A MultiModal Colorization App

Computer Vision Project

Teacher: Luigi Cinque, Marini

Student: Riccardo Scuto 2125102

Starting Paper

The paper “Control Color: Multimodal Diffusion-based Interactive Image Colorization “[1] introduces CtrlColor, a novel multi-modal image colorization framework based on the pre-trained Stable Diffusion model, designed to address several limitations present in existing colorization methods.

These limitations include lack of user interaction, inflexibility in local colorization, unnatural color rendering, insufficient color variation and color overflow.

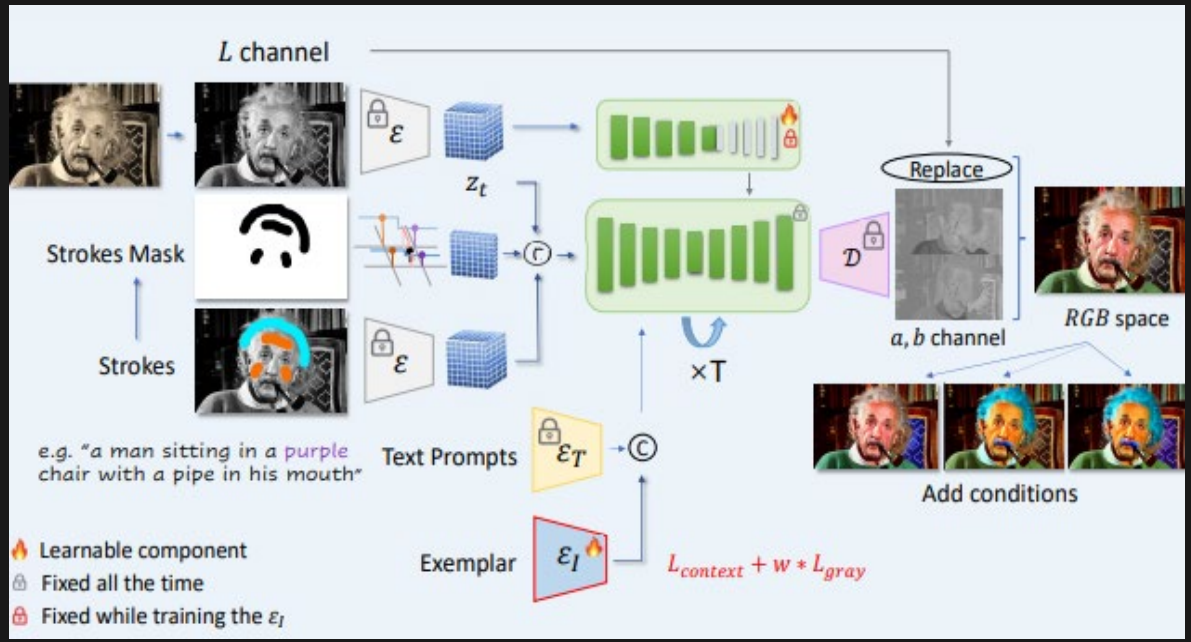
The CtrlColor framework is distinctive for its ability to unify various colorization tasks—unconditional, prompt-based, stroke-based, and exemplar-based colorization—within a single framework, leveraging the rich priors encapsulated in SD's latent diffusion model.

