Problem Approach :

After conducting extensive literature reviews, I determined that the CIE L*a*b color space should be used for training our model due to its ability to reduce the number of parameters that need to be estimated. This is further explained in detail in the DataSet selection and preprocessing section.

This problem can be approached as both a multi-modal classification problem and a regression problem. However, due to the simplicity and effectiveness of the regression method, I chose to approach it using regression. In this approach, my loss function is the Mean Square Error (MSE) between the estimated pixel colors in the a*b* space and their real values.

For a picture X, the MSE is given by:

$$C(\mathbf{X}, \boldsymbol{\theta}) = \frac{1}{2HW} \sum_{k \in \{a,b\}} \sum_{i=1}^{H} \sum_{j=1}^{W} (X_{k_{i,j}} - \tilde{X}_{k_{i,j}})^2,$$

where θ represents all model parameters, X_k and $\tilde{x}_k$ represents target and reconstructed images respectively.

After formulating the loss function, I must decide which architecture to try. First, I started using Conventional CNN architectures for this project. Still, those architectures were not learning anything but instead trying to minimize the loss function by predicting mean intensity, and that is the natural solution for the above equation.

After formulating the loss, I began researching possible architectures that fit our purpose. Our approach should be to minimize local loss rather than global loss. So the idea is to cluster similar colored, shaped, or shaded objects from the same image and apply the loss function on those clusters only, So I chose Autoencoders. The autoencoders map the image from high dimensional space to low dimensional subspace by creating new features after combining existing features. Thus, the problem of reconstruction of a*b channels from 256*256 features is reduced to reconstruction from n*n features, where n is the dimension of latent space.

To improve the accuracy further, I continued researching. I came across a paper discussing how fusing the encoded features with features generated from a pre-trained classification network enhances performance. For this problem, I used pre-trained Inception_v3 as my feature generator, which extracts global features from the image and adds the global context to the colorization process. Skip connections are added from encoders to decoders to improve the learning process further.

Architecture :

Model = Encoder + Decoder + Skip connections + Feature generator + Fusion layer

The goal of this architecture is to combine both local as well as global features from the gray-scale image and decode the colorful image accurately.

Encoder : The encoder reduces the dimension of the original image and preserves those features which are very much constrained to the input image itself i.e local features. These features are sufficient enough to reconstruct the input image back without much loss. Padding is used to preserve the layer's input size.

Feature generator : An additional pre-trained Inception_v3 network is used as feature generator which has the ability to extract the global level features which can be generalized across other images such as class of background, whether image is in forest or beneath the sea and many others. These features will give us the context of the image thus will improve the accuracy of our model when fused with the encoded features. It returns embedding of dimension 1*1*1000.

Decoder : The decoder then reconstructs the image using upsampling layers interleaved with convolutional layers. Skip connections are utilized to combine features from corresponding encoder layers, preserving both high-level semantic information and low-level spatial details. The final output is a two-channel image (ab channel) of 256*256 dimension, with the pixel values normalized using a Tanh() activation function.

Fusion: The fusion layer takes the feature vector from Feature generator, replicates it along 32*32 and concatenate it to encoder along the depth axis. So the final dimension of encoded feature is equivalent to 32*32*1257. This ensures that the semantic information conveyed by the feature vector is uniformly distributed among all spatial regions of the encoded vector.

Click the link below to visualize the model

model.pdf

DataSet Selection And Preprocessing :

The dataset is collected from Kaggle, which contains various landscape images. The total number of images used to train this model is around 7000.
First, all the images are converted to 256*256, and then they are converted from RGB to CIE L*a*b* color space. The reason behind it is in L*a*b* color space, L represents luminance, and our task is, given the L channel, we have to predict the a*b color space. Then, we can reuse the L channel to concatenate with the predicted a*b channel to obtain the predicted image. Whereas if we choose RGB colorspace, then we have to estimate all three channels, i.e., R, G, and B, from grayscale; thus, the number of parameters would have been increased.

Then, the whole train dataset is split into 75-25 rule for training and validation. Pixel values of all three channels are normalized between [-1, 1]. We applied transformations to all the training datasets, like random rotation up to 20 degrees, shear transformation up to 0.2, and random horizontal flipping. These transformations will increase the robustness of our architecture to distorted images.
For Feature extraction, grayscale images are stacked such that its channel can be increased from 1 to 3, as inception_v3 requires a three-channeled image as input.

Hyperparameters :

1. Train set = 0.75 and Validation set = 0.25
2. Activation functions: ReLU, LeakyReLU with negative slope = 0.1, and Tanh (only in last layer)
3. Batch Normalization is also applied to avoid covariate shifts in images.
4. Optimizer = Adam optimizer (learning_rate = 0.001)
5. Learning Rate scheduler = ReduceLROnPlateau(mode = min, patience = 10, factor = 0.1) where scheduler is applied on Validation loss
6. Batch size = 8
7. No. of epochs = 60

Experiment Design :

First started with Conventional CNN architecture, but most of the predicted images were brownish. Then I switched to an architecture which consists of Encoder-Decoder + Feature generator network. This improved the performance, but later in addition of skip networks, BatchNormalization layer and LeakyRelu(0.1) after a few layers, further improved the performance.

