Assignment 0 Report

The assignment mainly concerned manipulating pixels of an image to achieve transition effects like change in brightness, contrast, replacing green screens with suitable background, color to grayscale and vice versa without using any image processing libraries.

Reading an Image File to Array

Images are read into an array of pixels using "cv2.imread" function. The flag “cv2.IMREAD_UNCHANGED” is used to preserve the nature of image and return correct dimensions. By default, this flag is set to “cv2.IMREAD_COLOR” which converts all images irrespective of grayscale and color to a 3D array.

If after reading the image, the array is 3-dimensional it implies that the image is a color image. cv2.imread converts each pixel to a 3 tuple containing BGR values. "cv2.cvtColor" is then used to change the pixels in the array to standard RGB representation.

If the array after reading is 2-dimensional, it corresponds to a grayscale image in which case we simply return the array without any modifications.

```
def img_to_array(filepath):
    img = cv2.imread(filepath,cv2.IMREAD_UNCHANGED)
    if len(img.shape) > 2:
        img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        return np.array(img_rgb)
    else:
        return np.array(img)
```

Write an array to image file

An array containing all pixels data can be converted to image using "cv2.imwrite" function. Similar to imread, imwrite expected values in BGR format in case of color images. Hence if our image currently contains data in RGB format, we must convert it to "BGR" format using "cv2.cvtColor".

```
def array_to_img(array,filepath):
    if len(array.shape) > 2:
        image_bgr = cv2.cvtColor(array, cv2.COLOR_RGB2BGR)
        cv2.imwrite(filepath,image_bgr)
    else:
        cv2.imwrite(filepath,array)
```

Change brightness of the image

Two methods have been implemented to change the brightness of the image, namely addition and multiplication.

Brightness change using multiplication

1. The image is first converted to pixels array

2. Change the data type from **uint8** to **uint16** since multiplication might exceed 255. If the data type is uint8 then it will be converted to a negative number giving weird results. Hence converting to uint16 will keep the product intact.

3. Run 2/3 nested for loops based on grayscale or color image and multiply each pixel by the factor.

4. Clip the values to ensure they are between 0 to 255. All values above 255 are clipped to 255.

```
def change_brightness_mult(img_path,factor):
    # Convert image to array
    pixels = img_to_array(img_path)
    # Change data type of array to avoid overflow
    pixels = pixels.astype(np.uint16)

    # Multiply each pixel by factor to change brightness
    if len(pixels.shape) >= 3:
        for i in range(pixels.shape[0]):
            for j in range(pixels.shape[1]):
                for k in range(pixels.shape[2]):
                    pixels[i][j][k] = pixels[i][j][k]*factor

    else:
        for i in range(pixels.shape[0]):
            for j in range(pixels.shape[1]):
                pixels[i][j] = pixels[i][j]*factor

    # Clip the values to ensure they are between 0 and 255
    pixels = np.clip(pixels,0,255)
    return pixels
```

Brightness change using addition

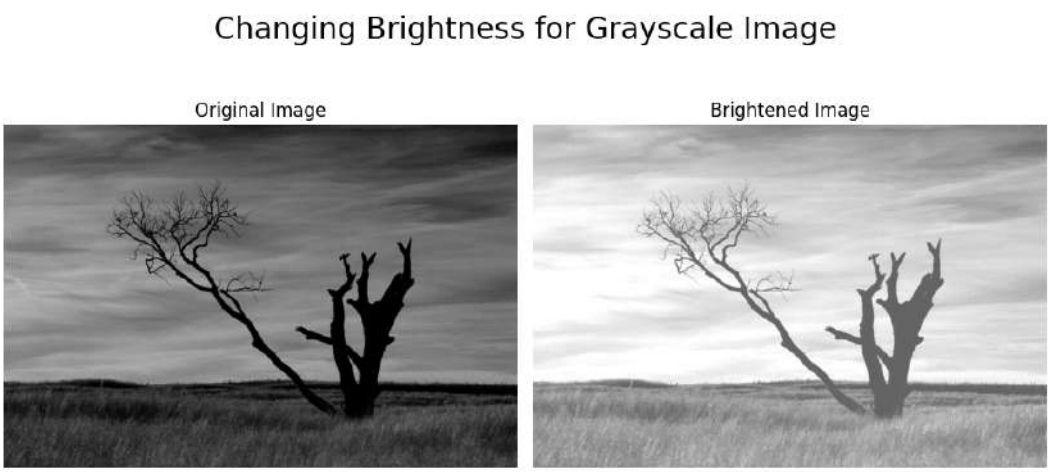
1. The image is first converted to pixels array

