        temp_conv = self.conv2d('conv2', temp_conv, [3, 3, 64, 64], stride=2)
        temp_conv = self.batch_norm('bn_1', temp_conv)

        # model_2
        temp_conv = self.conv2d('conv3', temp_conv, [3, 3, 64, 128], stride=1)
        temp_conv = self.conv2d('conv4', temp_conv, [3, 3, 128, 128], stride=2)
        temp_conv = self.batch_norm('bn_2', temp_conv)

        # model_3
```

```
        temp_conv = self.conv2d('conv5', temp_conv, [3, 3, 128, 256], stride=1)
        temp_conv = self.conv2d('conv6', temp_conv, [3, 3, 256, 256], stride=1)
        temp_conv = self.conv2d('conv7', temp_conv, [3, 3, 256, 256], stride=2)
        temp_conv = self.batch_norm('bn_3', temp_conv)

        # model_4
        temp_conv = self.conv2d('conv8', temp_conv, [3, 3, 256, 512], stride=1)
        temp_conv = self.conv2d('conv9', temp_conv, [3, 3, 512, 512], stride=1)
        temp_conv = self.conv2d('conv10', temp_conv, [3, 3, 512, 512], stride=1)
        temp_conv = self.batch_norm('bn_4', temp_conv)

        # model_5
        temp_conv = self.conv2d('conv11', temp_conv, [3, 3, 512, 512], stride=1,
    dilation=2)
        temp_conv = self.conv2d('conv12', temp_conv, [3, 3, 512, 512], stride=1,
    dilation=2)
        temp_conv = self.conv2d('conv13', temp_conv, [3, 3, 512, 512], stride=1,
    dilation=2)
        temp_conv = self.batch_norm('bn_5', temp_conv)

        # model_6
        temp_conv = self.conv2d('conv14', temp_conv, [3, 3, 512, 512], stride=1,
    dilation=2)
        temp_conv = self.conv2d('conv15', temp_conv, [3, 3, 512, 512], stride=1,
    dilation=2)
        temp_conv = self.conv2d('conv16', temp_conv, [3, 3, 512, 512], stride=1,
    dilation=2)
        temp_conv = self.batch_norm('bn_6', temp_conv)

        # model_7
        temp_conv = self.conv2d('conv17', temp_conv, [3, 3, 512, 512], stride=1)
        temp_conv = self.conv2d('conv18', temp_conv, [3, 3, 512, 512], stride=1)
        temp_conv = self.conv2d('conv19', temp_conv, [3, 3, 512, 512], stride=1)
        temp_conv = self.batch_norm('bn_7', temp_conv)

```

```
        # model_8
        temp_conv = self.deconv2d('conv20', temp_conv, [4, 4, 512, 256], stride=2)
        temp_conv = self.conv2d('conv21', temp_conv, [3, 3, 256, 256], stride=1)
        temp_conv = self.conv2d('conv22', temp_conv, [3, 3, 256, 256], stride=1)

        # out
        temp_conv = self.conv2d('conv23', temp_conv, [1, 1, 256, 313], stride=1, relu=
    False)

        conv8_313 = temp_conv
        return conv8_313

def colorize(img, ckpt, pts):
    if (len(img.shape) == 3):
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    img = img[None, :, :, None]
    input = (img.astype(dtype=np.float32)) / 255. * 100. - 50.

    net = Net()
    output = net.inference(input)

    saver = tf.train.Saver()
    with tf.Session() as sess:
        saver.restore(sess, ckpt)
        output = sess.run(output)

    input = input + 50
    _, h, w, _ = input.shape
    input = input[0, :, :, :]

    output = output[0, :, :, :]
    output = output * 2.63

    def softmax(x):
        e_x = np.exp(x - np.expand_dims(np.max(x, axis=-1), axis=-1))
```

```
        return e_x / np.expand_dims(e_x.sum(axis=-1), axis=-1)

    output = softmax(output)
    cc = np.load(pts)

    output_ab = np.dot(output, cc)
    output_ab = resize(output_ab, (h, w))
    img = np.concatenate((input, output_ab), axis=-1)
    img = color.lab2rgb(img)
    img = img[:, :, ::-1]
    img = img * 255
    img = img.astype(np.uint8)

