Computer Vision
Colorizing grayscale images

Fall 2024

Parsa Darban 810100141

This file contains the report and results of the simulations conducted.

Abstract

Colorizing grayscale images is an important field in image processing and computer vision, achieved through deep learning techniques and neural networks. Many historical photographs, originally captured in black and white, can be restored to their realistic colors using artificial intelligence and deep neural networks. This process not only helps in better understanding the past but also enhances visual appeal and provides deeper insights into the details of that era.

Another application is in the medical field, particularly in radiology. Medical images such as X-rays and MRIs are traditionally displayed in black and white. Colorizing these images can assist doctors in identifying abnormalities and medical conditions more easily. By leveraging deep learning techniques, these images can be reconstructed in color, allowing for more accurate disease diagnosis and analysis.

Additionally, colorizing grayscale images is useful in the film industry and video game development. Many old movies, recorded with early technology, are being colorized using artificial intelligence and machine learning to make them more appealing to modern audiences. In video games, this technique can enhance visual quality by adding color to black-and-white scenes or improving details in dark environments.

Overall, colorizing grayscale images not only improves visual experience but also has practical benefits in various scientific and industrial fields. In this project, we will explore this topic using convolutional neural networks (CNNs).

There are various CNN-based architectures that can be used to generate a new image after extracting features. Architectures such as Autoencoder (AE), Variational Autoencoder (VAE), Generative Adversarial Network (GAN), and others can be utilized for this purpose. In this project we use simple autoencoder.

Dataset & Preprocessing

Our dataset consists of nature images (mountains and fields), building images, and images of male and female faces. This dataset was created by us and downloaded from Kaggle and images.cv. It includes images of various sizes and formats.

Figure 1 (dataset images)

We know that the input to the neural network needs to have the same resolution. Therefore, we first resize the images to 256x256. It's important to note that due to the network architecture and the use of the ReLU activation function, we must normalize the images by dividing their values by 255. ReLU is an activation function that changes from zero to positive infinity if the input is positive. By normalizing, we limit this range from zero to one.

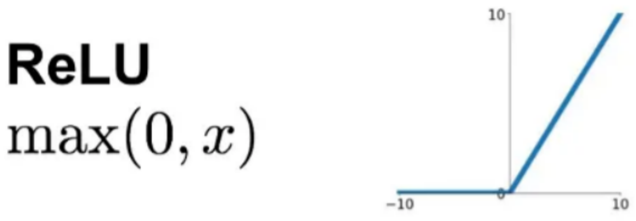


Figure 2 (ReLU activation function)

Our dataset has many samples, so we need to select a suitable number based on the memory capacity. This is achieved by choosing the batch size. The batch size determines how many images are fed to the model at each step of training or evaluation. If the batch size is too small, the model will learn from fewer data points each time the weights are updated. Therefore, selecting an appropriate batch size directly affects the performance of the model and may require experimentation.

As we know, this task is supervised, meaning we have both the input grayscale image and the corresponding output label (colorized image). There are different approaches to create the grayscale images. For example, we could convert color (RGB) images to grayscale and consider the labels to be the original RGB images. In this case, the output would have three channels. In RGB, we would need to predict three channels for each image, which increases complexity.

To address this, we have employed a technique to reduce the number of output channels. Here, we have used the Lab color space. The Lab model is one of the color models designed to represent colors independently of lightness (L). This model is widely used for image processing tasks such as image colorization, object detection, and medical image processing. The Lab model consists of three main channels:

- L (Lightness): Represents the lightness or intensity of the image. The L channel ranges from 0 (black) to 100 (white).

- a: Represents the color range between green and red. A negative a value indicates green, and a positive value indicates red. This channel ranges from -128 to 127.
- b: Represents the color range between blue and yellow. A negative b value indicates blue, and a positive value indicates yellow. This channel also ranges from -128 to 127.

The Lab model is suitable for applications like colorization and image editing because it separates the lightness channel (L) from the color channels (a and b), allowing changes in lightness without affecting the colors. This model is also more resistant to lighting variations and operates independently of light. Lab color space represents colors in a way that is similar to human vision, with greater sensitivity to lightness (L) differences than to color differences. Additionally, the Lab color space is broader than RGB and can display colors that cannot be accurately represented in RGB. Therefore, this approach allows the model to focus on the colors without worrying about lightness when predicting the colors.

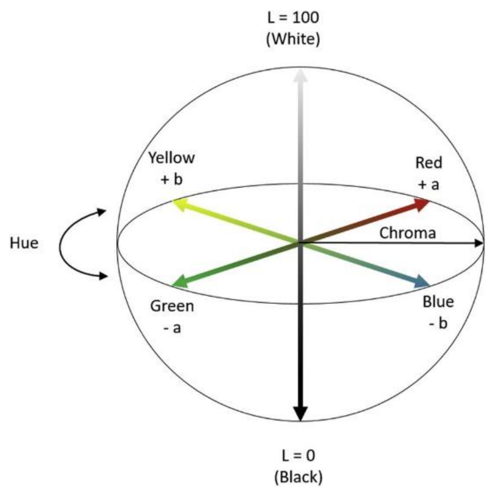


Figure 3 (LAB space)

In the LAB format, the values can be negative. On the other hand, we know that the tanh activation function can handle negative values, and its range is between -1 and 1. Therefore, after converting the dataset images and dividing them into their components, we need to normalize the a and b channels in the same way as the RGB image values in the dataset. Furthermore, we use the tanh activation function in the final layer of the network architecture to handle these values, as it can output values in the required range of -1 to 1, making it suitable for our color prediction task.