2. Change the data type from **uint8** to **uint16** since multiplication might exceed 255. If the data type is uint8 then it will be converted to a negative number giving weird results. Hence converting to uint16 will keep the product intact.

3. Run 2/3 nested for loops based on grayscale or color image and add RGB factors to RGB channels of the image.

4. Clip the values to ensure they are between 0 to 255. All values above 255 are clipped to 255.

```
def change_brightness_add(img_path,factor_red,factor_green,factor_blue):
    # Change image to array
    pixels = img_to_array(img_path)

    # Change datatype to avoid overflow
    pixels = pixels.astype(np.uint16)

    # Add values to RGB channels
    if len(pixels.shape) > 2:
        for i in range(pixels.shape[0]):
            for j in range(pixels.shape[1]):
                pixels[i][j][0] = pixels[i][j][0]+factor_red
                pixels[i][j][1] = pixels[i][j][1]+factor_green
                pixels[i][j][2] = pixels[i][j][2]+factor_blue
    else:
        for i in range(pixels.shape[0]):
            for j in range(pixels.shape[1]):
                pixels[i][j] = pixels[i][j]+factor_red

    # Clip to ensure pixel value is between 0 to 255
    pixels = np.clip(pixels,0,255)
    return pixels
```



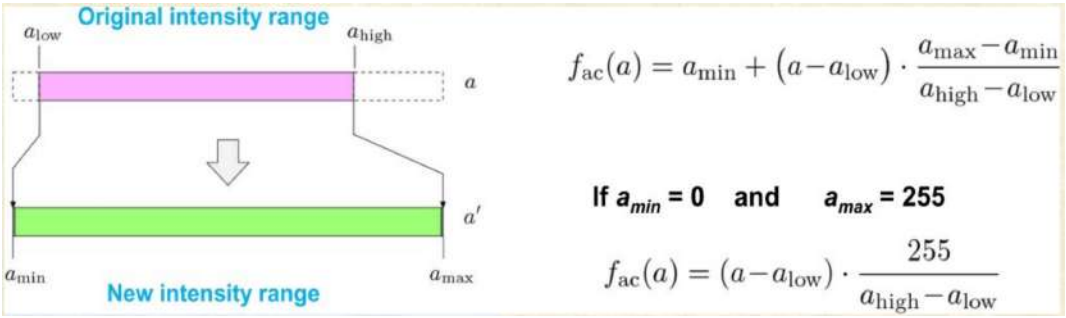
Observations

- **Proportional Adjustment**: Multiplication scales pixel values proportionally, maintaining the relative differences between pixel intensities. This preserves the contrast and the overall appearance of the image, resulting in a more natural and visually appealing effect.

Hence, multiplying pixel values by a scalar to change brightness generally yields better results compared to adding.

Change contrast of the image

We use the method of contrast stretching for adjusting the contrast of the image.