    return img
```

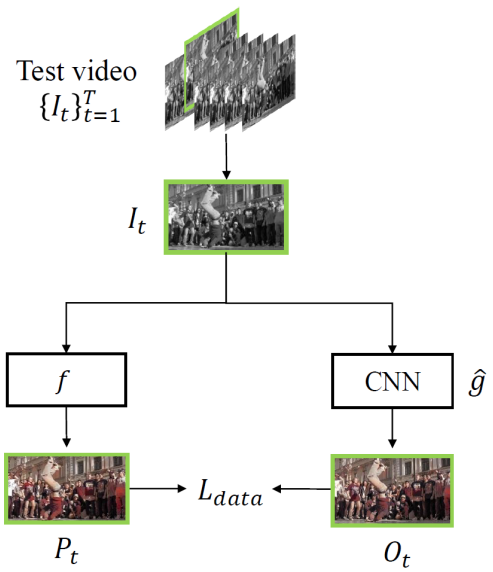

图 2: DVP 原理

2.4 视频上色的时序连续性

对于黑白视频的上色，采用逐帧上色的方法，可能会造成同一物体在不同帧上上色不同的时间不连续性，从而造成画面闪烁、跳动等现象。传统上一般使用光流等方法在训练时添加显式约束（正则项）。本次实验中我使用了名为 Deep-Video-Prior[4] 的方法，该方法没有显式使用光流等正则项约束，而是利

用了隐藏在视频映射关系之间的隐式先验和神经网络的性质来实现视频的时间连续性。本次设计中，我将 2.3 节中使用 CNN 对单帧图像上色部分和 Deep-Video-Prior 的开源代码相结合，实现了视频连续性上色。

2.4.1 原理

Deep-Video-Prior 的结构如图 2 所示。该方法使用一个 CNN 在给定的一组未处理视频和处理后的视频上进行训练，来学习从未处理视频到处理后视频的映射关系，并过滤掉颜色不一致、闪烁等时间不连续问题。

该方法基于的假设是，视频中出现的不连续、闪烁等现象，属于一种噪声，是一种高频信息；而未处理视频和处理后的视频的均一的、稳定的转换，则是一种低频信息。而比起高频信息，神经网络更倾向于学到那些低频信息。因此，只要通过限制神经网络的迭代次数，网络过拟合到闪烁等高频噪声前停止训练，就能有效获得时间连续的视频输出。

同时，对于图像上色等多模态问题（即同一个输入可以映射到多个输出），该方法使用称为 Iteratively Reweighted Training 的方法进行训练。以图像上色为例，若一个图像对应着两种模态，则可增加网络输出的通道数（如 2 个 RGB 图像 6 个通道），分别学习一个主模态和一个副模态。对于每次迭代中的每个像素，若训练图像的像素值更接近网络输出的主模态像素值，或者同时接近于主模态和副模态的像素值，则用该像素对主模态进行训练；否则，若训练值更接近网络输出的副模态，则用该像素对副模态进行训练。该方法是通过在训练时对图像增加一个 mask 实现的，能够有效处理多模态问题。