To perform the demo run one must run repo/demo_run.py by providing appropriate paths of test images.

```
# replace it with actual directory
for root, dirs, files in os.walk('/kaggle/input/landscape-image-colorization/landscape Images/color'):
    for file in files:
        test_paths.append(os.path.join(root, file))
```

One can use pretrained weights and provide its path in

```
# Load the model
model = enc_dec(input_shape = 256)
state_dict = torch.load('/kaggle/working/repo/colorization_model.pth')
model.load_state_dict(state_dict)
```

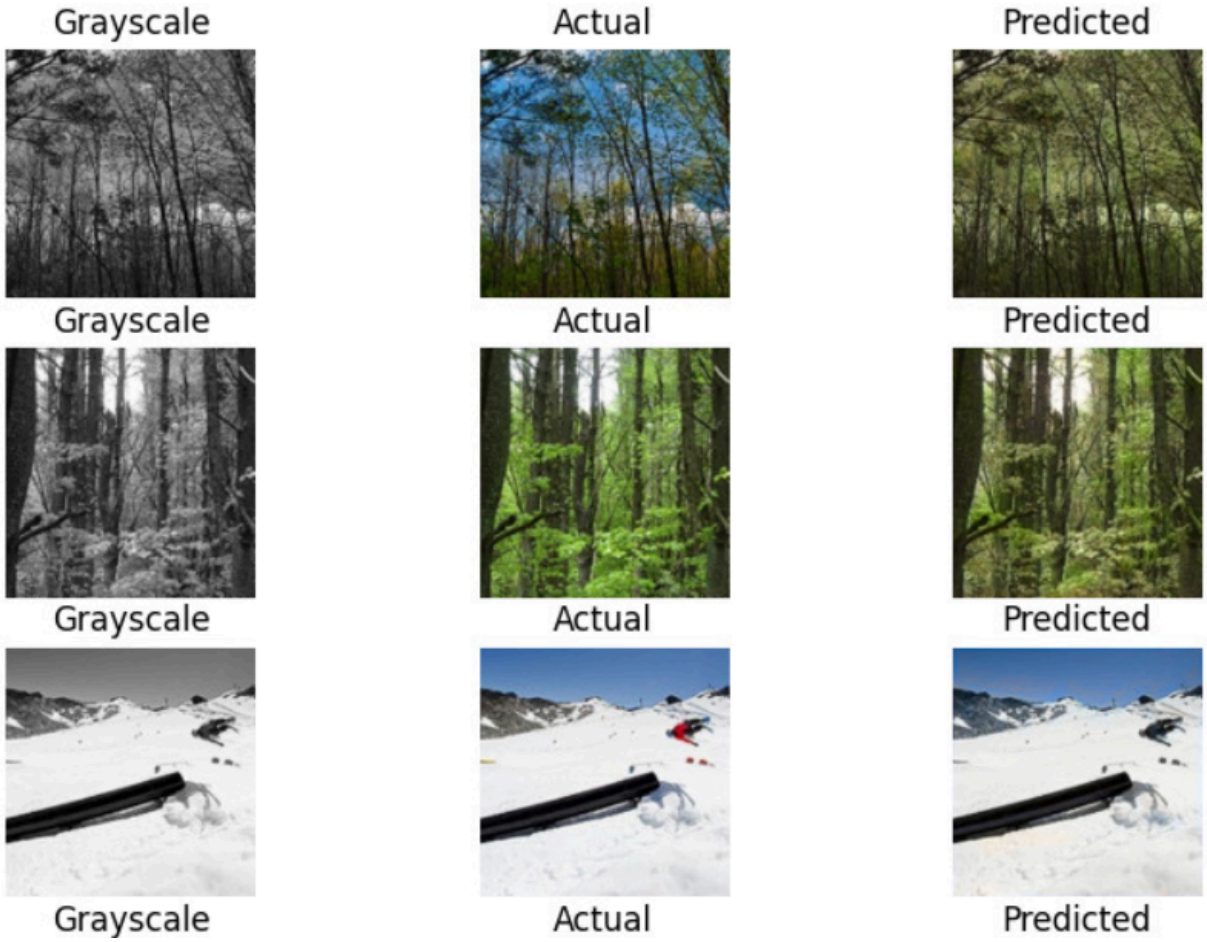
To train the model from scratch one must run repo/main.py by replacing hyperparams with preferred values.