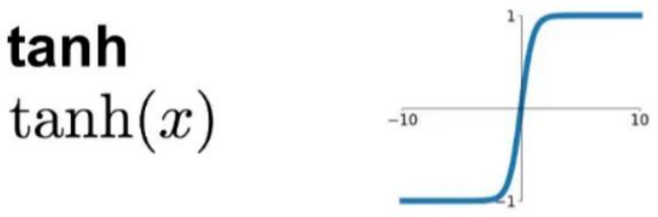


Figure 4 (tanh activation function)

Autoencoder

An Autoencoder is a type of neural network designed to learn a compressed and efficient representation of data (such as images or text). The network consists of two main parts: the Encoder, which converts the input into a compressed representation (code), and the Decoder, which reconstructs the original input from this compressed representation.

The primary goal of an Autoencoder is to enable the network to represent data in a more compact form while preserving its key features. This model is commonly used for dimensionality reduction, compression, denoising images, or learning complex features.

So, our model uses an Encoder-Decoder architecture that processes grayscale images (the L channel from the LAB color space) and predicts the A and B color channels.

- The encoder consists of multiple Conv2D layers with ReLU activation and BatchNormalization(). It progressively reduces the spatial dimensions of the input image using MaxPooling2D() to extract high-level features. The final encoder layer produces a compressed feature map of the image. This helps the model learn important features (such as edges, shapes, and textures) necessary for predicting realistic colors.
- The decoder includes UpSampling2D(), which restores the compressed features back to the original image size (256×256). As mentioned, the final layer has two output channels, predicting only the A and B color channels. The output also uses the tanh activation function, scaling values between -1 and 1 to match the LAB color space. The decoder reconstructs the missing color information that was not present in the grayscale input image.

Thus, the model learns from grayscale and color images to understand how grayscale intensities correspond to real colors. The input is a grayscale image (L channel from LAB) of size (256×256×1). The encoder compresses it to extract important features, and the decoder reconstructs the colors, producing an output of size (256×256×2) (the A and B color channels). The L channel (input) is then combined with the predicted AB channels to obtain the final colorized image.

Since the model learns an average of observed colors, the predicted colors may sometimes appear dull or desaturated.

Batch Normalization is a deep learning technique used to stabilize training by normalizing activation values in a neural network. This method standardizes the inputs of each layer so that their mean is zero and variance is one, reducing internal covariate shift—the changes in data distribution during training. Batch normalization is applied over mini-batches, ensuring that the distribution of activation values remains stable throughout training. This technique improves gradient flow, allows for higher learning rates, and acts as a form of regularization by introducing slight noise into the data, reducing the risk of overfitting.

Models using BatchNormalization typically converge faster and generalize better. As a result, this technique is widely used in modern architectures such as ResNets, various VGG versions, and Generative Adversarial Networks (GANs).

MaxPooling is a layer in Convolutional Neural Networks (CNNs) used to reduce the dimensionality of feature maps and increase the model's resistance to spatial variations in an image. This layer works by dividing the input into smaller sections (e.g., 2x2 or 3x3) and selecting the maximum value from each section, thereby reducing the size of the feature map while retaining important information.

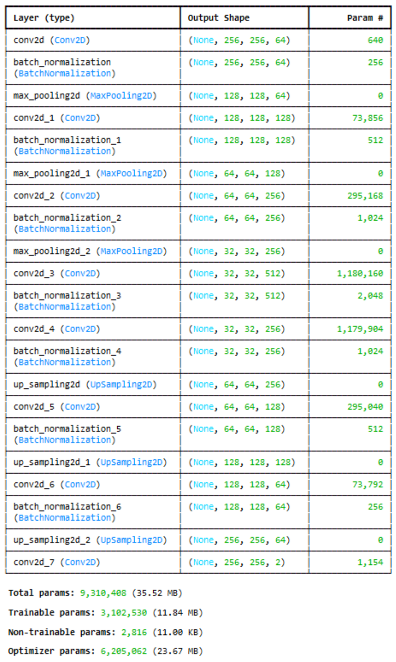
This process helps reduce computations, prevents overfitting, and enhances the model's efficiency in detecting key patterns without the need to store unnecessary information.

We use optimizer='adam'. Adam (short for Adaptive Moment Estimation) is an advanced optimization algorithm that combines the benefits of two other popular optimizers: AdaGrad and RMSProp. It computes adaptive learning rates for each parameter by considering both the first moment (mean) and second moment (uncentered variance) of the gradients. This allows Adam to adjust learning rates dynamically, improving convergence speed and reducing the need for manual tuning of the learning rate, making it highly effective in training deep learning models.