Key Features

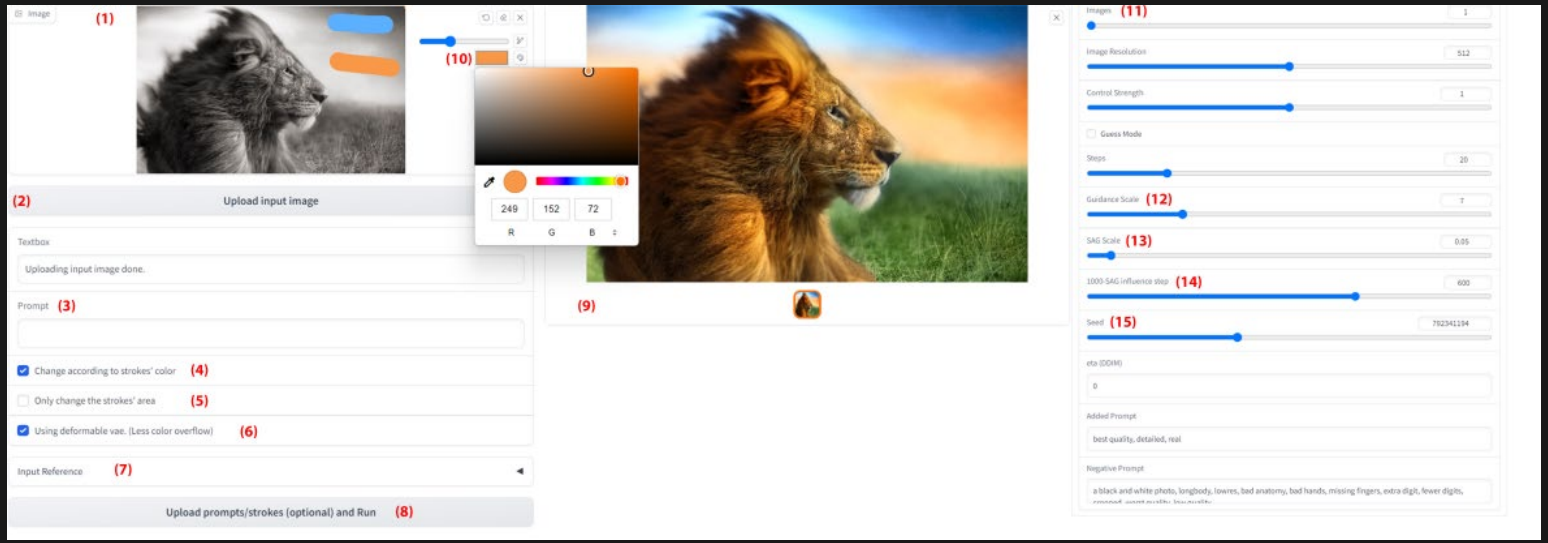
- **Highly Controllable Multi-modal Colorization:** Supports various colorization modes (prompt-based, stroke-based, exemplar-based, and their combinations) for both local and global color adjustments.
- **Addressing Color Overflow and Incorrect Color Issues:** Incorporates training-free self-attention guidance and a learned content-guided deformable autoencoder to tackle color overflow and inaccuracies.
- **User Strokes for Local Color Control:** Introduces a novel method for precise local color manipulation through user strokes, encoded directly into the diffusion process.
- **Leverage of Pre-trained SD Model's Advantages:** Utilizes the SD model's rich color variation and quality to produce superior colorization results compared to existing methods.

CtrlColor Modes

The framework's architecture integrates user inputs (e.g., strokes, exemplars) and text prompts into the colorization process, using self-attention and deformable convolution layers to mitigate color overflow and ensure color accuracy. This design enables precise control over the colorization output, allowing for highly customizable and natural-looking images.



The main structure of CtrlColor Model achieves multi-modal controllable colorization by blending controls from diverse components.



The interactive interface of our CtrlColor

CtrlColor represents a significant advancement in image colorization technology, providing a versatile and effective tool for achieving highly controllable and aesthetically pleasing colorizations. Its innovative use of the SD model, combined with novel color control mechanisms, addresses longstanding challenges in the field, opening new possibilities for creative and practical applications of image colorization.

Colorization App

The goal of the project was to create a program capable of coloring black and white images and videos and saving the converted versions. The program consists of three coloring modes:

- Automatic colorization
- Dot Colorization
- Palette Colorization

Automatic colorization

Automatic colorization uses deep neural networks and machine learning to predict and apply realistic colors to a grayscale image.

Is based on a pre-trained model known as Colorful Image Colorization published by Zhang[2] et al. in 2016. The model is trained on millions of images to learn correlations between an image's luminance (brightness or grayscale information) and its corresponding color.

```python
def load_model(self):
        prototxt_path = 'models/colorization_deploy_v2.prototxt'
        model_path = 'models/colorization_release_v2.caffemodel'
        kernel_path = 'models/pts_in_hull.npy'
        net = cv2.dnn.readNetFromCaffe(prototxt_path, model_path)
        points = np.load(kernel_path)
        points = points.transpose().reshape(2, 313, 1, 1)
        net.getLayer(net.getLayerId("class8_ab")).blobs = [points.astype(np.float32)]
        net.getLayer(net.getLayerId("conv8_313_rh")).blobs = [np.full([1, 313], 2.606, dtype="float32")]
        return net
```

Automatic colorization

- **Model Loading:** The pre-trained model with weights (**colorization_release_v2.caffemodel**) and network configuration (**colorization_deploy_v2.prototxt**) is loaded. Additionally, a set of anchor points (**pts_in_hull.npy**) representing ab cluster centers in the Lab color space is loaded. These points assist the network in normalizing the color output.