1. Extract red,green and blue channels from the image.

2. Compute minimum and maximum intensities for each channel which correspond to a_low and a_high in the above image.

3. Set min and max intensities as 0 and 255.

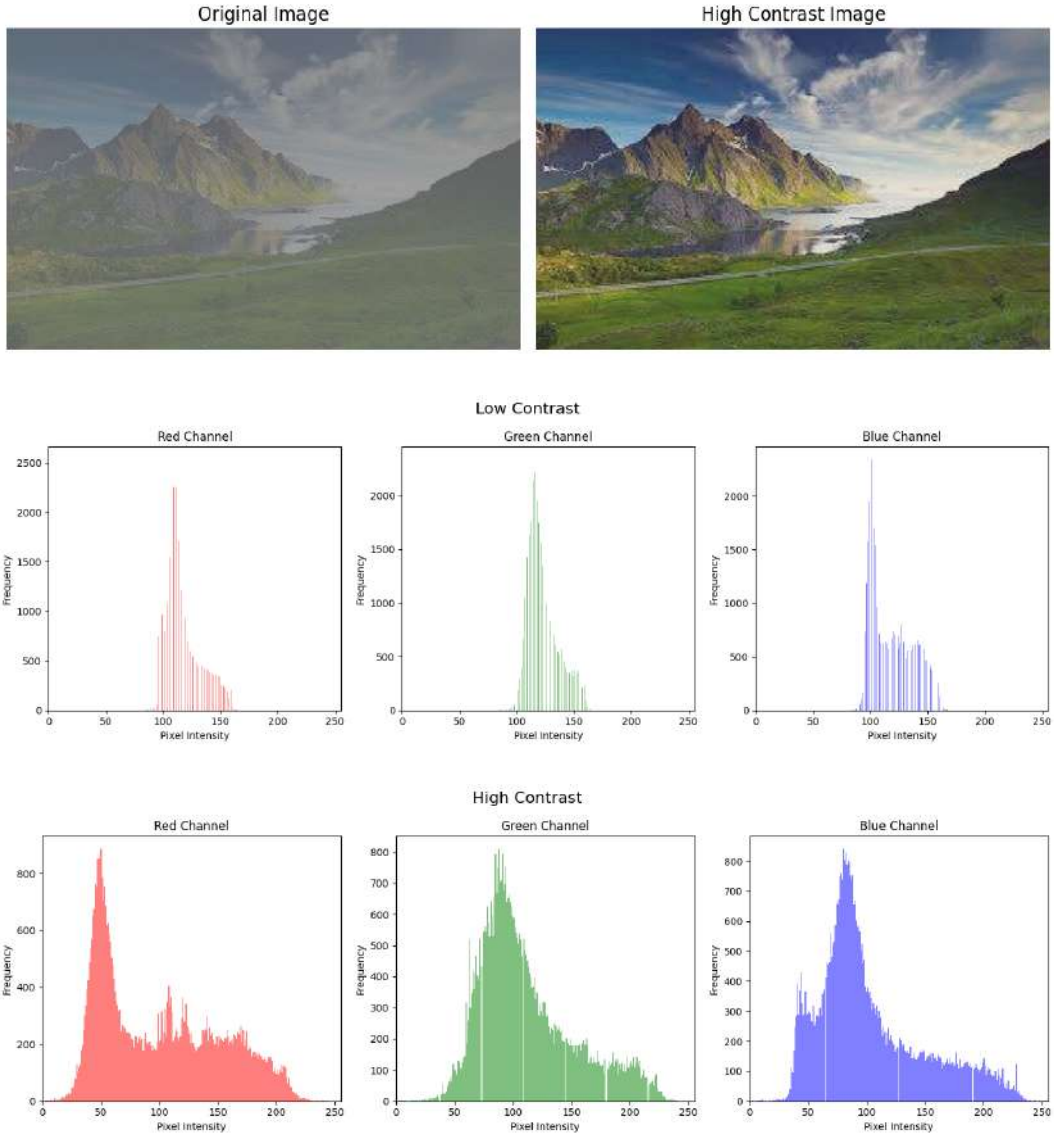
4. Adjust value of each pixel as shown in the formula above.