由于本方法使用的是开源的代码，这里略去代码实现的张贴。

2.5 图形界面

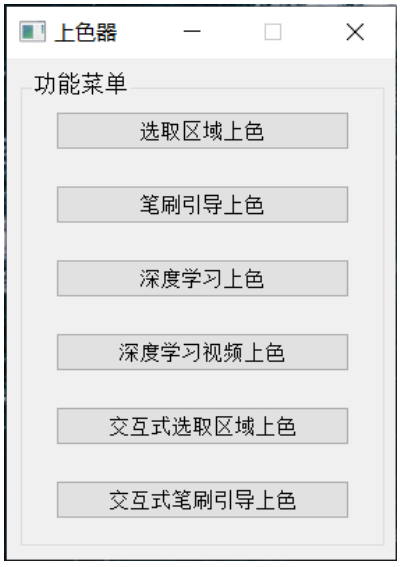

图 3: 主菜单

本次实验中，我使用 PyQt5 实现了一个图形界面，将上述图像上色方法封装在了一起。主菜单界面如图 3 所示，提供了一个统一的界面，可以点按不同的按钮打开不同的窗口，来实现不同的上色功能。

部分功能如选取区域上色、笔刷引导上色提供了鼠标交互式的交互界面。如图 4 所示，可以通过鼠标直接在图像上绘画来添加引导，可以控制笔画的颜色和粗细。

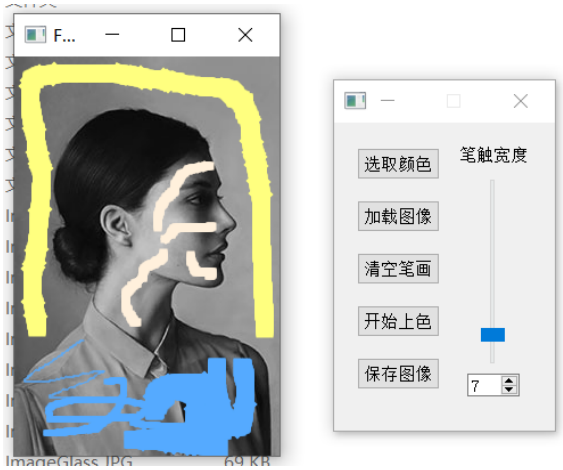

图 4: 交互式引导上色

选取区域上色的界面如图 5 所示，可以通过鼠标选定区域之后进行上色。

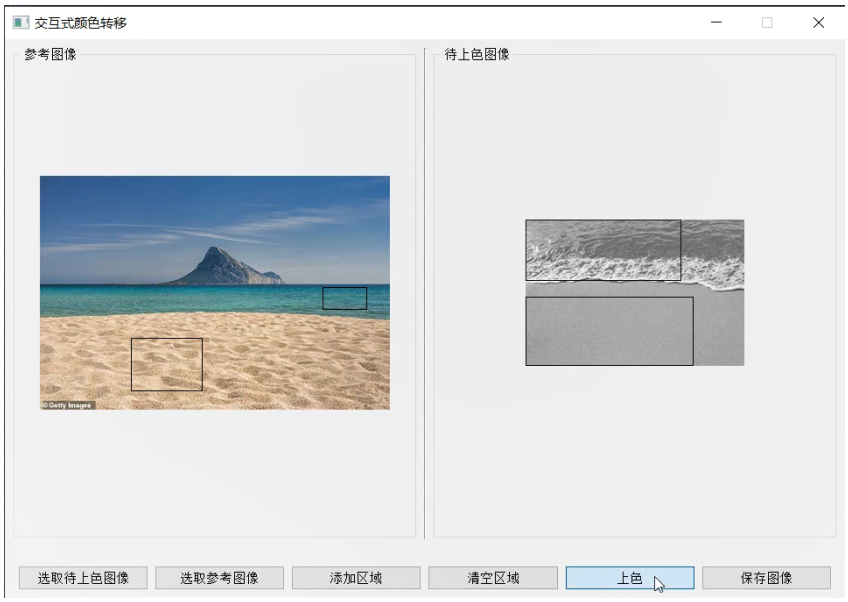

图 5: 交互式引导上色

图形界面中图像的效果主要借助 PyQt5 的图像功能。如使用 QPixmap 在图形界面中显示图像，使用 QPainter 在图像上进行绘制，使用 QMouseEvent 获得鼠标点击的位置。在上色过程中，只需要将前面实现的上色 api 绑定到按钮的槽函数中，即可实现图像上色。

3 测试和分析

3.1 用户引导上色

在测试过程中，用户引导上色表现鲁棒性较强，上色结果也较为准确。只要添加的涂鸦色彩引导合理，基本上上色结果是比较好的。

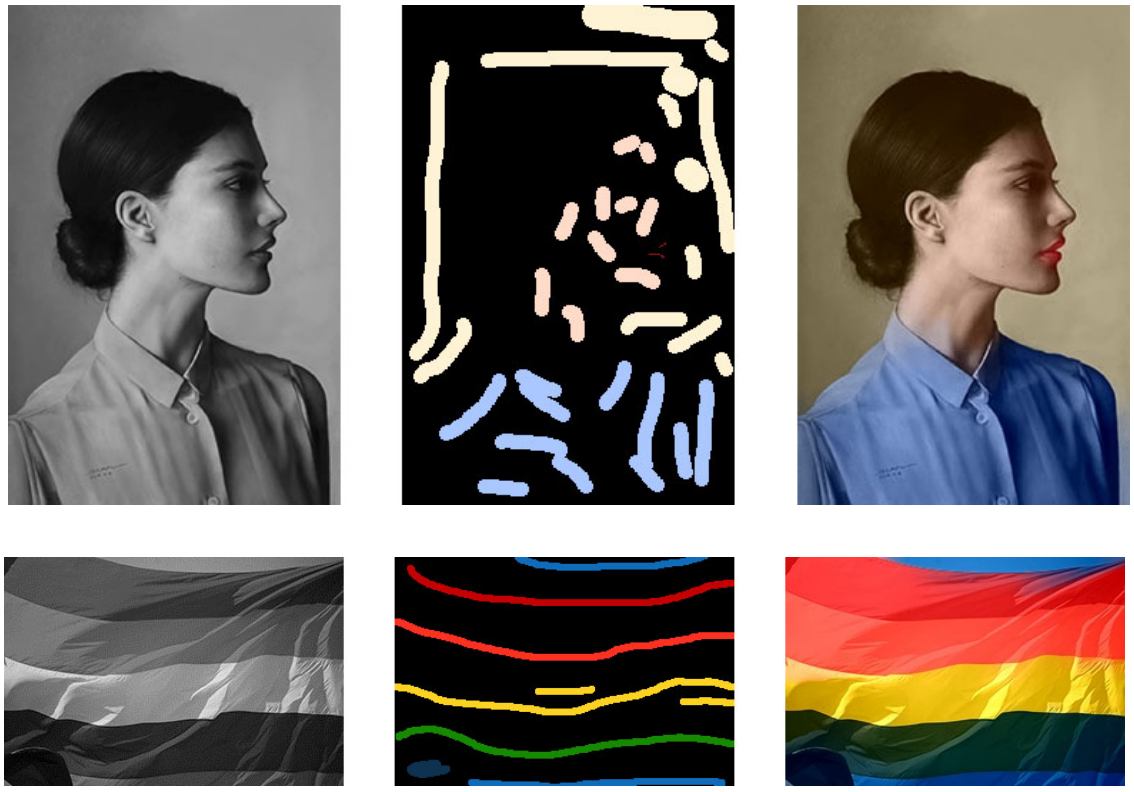

图 6: 用户引导上色