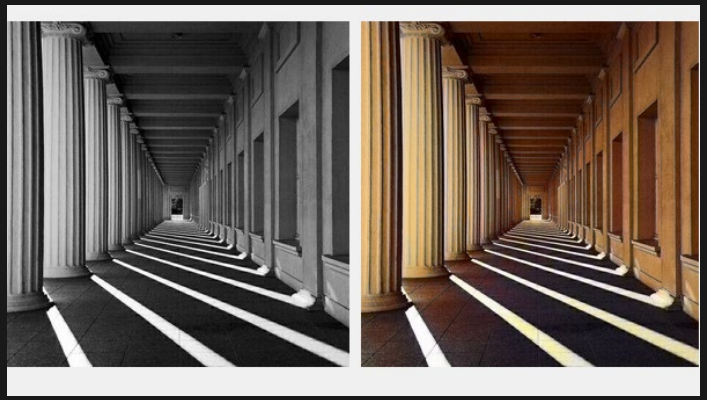
- **Image Pre-processing:** The black and white image is pre-processed to fit the network's input requirements. This involves converting it to grayscale, normalizing, and scaling it to an image size of 224x224. This step also includes subtracting 50 from the L channel to center the data around zero, which is common practice in neural network training.

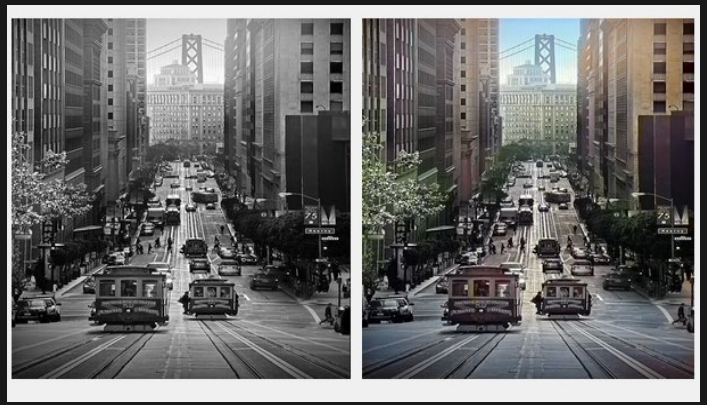
- **Color Prediction:** The pre-processed image is input into the network, which predicts the a* and b* channels in the image, representing chromatic information in the Lab color space.

- **Post-processing:** The predicted a* and b* channel outputs are upscaled to the original image size and combined with the L channel (luminance) to form a complete image in Lab format. The image is finally converted into the RGB color space, which is suitable for display.

- **Edge Enhancement:** After obtaining the colorized image, an edge mask is applied to preserve the original image details. This is done using the Canny algorithm for edge detection and replacing the corresponding pixel values at the edges with those from the original grayscale image. This step maintains sharp and well-defined edges, which may be lost during the colorization process.

Some examples





Some examples

Dot colorization

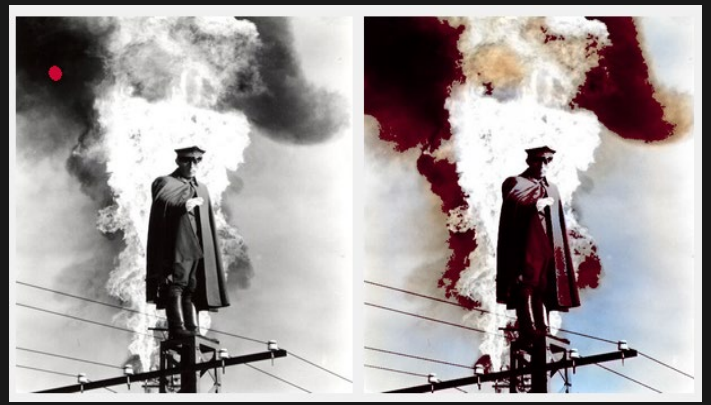
The Dot-based colorization process allows users to manually add color hints to a grayscale image, which then influences the colorization result.

```python
def process_stroke_based_colorization(self):
        if not self.strokes or self.selected_file_path is None:
            return
        original_image = cv2.imread(self.selected_file_path)
        gray_image = cv2.cvtColor(original_image, cv2.COLOR_BGR2GRAY)
        segments = self.segment_image(gray_image)
        net = self.load_model()
        colorized_auto = self.process_image(net, original_image)
        colorized_auto = np.array(Image.fromarray(cv2.cvtColor(colorized_auto, cv2.COLOR_BGR2RGB)))
        for stroke in self.strokes:
            x, y, stroke_color = stroke
            stroke_palette = self.get_stroke_palette(stroke_color)
            segment_id = segments[y, x]
            self.apply_interpolated_colors(colorized_auto, segment_id, stroke_palette, segments)
        self.colorized_image_pil = Image.fromarray(cv2.cvtColor(colorized_auto, cv2.COLOR_BGR2RGB))
        self.colorized_image_pil = Image.fromarray(cv2.cvtColor(np.array(self.colorized_image_pil), cv2.COLOR_BGR2RGB))
        self.update_frames(None, self.colorized_image_pil)
```