5. Clip the values to ensure they do not exceed 255.

6. Reconstruct the image by stacking these new RGB channels.

```
def change_contrast(img_path):
    # Convert image to array
    pixels = img_to_array(img_path)

    # Change datatype to avoid overflow
    pixels = pixels.astype(np.uint16)

    # Set minimum and maximum intensities
    a_max = 255
    a_min = 0

    if len(pixels.shape) == 2:
        min_intensity, max_intensity = pixels.min(), pixels.max()
        pixels = (pixels - min_intensity) * (a_max - a_min) / (max_intensity - min_in
        pixels_norm = np.clip(pixels, 0, 255).astype(np.uint8)
        reconstructed_img = pixels_norm

    else:
        # Extract RGB channels
        red_channel = pixels[:, :, 0]
        green_channel = pixels[:, :, 1]
        blue_channel = pixels[:, :, 2]

        # Compute minimum and maximum intensity for each channel
        red_min, red_max = red_channel.min(), red_channel.max()
        green_min, green_max = green_channel.min(), green_channel.max()
        blue_min, blue_max = blue_channel.min(), blue_channel.max()

        # Modify the pixels for contrast streching
        red_channel = (red_channel-red_min)*(a_max-a_min)/(red_max-red_min) + a_min
        green_channel = (green_channel-green_min)*(a_max-a_min)/(green_max-green_min)
        blue_channel = (blue_channel-blue_min)*(a_max-a_min)/(blue_max-blue_min) + a_

        # Clip the values to ensure they are between 0 and 255
        red_channel_norm = np.clip(red_channel, 0, 255).astype(np.uint8)
        green_channel_norm = np.clip(green_channel, 0, 255).astype(np.uint8)
        blue_channel_norm = np.clip(blue_channel, 0, 255).astype(np.uint8)
```

```
        # Reconstruct image using new RGB channels
        reconstructed_img = np.stack([red_channel_norm, green_channel_norm, blue_chan
    return reconstructed_img
```



As can be seen from the plots, the range from lowest intensity pixel and highest intensity pixel changes as we change the contrast. Lighter pixels get lighter and darker pixels get darker.

Change a color image to grayscale

Used the following formula:

Intensity_grayscale = alpha*(intensity_red) + beta*(intensity_green) + gamma*(intensity_blue)

where alpha+beta+gamma = 1

First alpha is randomly sampled from [0,1].

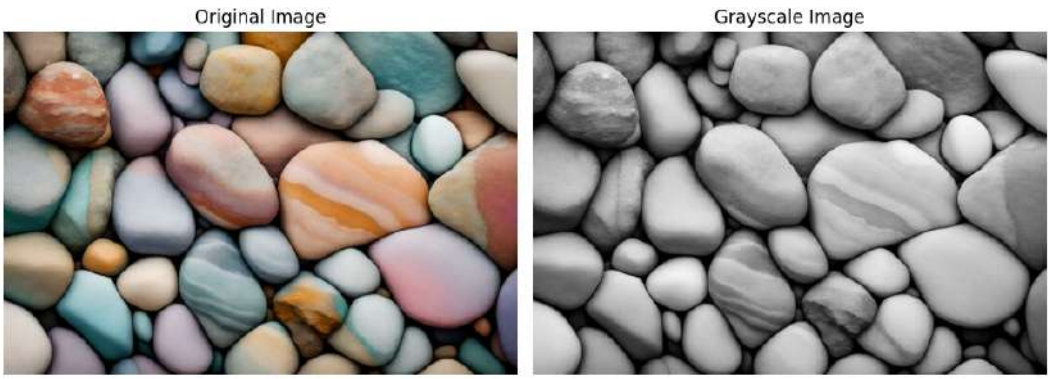
Then beta is sampled from [0,1-alpha]