The loss function chosen here is Mean Squared Error (MSE), which is commonly used for regression tasks. MSE measures the average of the squared differences between the predicted values and the true values. It penalizes larger errors more heavily, making it useful when you want to minimize the overall prediction error. In the context of tasks like image reconstruction or colorization, MSE helps to ensure the model produces output that closely matches the target values.

Mean Absolute Error (MAE) is used here as an evaluation metric. MAE calculates the average of the absolute differences between the predicted and actual values, providing a simple and intuitive measure of model performance. Unlike MSE, which squares the error, MAE treats all errors equally and is less sensitive to outliers. It's commonly used when you want to understand the average magnitude of errors in predictions, especially when dealing with regression tasks.

The bottleneck in our architecture is well-designed and does not cause common issues like vanishing gradients or poor feature representation. By using 512 filters in the final convolutional layers, rich features are preserved, and the decoder is able to effectively reconstruct the image. Additionally, using BatchNormalization after each convolutional layer helps stabilize training and reduces issues such as gradient explosion or vanishing gradients. The UpSampling2D layers progressively reconstruct spatial information, and excessive compression in the bottleneck is avoided. This architecture effectively preserves and reconstructs information without the need for skip connections.



Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 256, 256, 64)	640
batch_normalization (BatchNormalization)	(None, 256, 256, 64)	256
max_pooling2d (MaxPooling2D)	(None, 128, 128, 64)	0
conv2d_1 (Conv2D)	(None, 128, 128, 128)	73,856
batch_normalization_1 (BatchNormalization)	(None, 128, 128, 128)	512
max_pooling2d_1 (MaxPooling2D)	(None, 64, 64, 128)	0
conv2d_2 (Conv2D)	(None, 64, 64, 256)	295,168
batch_normalization_2 (BatchNormalization)	(None, 64, 64, 256)	1,024
max_pooling2d_2 (MaxPooling2D)	(None, 32, 32, 256)	0
conv2d_3 (Conv2D)	(None, 32, 32, 512)	1,180,160
batch_normalization_3 (BatchNormalization)	(None, 32, 32, 512)	2,048
conv2d_4 (Conv2D)	(None, 32, 32, 256)	1,179,904
batch_normalization_4 (BatchNormalization)	(None, 32, 32, 256)	1,024
up_sampling2d (UpSampling2D)	(None, 64, 64, 256)	0
conv2d_5 (Conv2D)	(None, 64, 64, 128)	295,040
batch_normalization_5 (BatchNormalization)	(None, 64, 64, 128)	512
up_sampling2d_1 (UpSampling2D)	(None, 128, 128, 128)	0
conv2d_6 (Conv2D)	(None, 128, 128, 64)	73,792
batch_normalization_6 (BatchNormalization)	(None, 128, 128, 64)	256
up_sampling2d_2 (UpSampling2D)	(None, 256, 256, 64)	0
conv2d_7 (Conv2D)	(None, 256, 256, 2)	1,154

```
Total params: 9,310,408 (35.52 MB)
Trainable params: 3,102,530 (11.84 MB)
Non-trainable params: 2,816 (11.00 KB)
Optimizer params: 6,205,062 (23.67 MB)
```

Figure 5 (Model summary)

Train Model

The designed neural network is trained with 800 and 1000 epochs and a mini-batch size of 128. The error curve is shown below:

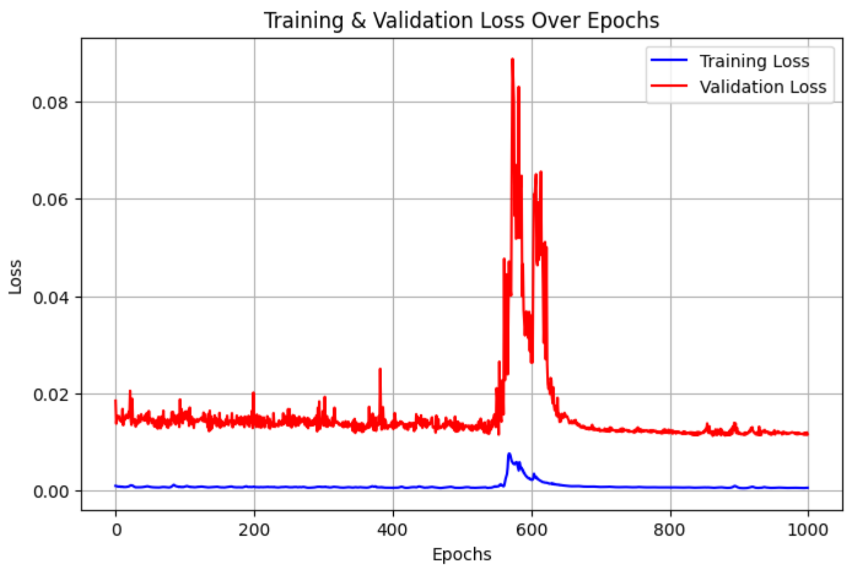


Figure 6 (Train and Validation Loss)

Test Model

We save the model and load it in the Test_Model.ipynb file to apply it to images. For the preprocessing step, since in a real scenario we do not have the color images and need to provide grayscale images to the model, we first repeat the grayscale image three times to convert it into an RGB format. Then, we convert these images to the Lab color space and pass the L channel to the model, allowing it to predict the colorized image.