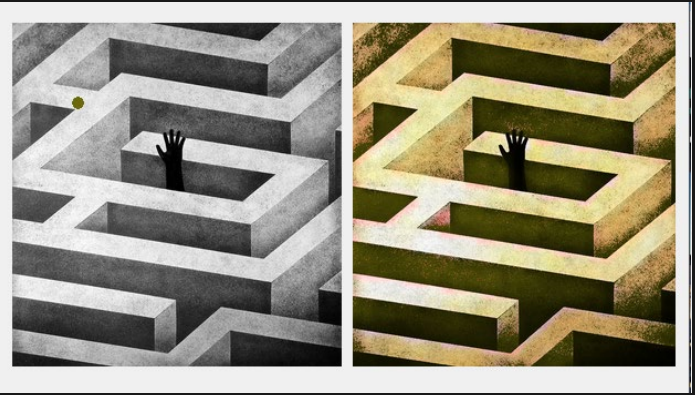
Dot colorization

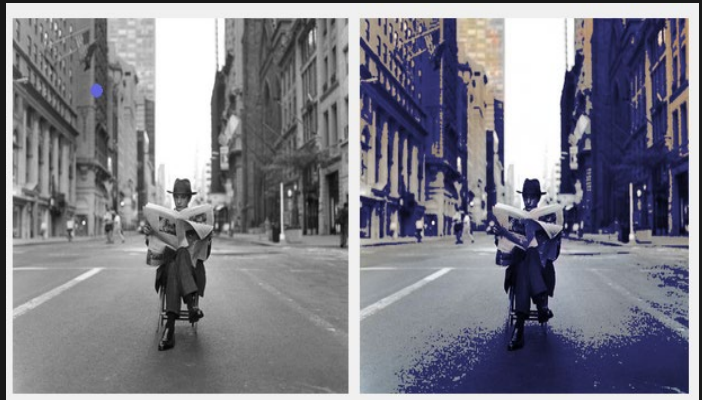
- **Dot Data:** The application allows users to draw a Dot on the grayscale image. This is associated with a specific color chosen by the user.
- **Identifying Unique Regions:** To apply user input accurately, the image is segmented into distinct regions. This process helps in mapping user strokes to specific parts of the image, ensuring that the colors are applied precisely where the user intended..
- **Dot Application:** For each stroke made by the user, the application identifies which segment of the image it belongs to. It then adjusts the colors of that segment according to the color specified by the user's stroke. This is done by mapping the stroke color to the Lab color space. Interpolating the stroke color across the identified segment, modifying the baseline colorization with the user's chosen colors.