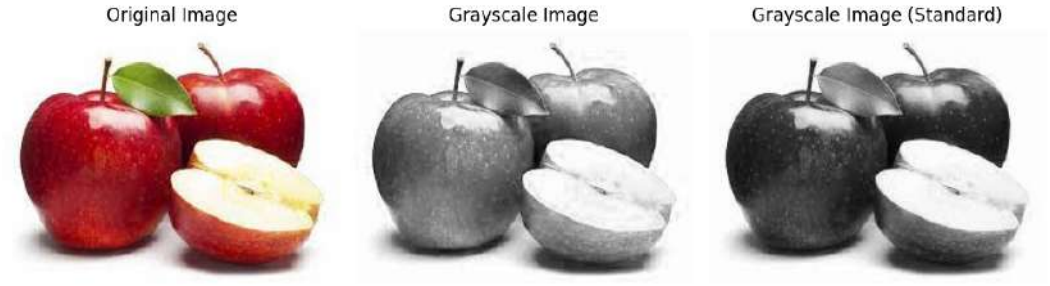
Finally gamma is set to 1-alpha-beta.

Depending on the values of alpha,beta and gamma, we get a variety of results.

```
def color_to_grayscale(img_path):
    # Determine the values of alpha, beta and gamma
    alpha = np.random.uniform(0, 1)
    beta = np.random.uniform(0, 1 - alpha)
    gamma = 1 - alpha - beta

    # Convert image to array
    pixels = img_to_array(img_path)

    # Change data type to avoid overflow
    pixels = pixels.astype(np.uint16)

    # Declare 2-D array to store grayscale values
    gray_scale_pixels = np.empty([pixels.shape[0], pixels.shape[1]], dtype=np.uint8)

    # Convert color pixels to grayscale pixels
    for i in range(pixels.shape[0]):
        for j in range(pixels.shape[1]):
            gray_scale_pixels[i][j] = int(alpha*pixels[i][j][0] + beta*pixels[i][j][1
```

```
    # Clip to ensure they are between 0 to 255
    gray_scale_pixels = np.clip(gray_scale_pixels, 0,255)

    return gray_scale_pixels
```





In most cases,

- `alpha = 0.299`
- `beta = 0.587`
- `gamma = 0.114`

These weights are derived from the luminescence formula , which is commonly used for grayscale conversion. This method approximates how the human eye perceives different colors that is perception of green is the highest followed by red and lastly blue. Thus these weights tend to give more real and impressive grayscale conversions than randomly assigned weight.

Convert a grayscale image to color using a pseudo color mapping

The first step is to create a color map that is each value from 0 to 255 maps to a RGB value. The color map can be created in any fashion.

Our color map is as follows:

0-86: R = 255, G = 0 to 255 equally divided in 86 parts, B = 0

86-171: R = 255 to 0 equally divided in 85 parts, G = 255, B = 0 to 255 divided in 85 parts

171-255: R = 0, G = 255 to 0 equally divided in 85 parts, B = 255

```
def create_colormap():
    colormap = np.zeros((256, 3), dtype=np.uint8)

    # Red to Green (0-85)
    colormap[0:86, 0] = 255            # Red channel
    colormap[0:86, 1] = np.linspace(0, 255, 86, dtype=np.uint8)  # Green channel
    colormap[0:86, 2] = 0              # Blue channel

    # Green to Blue (86-170)
    colormap[86:171, 0] = np.linspace(255, 0, 85, dtype=np.uint8)  # Red channel
    colormap[86:171, 1] = 255            # Green channel
    colormap[86:171, 2] = np.linspace(0, 255, 85, dtype=np.uint8)  # Blue channel

    # Blue to Red (171-255)
    colormap[171:, 0] = 0              # Red channel
    colormap[171:, 1] = np.linspace(255, 0, 85, dtype=np.uint8)  # Green channel
    colormap[171:, 2] = 255            # Blue channel

    return colormap
```

For changing grayscale to color, iterate through all pixels and pick the RGB values from the color map with index as the grayscale pixel intensity. Hence a single grayscale value is replaced with a 3-tuple forming a color image.

```
def grayscale_to_color(img_path):
    # Convert image to array
    pixels = img_to_array(img_path)

    # Change datatype to avoid overflow
    pixels = pixels.astype(np.uint16)

    gray_image = pixels[:, :, 0]
    colormap = create_colormap()
    color_pixels = colormap[gray_image]
    return color_pixels
```

Different images would need different color maps for practical coloring.