Another approach is to read grayscale images as colored images and then convert them to the LAB color space. However, the results show that this method is not as effective in colorizing images compared to the previous approach.

The results, along with the preprocessing method and model type, are as follows:

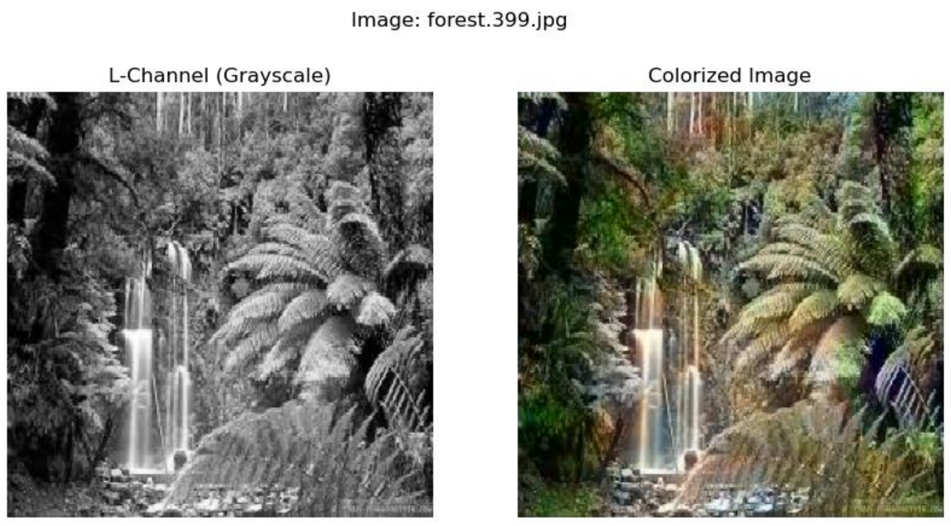


Figure 7 (800 epochs & first preprocessing)

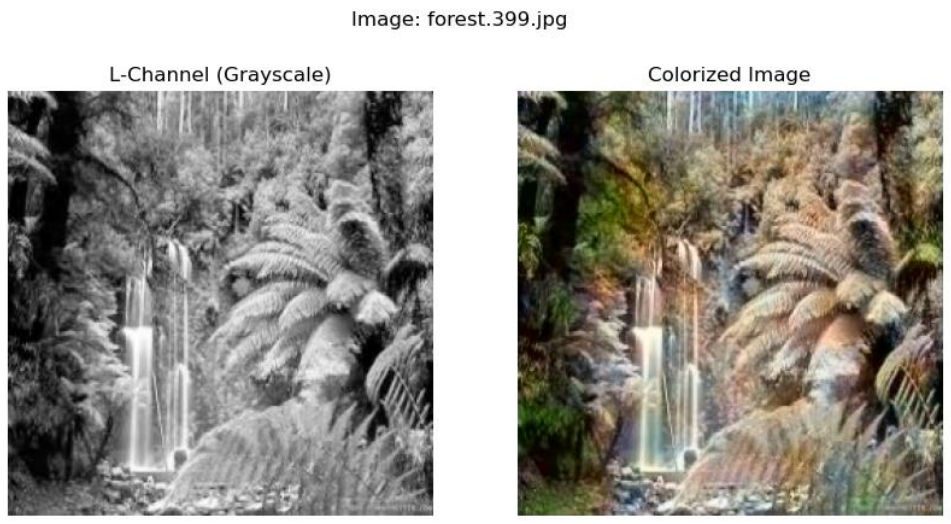


Figure 8 (800 epochs & second preprocessing)

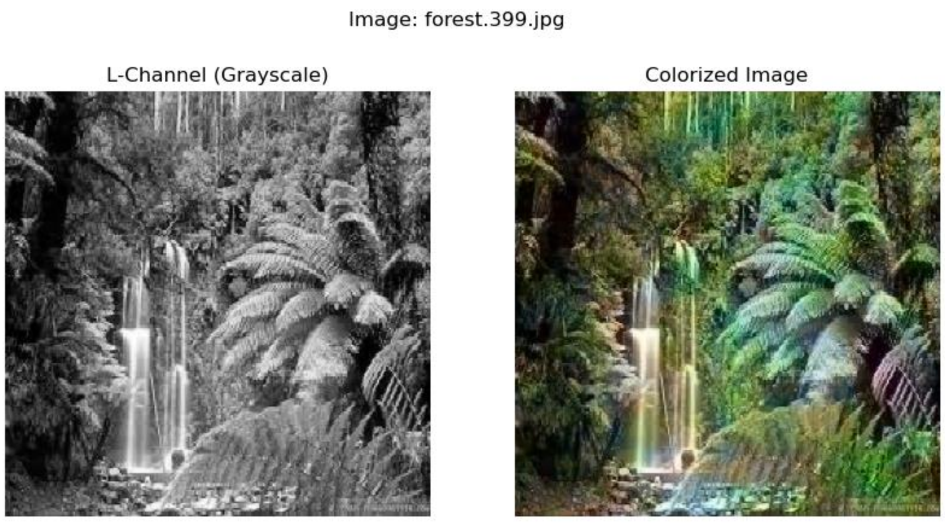


Figure 9 (1000 epochs & first preprocessing)