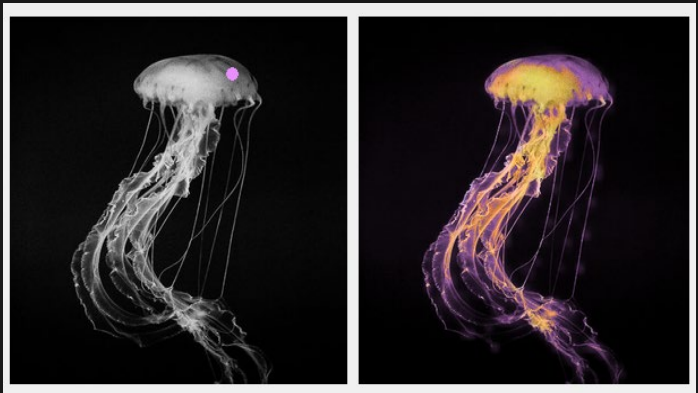
Some examples



Some examples







Palette colorization

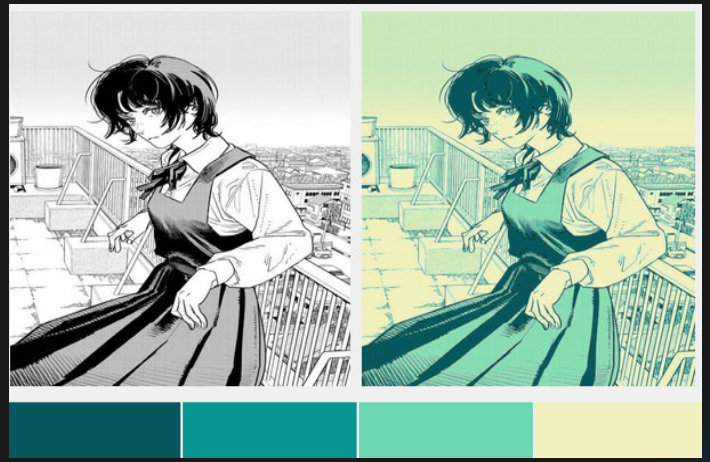
Palette colorization involves coloring a grayscale image based on API fetched palettes. This method automates the colorization process by mapping the grayscale tones of the image to specific colors from the palette, providing a uniform and consistent color scheme throughout the image.

```python
def apply_palette_colorization(self, image_path):
        selected_palette_name = self.palette_var.get()
        selected_palette = self.palettes[selected_palette_name]
        bw_image_cv = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
        colorized_image_cv = np.zeros((bw_image_cv.shape[0], bw_image_cv.shape[1], 3), dtype=np.uint8)
        edges = self.detect_edges(bw_image_cv)
        gray_scale_values = np.linspace(0, 255, num=len(selected_palette))
        interpolated_palette = list(zip(gray_scale_values, selected_palette))
        for i in range(bw_image_cv.shape[0]):
            for j in range(bw_image_cv.shape[1]):
                grayscale_value = bw_image_cv[i, j]
                if not edges[i, j]:
                    colorized_image_cv[i, j] = self.interpolate_color(grayscale_value, interpolated_palette)
                else:
                    colorized_image_cv[i, j] = [grayscale_value, grayscale_value, grayscale_value]
        colorized_image_pil = Image.fromarray(colorized_image_cv)
        self.colorized_image_pil = colorized_image_pil
        self.update_frames(None, self.colorized_image_pil)
```

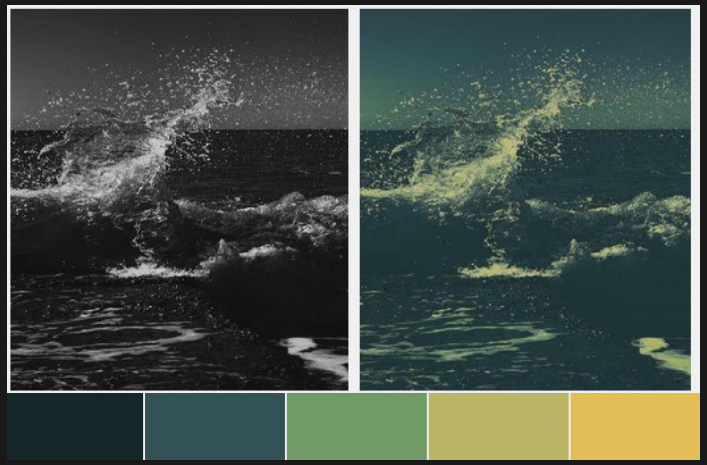
Palette colorization

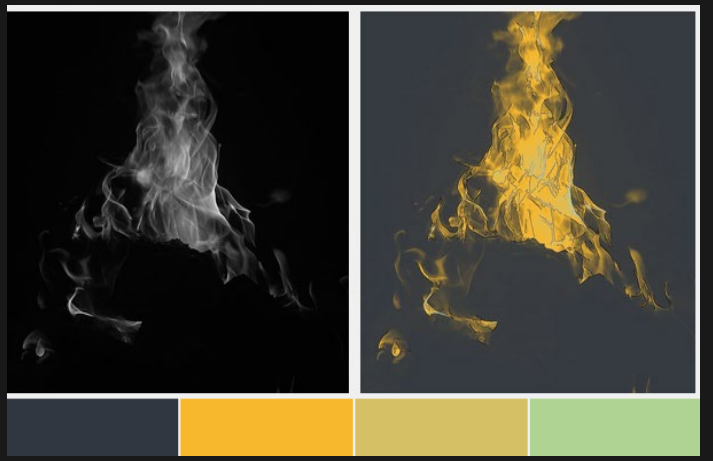
- **Palette Selection and API Fetching:** Utilizing the fetch_palettes method, the app retrieves color palettes from Colormind API[3], Users can select a desired palette, from 1 to 5 colors, dynamically loaded into the application from the fetched results.
- **Pre-processing for Colorization:** A grayscale image selected by the user is read and processed. The edges of this image may be detected using the Canny edge detection algorithm to preserve details during colorization.
- **Palette Application:** The selected color palette is mapped onto the grayscale image based on the intensity of each pixel. The apply_palette_colorization method interpolates colors from the palette across the grayscale range, assigning colors to pixels based on their luminance.
- **Post-processing and Display:** The now colorized image is post-processed, which may involve adjusting the color balance, saturation, and brightness to ensure a natural look.The colorized image is then displayed alongside the original for comparison.