如图 6 所示，左侧为未上色的黑白图像，中间为用于上色的引导图，右侧为上色之后的图像。对于第一张人像，添加良好的引导后，可以使人像具有非常逼真的色彩；值得注意的是，在中间嘴唇的位置，引导图添加了两条红色细线，而在得到的上色结果上人的嘴唇也呈现了红色，效果良好。对于下方的旗帜，上色结果整体也较为准确，色块之间分割较为明确。

不过，受限于算法本身的因素，一些区域也出现了本该是清晰的区域上色结果混杂到一起的。如第一行中人像右下角的缝隙，本来应当和墙壁具有相同的颜色，但却错误地染上了衣服的颜色，而且和墙壁区间的连接位置较为模糊。第二行的气质上方红色和蓝色的交界处也是这样，本该是红色的位置被蓝色有所侵染。猜测可能是算法在计算像素颜色时，只考虑了邻域内颜色的影响；而当一块区域内添加的颜色引导较稀疏，距离较远时，距离引导较远的区域可能受到不属于同一块区域但距离较近的上色结果的影响。因此，该算法对引导添加的均匀性可能有一定要求。

我还测试了使用该算法进行重上色的结果，如图 7 所示。在想要保留原色的区域涂上白色引导，在想要重上色的区域涂上新的颜色，上色算法成功将图上的女性衣服从蓝色置换成了粉色。

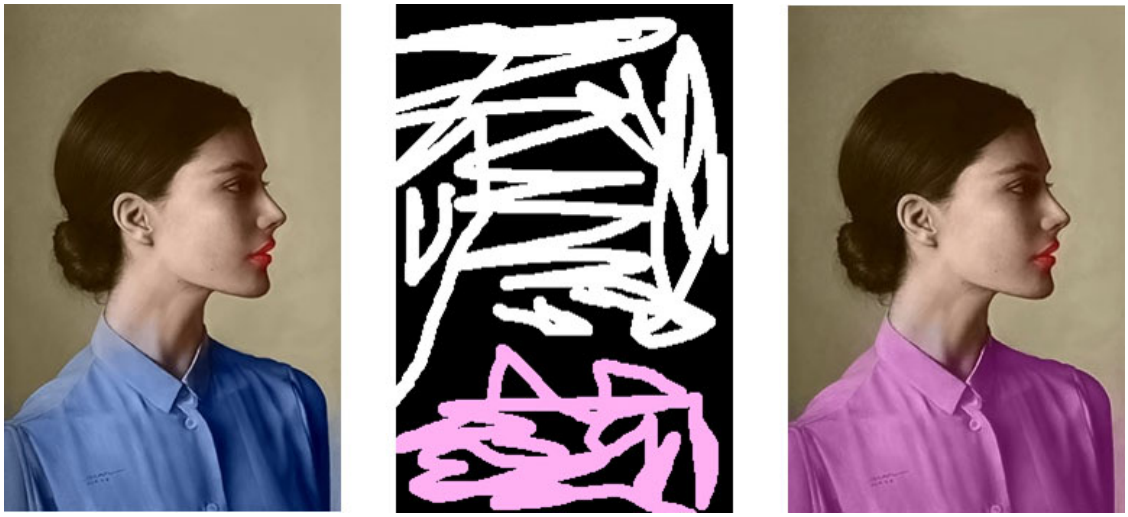

图 7: 用户引导重上色

3.2 颜色转移

在实验过程中，颜色转移算法表现有好有坏。图 8 是一些成功的样例。

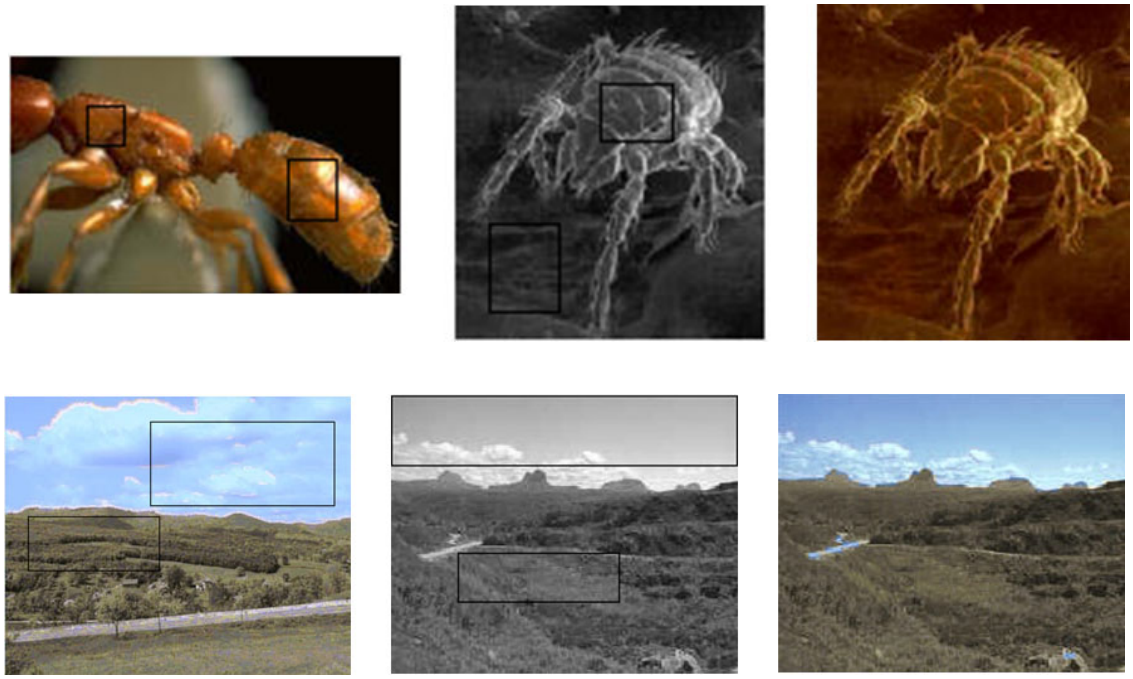

图 8: 转移颜色

可以看到，算法成功地将颜色在指定区域之间进行了转移，并扩展到整张图像上。

图 9 展示了使用全局上色失败而选取区域上色成功的一个案例。在参考图像上，天空的一部分区域和沙滩的一些区域亮度相近，导致在转移色彩时天空、沙滩的颜色混到了一起，海面和沙滩都是半黄半蓝的颜色，上色失败。而通过选取限制上色的区域，有效地限制了上色的区域，取得了较好的结果。

不过，仔细观察沙滩和海浪中间的接合处，可以看到有些不规则的蓝色方块状缝隙。这是由于在颜