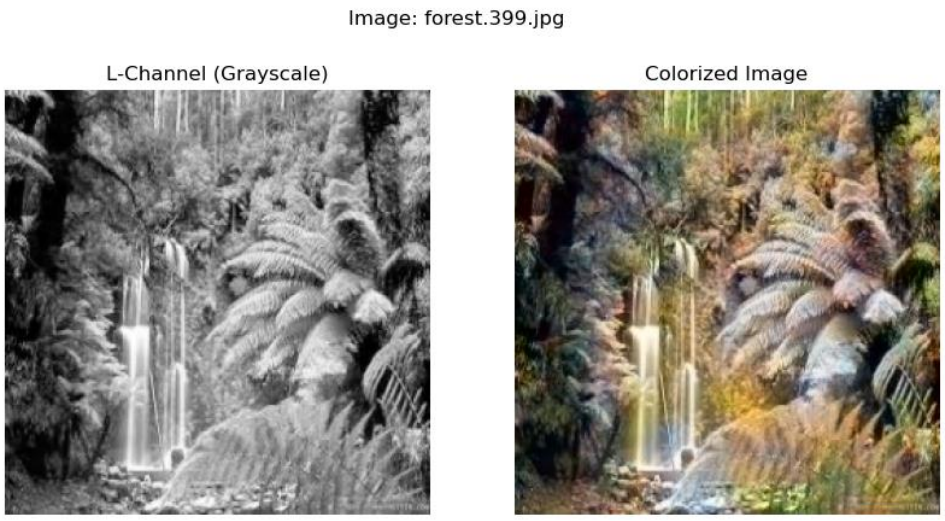


Figure 10 (1000 epochs & second preprocessing)

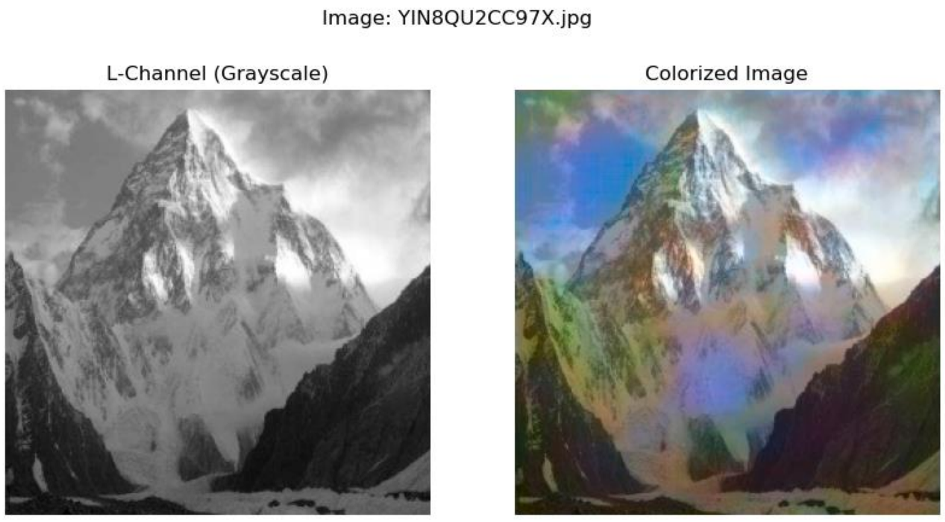


Figure 11 (800 epochs & first preprocessing)

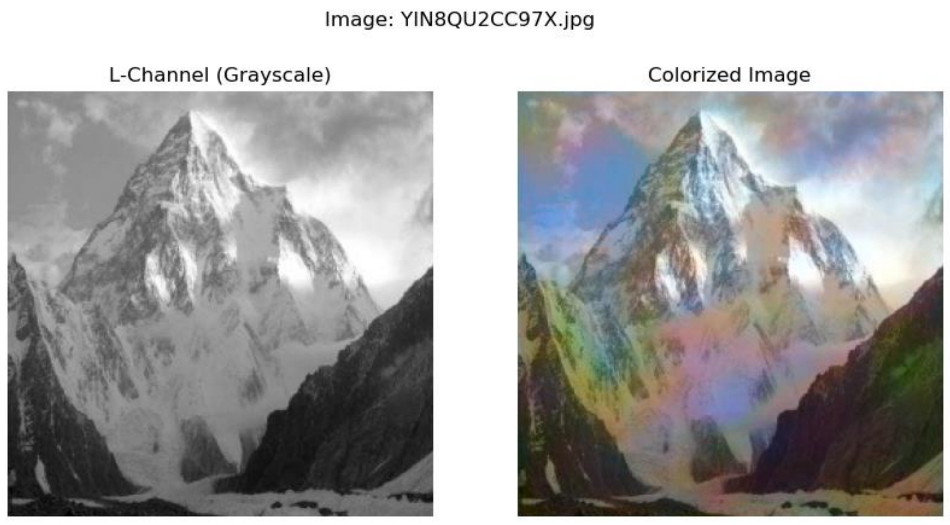


Figure 12 (800 epochs & second preprocessing)