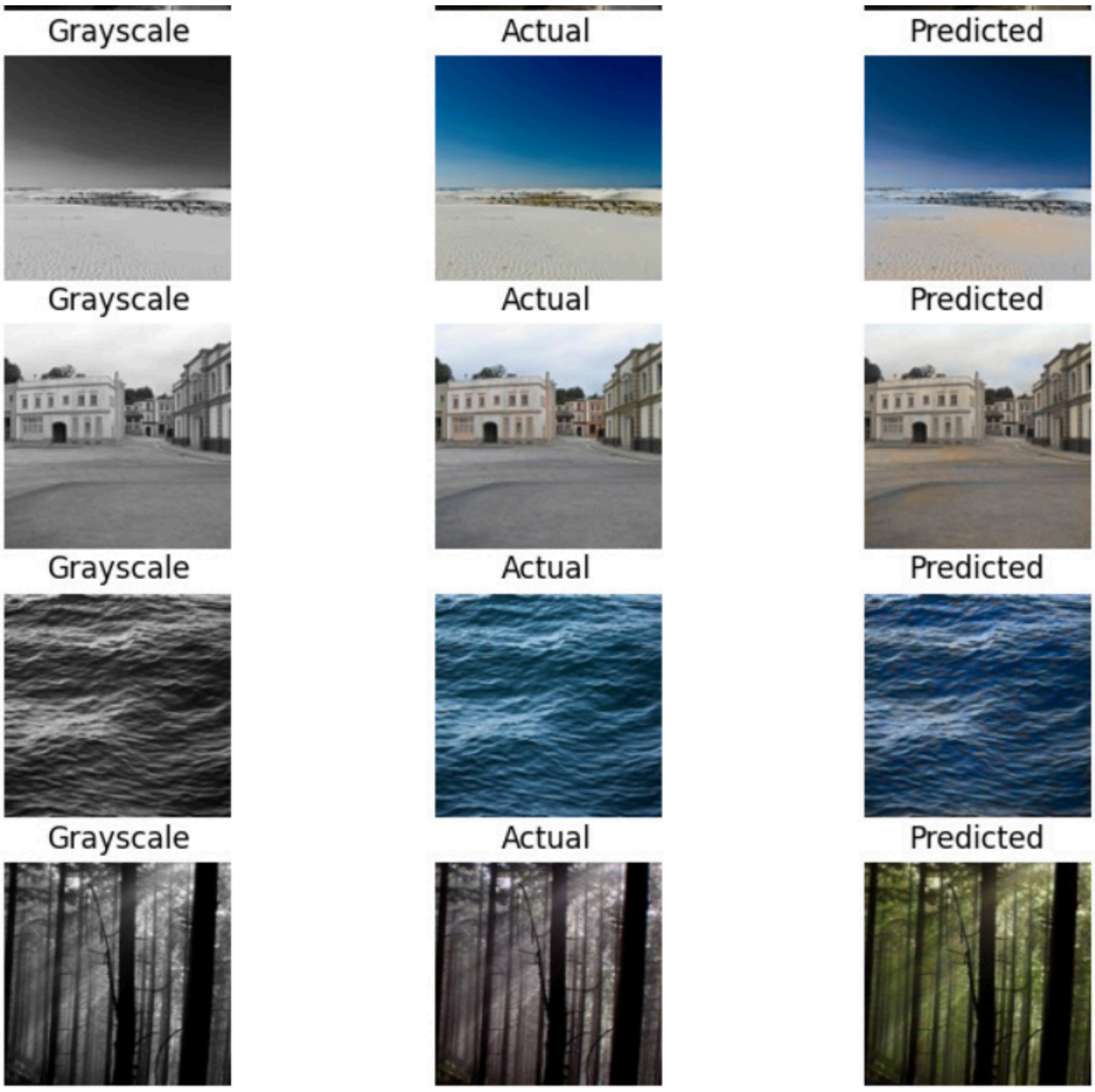
```
if __name__ == "__main__":
    # Specifying the directory containing images
    img_dir = "/kaggle/input/pokemon-image-dataset/images"  # Specify the directory containing the images
    main(img_dir, batch_size = 8, epochs = 50,
         patience = 10,learning_rate=0.001,
         factor = 0.05, test_size = 0.20)  # factor, patience are params for learning rate scheduler
```

References :

1. Colorizing black & white images with U-Net and conditional GAN — A Tutorial | Towards Data Science
2. [1603.08511] Colorful Image Colorization
3. How to colorize black & white photos with just 100 lines of neural network code
4. An Improved Encoder-Decoder CNN with Region-Based Filtering for Vibrant Colorization
5. https://doi.org/10.48550/arXiv.1712.03400

Test Images :




Grayscale
Actual
Predicted
Grayscale
Actual
Predicted
Grayscale
Actual
Predicted
Grayscale
Actual
Predicted