色扩展的过程中使用 5×5 大小的方块，导致边缘无法做到像素级的精细，比较粗糙。

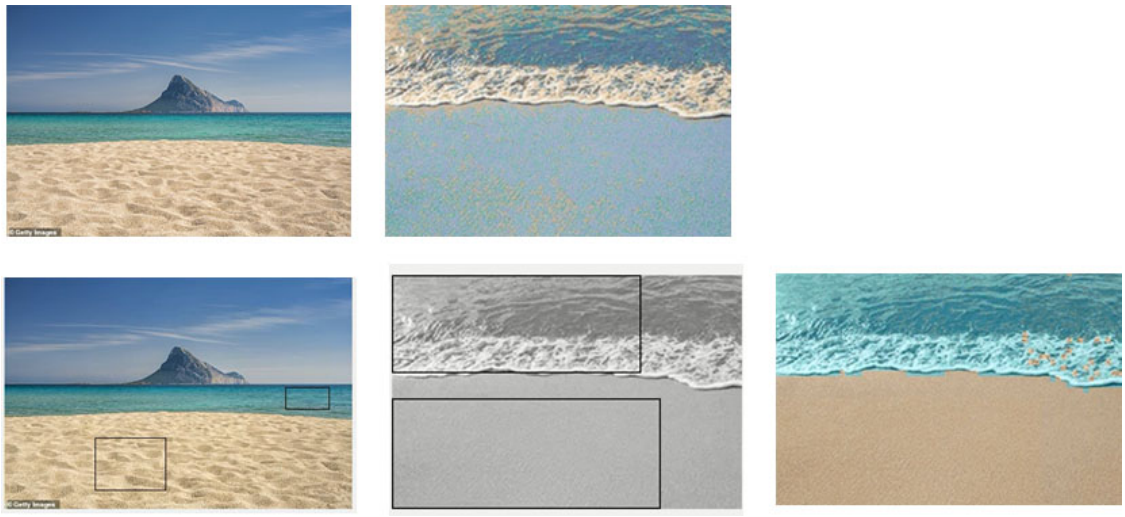

图 9: 转移上色：全局和区域

除此之外，在进行测试过程中，也出现了一些失败的情况。如图 10 所示；在第一行，由于选取上色的区域中道路和天空的纹理非常接近，导致在扩展颜色的时候天空的剩余部分错误地选取了道路的土黄色，导致上色失败。

在第 2 行中，画面左上角的海浪的纹理和其他部分海浪有所不同，更加接近于沙滩的纹理，因此在颜色扩展的过程中也出现了失败。这种情况和图 9 中的案例形成对比，说明区域的选取对上色结果有很重要的影响。

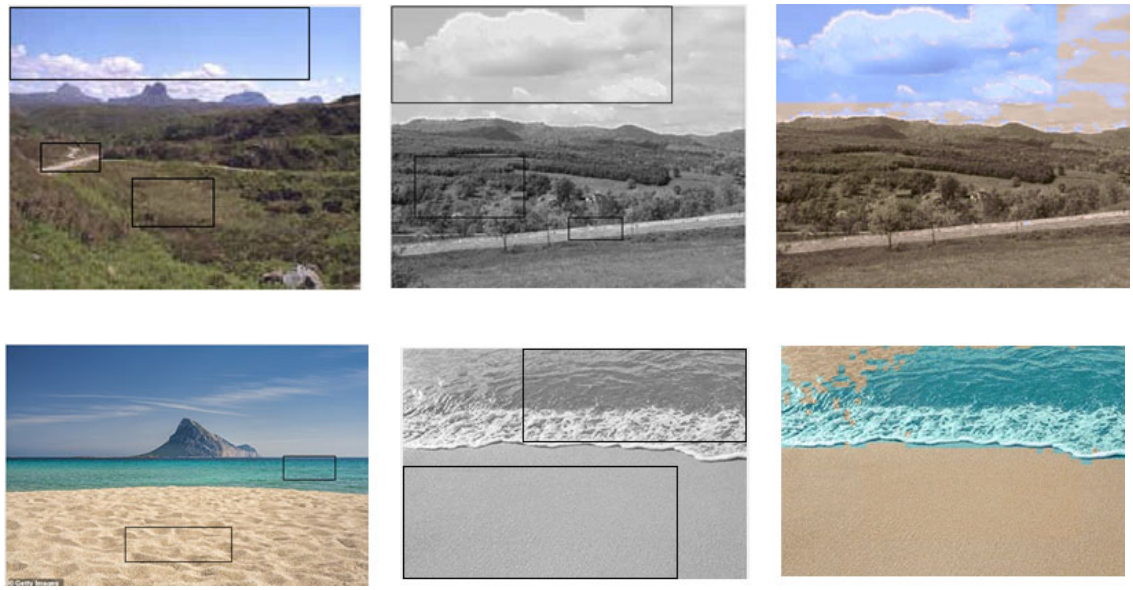

图 10: 转移上色：失败的情况

从上面的结果可以总结出，转移色彩虽然能取得一定的效果，但具有两大显著缺陷：

1. 在进行颜色扩展的时候，已经上色的区域中可能有多个区域和待上色部分纹理相近，这时不一定能选取正确的区域，从而造成一些 patch 的上色错误；

2. 由于颜色扩展时以 patch 为单位，上色区域的边界无法做到像素级的精细，会比较粗糙；

因此，该方法要求待上色图像区域之间有明显的纹理差异，且对上色区域的依赖性强，不是一个很鲁棒的算法。

3.3 深度学习上色

使用深度学习进行上色，我使用了 Github 上提供的 ImageNet 数据集上的预训练模型，还有我自己在 MSCOCO 17 数据集上训练的模型。其中 MSCOCO 17 上训练的模型我迭代了 129000 次，训练参数如下：

- weight_decay: 0.01
- learning_rate: 0.00000316
- moment: 0.9
- lr_decay: 0.316
- decay_steps: 210000