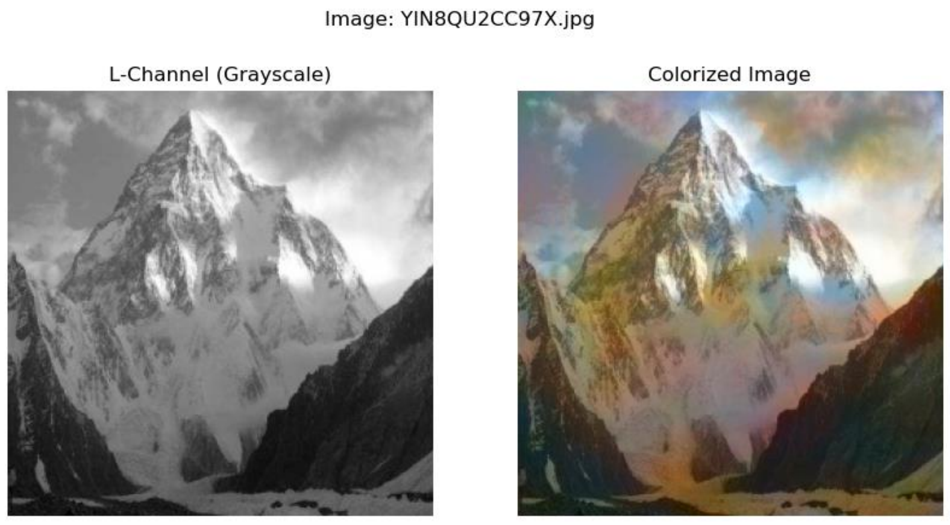


Figure 13 (1000 epochs & first preprocessing)

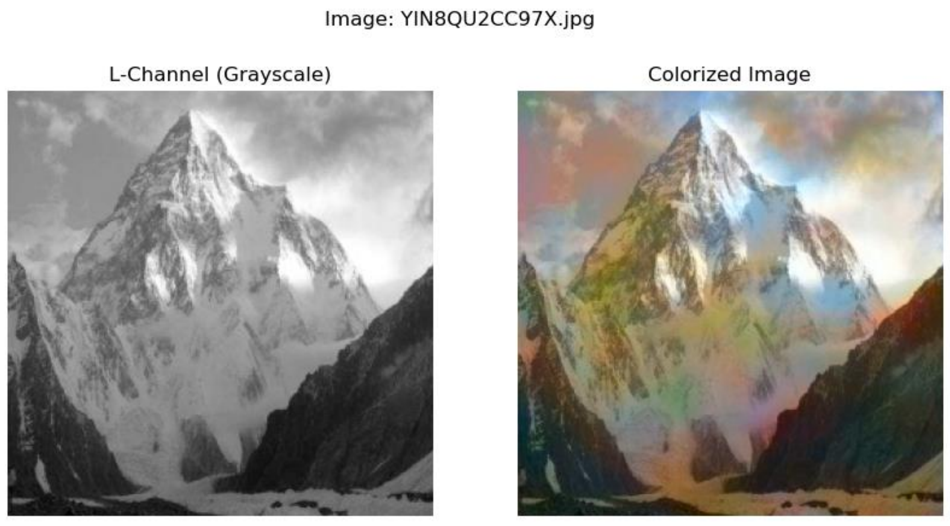


Figure 14 (1000 epochs & second preprocessing)

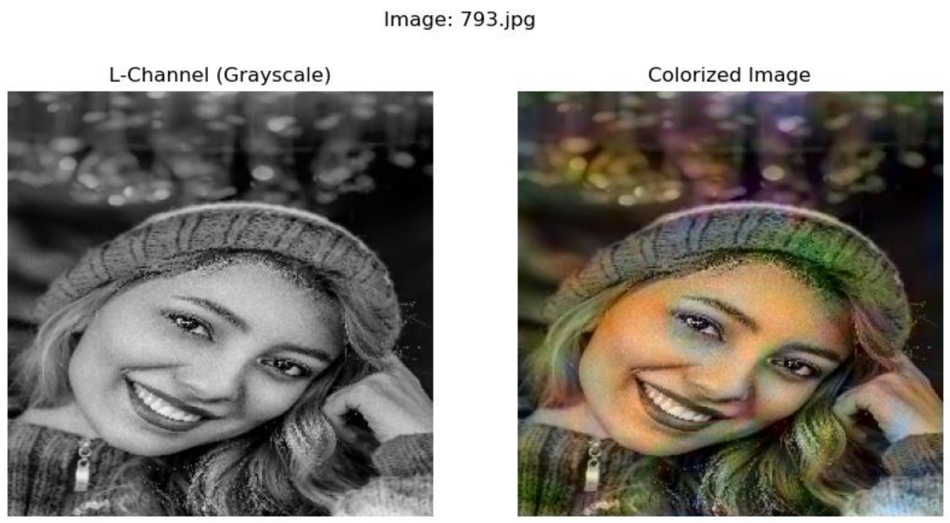


Figure 15 (800 epochs & first preprocessing)