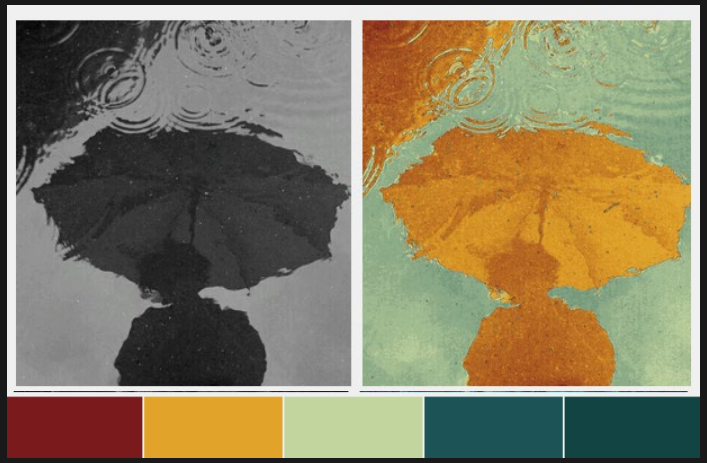
Some examples



Some examples







Video colorization

Video colorization follows a process that extends the automatic image colorization approach to handle sequences of frames.

```
def colorize_video(self, video_path):
        net = self.load_model()
        cap = cv2.VideoCapture(video_path)
        self.colorized_frames = []
        while True:
            ret, frame = cap.read()
            if not ret:
                break
            colorized = self.process_image(net, frame)
            self.colorized_frames.append(colorized)
        cap.release()
        self.show_video_preview()
```

Video colorization

- **Automatic Colorization of Frames:** Each extracted frame undergoes the same colorization process as described for automatic image colorization. This involves using a pre-trained deep learning model to predict the chromatic channels (a^* and b^*) for the frame based on its luminance channel (L). The deep learning model, trained on a dataset of color images, applies learned colorization patterns to the grayscale frames, effectively predicting realistic colors for each frame.

- **Post-processing:** Like image colorization, post-processing steps might be applied to each colorized frame to enhance visual quality. This can include adjustments to color balance, contrast, and saturation to ensure consistency across the video sequence. Edge enhancement techniques might also be used to preserve details and sharpness in the colorized frames.

- **Reconstruction and Output:** Once all frames have been colorized and post-processed, they are recompiled back into a video format. This recompilation involves setting the correct frame rate and resolution to match the original video, ensuring that the output video retains the smooth playback of the original.

Some examples

Videos

Future remarks

Improving Dot Colorization System

Utilizing Improved Segmentation Algorithms: Implement advanced segmentation algorithms like Mask R-CNN or U-Net for accurately identifying regions to color
ser Interface for Precise Selection: Enhance the user interface with more sophisticated selection tools, such as variable-size brushes, erasers, and zoom options, to allow users to apply colors with greater accuracy.

Optimization and Crash Reduction

Error Handling and Logging: Incorporate robust error handling and detailed logging systems to quickly identify and resolve crash causes.
Code Optimization: Review and optimize the code for better performance and memory efficiency, for instance, by minimizing the use of resource-intensive operations and optimizing loops.

Adding New Coloring Modes

Coloring via Text Prompt: Implement an interface where users can input text prompts describing the desired colors for specific parts of the image, e.g., "the sky is blue, and the grass is green."
Use natural language models along with neural networks to interpret prompts and apply corresponding colors to the relevant parts of the image.

References

[1] https://zhexinliang.github.io/Control_Color/

[2] https://richzhang.github.io/colorization/

[3] http://colormind.io/api-access/