选取的部分测试结果如图 11 所示。每行左侧为未上色的原图，中间为 ImageNet 上的预训练模型的上色结果，右侧为我在 MSCOCO 上训练模型的上色结果。

可以看到，ImageNet 上训练的模型整体效果较好，上色接近真实，饱和度高，模式均匀。而 MSCOCO 上训练的模型效果较差，但也具一定的真实感。

以第一行为例，ImageNet 上训练的模型的上色结果天空的蓝色较均匀，饱和度较高；而 MSCOCO 上训练的模型输出的天空则发灰，蓝色和灰色混杂在一起，饱和度低，不够均匀。草坪靠近建筑物的地方也是 ImageNet 模型的输出更为准确。

在第二行中，ImageNet 模型输出的树林颜色饱和度更高，更加真实，天空颜色也较为准确；而 MSCOCO 模型靠近地平线的位置天空出现了奇怪的黄色，树木的饱和度也不高。

而在第三行的人像上色中，ImageNet 模型和 MSCOCO 模型输出结果更是大有参差。ImageNet 模型输出的人像肤色均匀、真实感强，且发色清晰；而 MSCOCO 模型输出的人像脸色完全还是灰白色的，手臂却出现不正常的金色，且右上方还出现了诡异的蓝色调。

从该分析可知，ImageNet 上预训练的模型明显优于我在 MSCOCO 上训练的模型。分析其原因，可能是由于我自己训练使用的数据集较小，且训练量不足。理想的训练量应该达到 350000 次迭代，而受限于时间，我只迭代了 129000 次。此外，ImageNet 数据集中图像量大、种类丰富全面，训练出来的模型适应性更强；而我使用的 MSCOCO 可能在实际训练过程中用到的图像偏少，涵盖范围小，导致效果偏差。

同时，我也尝试了使用这个模型进行图像重上色。如图 12 所示。上色结果整体比较接近原图像的色彩，真实感较强。不过颜色的饱和度相较于原图会偏低。

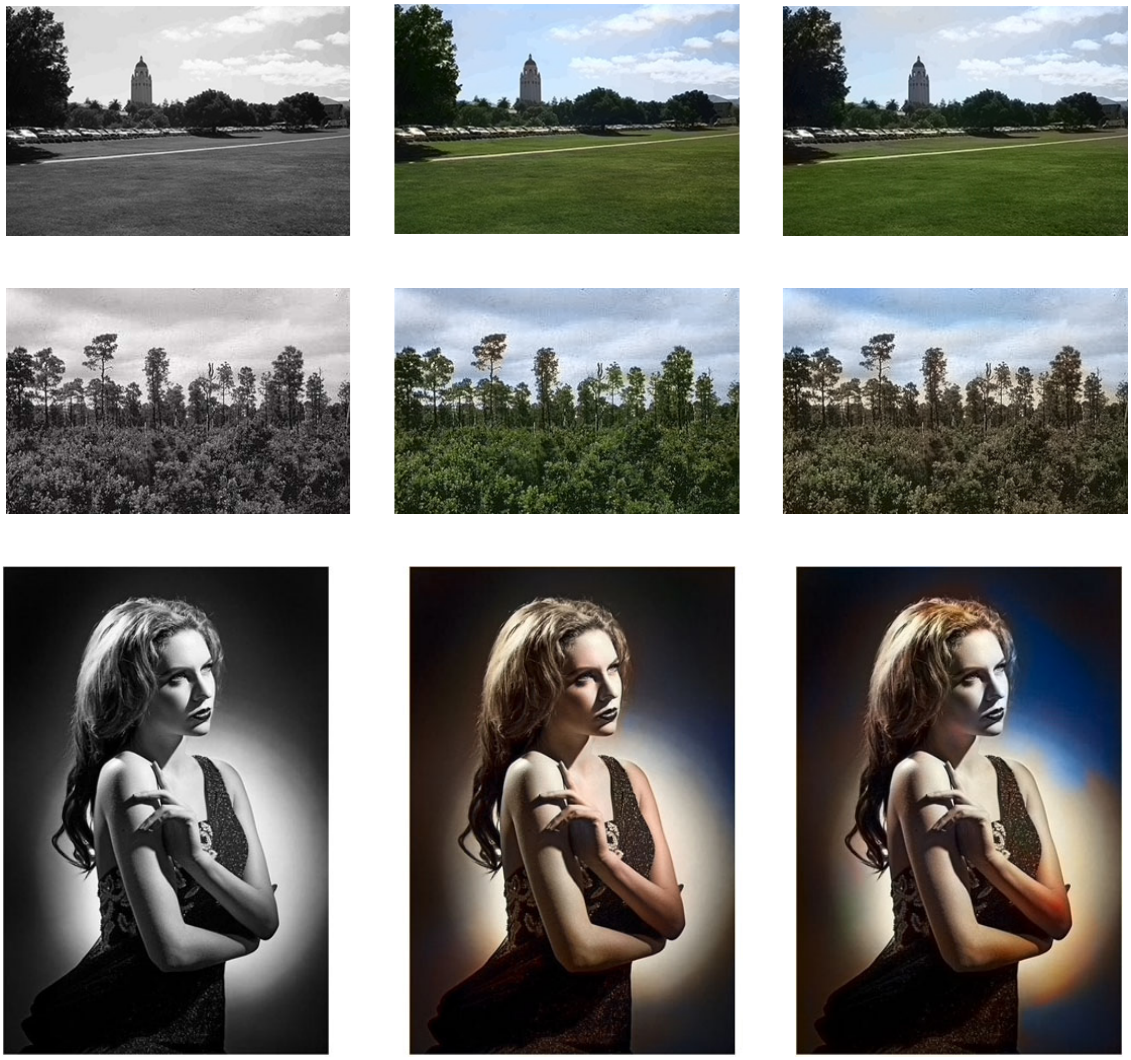