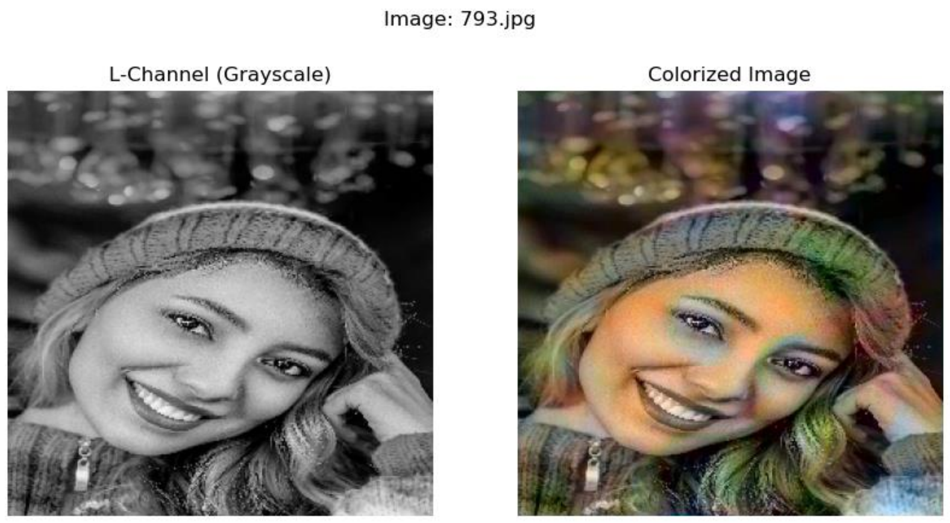


Figure 16 (800 epochs & second preprocessing)

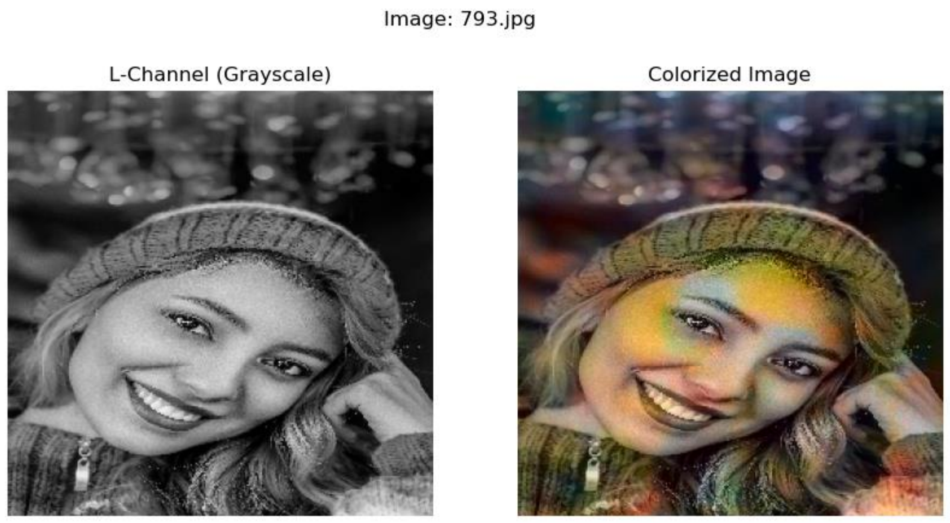


Figure 17 (1000 epochs & first preprocessing)

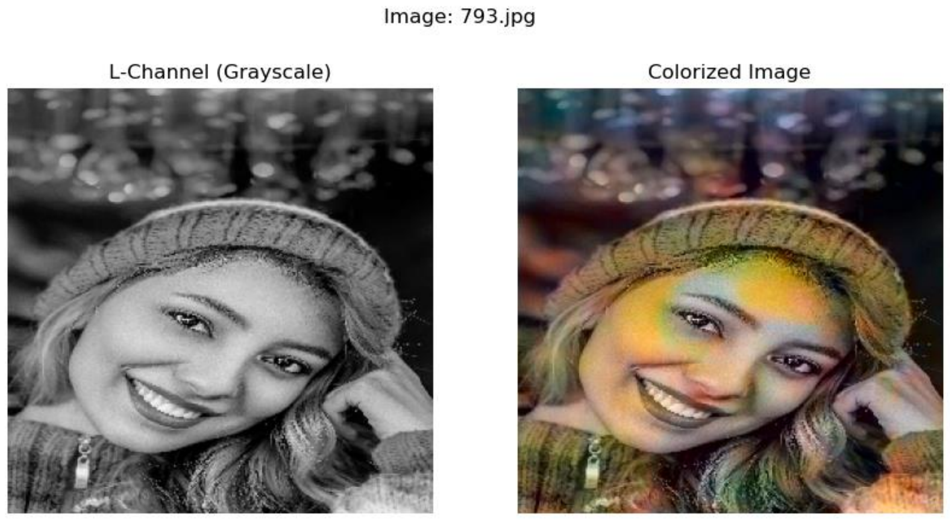


Figure 18 (1000 epochs & second preprocessing)