Green Screen

1. Convert the foreground and background both into arrays.
2. If the shapes do not match, resize background image to match foreground image.
3. We now need to identify the green pixels and replace them with pixels from background. For doing so, iterate through pixels using 2 for loops. Any pixel which has higher green intensity than the threshold, identify as green and replace with background. The threshold will vary from image to image and needs to be set accordingly.

```
def manual_resize(img,new_width,new_height):
    orig_height, orig_width = img.shape[:2]
    resized_img = np.zeros((new_height, new_width, 3), dtype=np.uint8)

    # Calculate the ratio of the old dimensions to the new dimensions
    row_ratio = orig_height / new_height
    col_ratio = orig_width / new_width

    # Iterate over the pixels of the resized image
    for i in range(new_height):
        for j in range(new_width):
            orig_i = int(i * row_ratio)
            orig_j = int(j * col_ratio)

            resized_img[i,j] = img[orig_i,orig_j]

    return resized_img

def replace_background(img_path,bg_path):
    img = img_to_array(img_path)
    bg = img_to_array(bg_path)

    if img.shape[:2] != bg.shape[:2]:
        bg = manual_resize(bg, img.shape[1], img.shape[0])

    for i in range(img.shape[0]):
```

```
        for j in range(img.shape[1]):
            green = img[i][j][1]
            if green>190:
                img[i][j] = bg[i][j]

    return img
```

Observation:

For most cases, a slight green line can be seen bordering the foreground image. This can be avoided by considering higher resolution images so that these green pixels very close to the foreground image can also be detected and replaced.

Reading A Video File

1. Use **cv2.VideoCapture** to open the video file.
2. Run an infinite loop to capture frames in the video till the video is completed. Use **video.read()** to read individual frames and capture as images. Store all the frames in a list.
3. Release the video object using **video.release().**

The file read is “earth_rotating.avi”.

```
def video_to_frames(video_path):
    # Open the video file
    video = cv2.VideoCapture(video_path)
    frames = []

    while True:
        # Read a frame from the video
        ret, frame = video.read()
        if not ret:
            break
        frames.append(frame)

    # Release the video capture object
    video.release()
    return frames
```

Writing frames to Video

1. Extract the dimensions of one frame.
2. Set the codec i.e encoder and decoder for writing video into a file according the format required.
3. Decide the frames per second frequency.
4. Use cv2.VideoWriter to make the video object.
5. Iterate through the list of frames and write one frame at a time to the video object using “video.write()”.
6. Finally release the video object.

The output video is stored in “earth_rotating_fast.avi”. I have increased the fps to double hence the recreated video is half the duration of the original video. We can also reverse the frames list and write to the video to reverse the original video. This can be used to create reverse blooming/ reverse aging videos.

```
def frames_to_video(frames, output_path, fps):
    # Dimensions of the frame
    height, width, layers = frames[0].shape

    # Define the codec and create a VideoWriter object
    fourcc = cv2.VideoWriter_fourcc(*'XVID')
    video = cv2.VideoWriter(output_path, fourcc, fps, (width, height))

    # Write each frame to the video file
    for frame in frames:
        video.write(frame)

    cv2.destroyAllWindows()
    video.release()
```

Create a 1 second transition video from one image to another

Use the same process as used to create videos from frames. Here instead of frames we have 2 images only and our frames per sec is 2 since we need to create a total of 1 second transition.

The transition video is stored in “transition.mov”.

```
def create_transition(img1_path,img2_path, output_path):
    img1=cv2.imread(img1_path)
    img2=cv2.imread(img2_path)

    height,width,layers = img1.shape
    img2 = cv2.resize(img2,(width,height))

    fourcc = cv2.VideoWriter_fourcc(*'mp4v')  # You can change this for different for
    video = cv2.VideoWriter(output_path, fourcc, 2, (width, height))

    video.write(img1)
    video.write(img2)
```

```
    cv2.destroyAllWindows()
    video.release()
```