图 11: 神经网络上色对比

3.4 视频上色的时间连续性

本次项目中，我使用 Deep-Video-Prior 的开源代码，对单帧逐帧上色的视频进行了测试。由于单个视频处理结果较长，

报告中无法显示视频，但是我们可以观察一些帧的处理结果来观察处理效果。如图 13 所示，左侧为单帧上色结果，右图为 DVP 输出结果。可以重点观察左下角的石头，单帧上色时石头被染上了诡异的绿色，但仅有这一张石头具有不连续的绿色。而在处理后的视频中，石头的绿色消失了，恢复了比较正常的颜色。除此之外，背景山上的一部分植物本来显现出红色，在处理后的图像中红色变淡了。

然而，背景山上的植物显现出不正常的青绿色的部分，在修复中仍然没有被解决。这是由于在大部分帧中，这一块区域都被上成了这种不正常的青绿色。这说明如果初期的单帧上色结果太差，即使通过 DVP 进行修复，也是没有办法取得太好效果的。

在提交报告附带的文件中，我提供了经 DVP 处理前的视频和 DVP 处理后的视频。除了左下角石头的颜色变化外，若仔细观察，也可以看到背景中山上的植物的颜色在有些帧中发生了跳变，而在 DVP 输

图 12: 神经网络重上色

出的视频中这些跳变被修复了。因为植物的饱和度较低，跳变可能不很明显。但由于单个视频的处理时间较长，没有来得及做更好的样例了。

图 13: 通过 DVP 调整上色结果

4 小结

本次实验中，我实现了多种图像上色算法，并将其整合到一个图形界面中，还尝试了对视频上色进行连续性处理，最终完成了一个具有中等规模的程序。整个过程还是比较有挑战性的，结果也比较让人有收获感。

我所实现的几个算法中，我比较喜欢用户引导上色。这一算法取得的结果视觉效果好，且自由度较高，用户可以根据自己的喜好添加喜欢的颜色。我给这种方法提供了鼠标交互的 GUI 界面，使用起来也比较有趣味性。

通过这次课程设计，我对图像上色算法的理解大大加深了，也锻炼了一些工程开发的技能。可以说收获比较多。

展望该项目的未来，可以尝试使用一些更加新的、效果更好的深度学习网络。以及在 2021 年，谷歌最先使用了 Transformer 来完成图像上色这一任务，取得了相当好的结果。限于时间和个人能力，没有对这一方向进行探索，如果未来有时间，希望能在这一方面进一步拓展。

参考文献

[1] H. B. Kekre and S. D. Thepade, "Color Traits Transfer to Grayscale Images," 2008 First International Conference on Emerging Trends in Engineering and Technology, 2008, pp. 82-85, doi: 10.1109/ICETET.2008.107.

[2] Levin, Anat, Dani Lischinski, and Yair Weiss. "Colorization using optimization." ACM SIGGRAPH 2004 Papers. 2004. 689-694.

[3] Zhang, Richard, Phillip Isola, and Alexei A. Efros. "Colorful image colorization." European conference on computer vision. Springer, Cham, 2016.

[4] Lei, Chenyang, Yazhou Xing, and Qifeng Chen. "Blind video temporal consistency via deep video prior." Advances in Neural Information Processing Systems 33 (2020).