Video's part processes a grayscale video and applies a trained neural network model to colorize each frame. It first reads the video frame-by-frame using OpenCV, converts each frame to RGB, resizes it to 256×256, and transforms it into the Lab color space. The L channel (grayscale information) is extracted and fed into the trained model, which predicts the a and b color channels. The predicted channels are merged with the L channel to reconstruct the colored frame, which is then converted back to RGB, resized to the original video dimensions, and saved in a new video file. Additionally, an FPS (frames per second) overlay is added to each frame, displaying a fixed FPS of 30.20. The script ensures smooth frame processing by adjusting sleep time if necessary.

The output video, saved as "output_colorized_fps.mp4", is a colorized version of the original grayscale input, with each frame processed and displayed at 30.20 FPS. The actual processing speed depends on hardware capabilities, particularly GPU acceleration. If the neural network model processes frames faster than the target FPS, the script introduces a slight delay to maintain consistent playback speed. If the model is slow, FPS may drop below the target value, leading to frame skips or delays.

Real time's part processes a grayscale video frame by frame and colorizes it using a deep learning model. First, the video is loaded using OpenCV, and its properties, including width, height, and frame rate (FPS), are obtained. Then, the display frame size is reduced to speed up processing. To calculate the real-time frame rate, the time of each frame is recorded and compared with the previous frame.

In each loop, a new frame is read from the video and converted from BGR to RGB, as OpenCV reads images in the BGR format, while the neural network processes them in RGB format. The frame is then resized to 256×256 and converted to the Lab color space. The L channel, which contains luminance information, is extracted and fed into the model as input. The deep learning model predicts the a and b channels, and these values are combined with the L channel to reconstruct the colorized frame.

After being processed by the model, the colorized image is converted from the Lab color space to RGB, and its values are adjusted to the range of 0 to 255. The colorized frame is then resized back to the original video dimensions and converted back to BGR format for display in OpenCV. Finally, the colorized frame is shown, and if the 'q' key is pressed, the loop stops, and the video processing ends.

User Interface

This Python code creates a graphical user interface (GUI) using the tkinter library for colorizing grayscale images. The application allows users to upload an image, process it through a pre-trained model, view the colorized result, and save the colorized image. The main functionality involves converting grayscale images into colorized versions using a deep learning model that predicts the missing color information in the image. This process is achieved through converting the image into the LAB color space, extracting the luminance (L) channel, and predicting the chrominance (a and b) channels, which are then combined to generate the final colorized image.

The code begins by defining the colorize_image function, which takes the path of a grayscale image and a trained model as inputs. It preprocesses the image by resizing it and converting it into a format suitable for the model. The image is first transformed into the LAB color space, where only the luminance (L) channel is passed to the model. The model outputs the color information (a and b channels), which is then combined with the L channel to create a colorized image in the LAB color space. The resulting LAB image is converted back to RGB format to display as a colorized output.

The user interface consists of three main buttons: one to upload an image, one to display the colorized image, and one to save the colorized image. When the user clicks the "Upload Image" button, a file dialog is opened, allowing the user to select an image. The image is then processed by the colorize_image function, and the user is notified that the image has been processed successfully. However, the colorized image is not displayed immediately. Instead, the user must click the "Show Colorized Image" button to view the result.

Finally, the "Save Colorized Image" button allows the user to save the colorized image to their local file system. A save file dialog appears, enabling the user to choose the location and file format for saving the image. If the user selects a location, the image is saved in the chosen format. If any errors occur during the image processing or saving steps, the program displays an appropriate error message. This code provides an interactive way for users to utilize a deep learning model for image colorization with an intuitive GUI.

Conclusion

As we can see from the results of the images, the model is more sensitive to certain features like grass-like textures, hair tips, and so on at higher epochs. Another issue is the color of people's faces. The dataset related to people's faces contains various colors for clothing, and the backgrounds are not consistent. As a result, many of the backgrounds in these images are rendered in sky colors because the dataset images have preserved the sky feature quite well.

To improve the model, we can use pre-trained networks and create our own decoder for them. This approach, known as transfer learning, allows us to leverage the knowledge learned by a pre-trained network on a large dataset (such as ImageNet) and adapt it to our specific task. The idea is to freeze the layers of the pre-trained network (usually the encoder part) and then add our own decoder network that can generate the output in the desired format, in this case, colorizing the grayscale images. This helps the model learn the relevant features more effectively, especially when dealing with limited data, and can lead to better results.