

A Project on
Image Recolorization using Conditional GANs

Submitted for partial fulfilment of award of

MASTER OF SCIENCE

DR. HOMI BHABHA STATE UNIVERSITY, MUMBAI



Degree in
DATA SCIENCE

By
Simran Kishan Kanojia

Under the Guidance of
Dr. Subhash Kumar
Mrs. Vasanthi Krishnan

Department of Mathematics
The Institute of Science
Mumbai – 400032

December, 2023

Declaration

We hereby declare that the project work entitled “**Image Recolorization using Conditional GANs**” carried out at the Department of Mathematics, The Institute of Science, Mumbai, is a record of an original work done by me under the guidance of Dr. Subhash Kumar, The Institute of Science, and this project work is submitted in the partial fulfilment of the requirements for the award of the degree of Master of Science in Data Science, Dr. Homi Bhabha State University. The results embodied in this report have not been submitted to any other University or Institute for the award of any degree or diploma.

SIMRAN KANOJIA

DS2216

Certificate

This is to certify that the project report entitled Image Recolorization using Conditional GANs, carried out at the Department of Mathematics, The Institute of Science, Mumbai, in partial fulfilment for the award of the degree of Master of Science in Data Science, Dr. Homi Bhabha State University, is a record of bonafide work carried out by Ms. Simran Kanojia, Seat No. DS2216 under the supervision and guidance of Dr. Subhash Kumar and Mrs. Vasanthi Krishnan during the academic year 2023-2024.

Dr. Subhash Kumar
Project Guide

External Examiner

Head of Dept.
Mathematics

Place: Mumbai

Date:

Acknowledgment

Our sincere gratitude is owed for the chance to collaborate on this research project, among other things. We would especially like to express our sincere gratitude to each other's parents for their consistent support and encouragement throughout. We are immensely grateful to have had such a fantastic support network, since their unique views and advice were invaluable to the success of our project.

The Department of Mathematics and its Head, Dr. Selby Jose, have our sincere gratitude for giving us this exceptional chance to work on this project.

Here is our chance to sincerely thank Dr. Subhash Kumar and Mrs. Vasanthi Krishnan ma'am, our project guides, for their tremendous help with our effort. Their expertise in the field has been really helpful to us as we've navigated the difficulties and complexities of this project, and their advice and criticism have allowed us to improve our concepts and method of working. They have inspired and encouraged us, and we sincerely appreciate their unwavering dedication to our achievement. We would want to express our sincere gratitude to everyone who was encountered, inspired, and helped. We sincerely appreciate the chance to collaborate with the gifted and committed members of the group. We have always been inspired and motivated by their perseverance, hard work, and dedication to greatness.

For us personally, the entire project has been an experience of development and learning, and we are appreciative of the opportunity. Finally, we would want to express our gratitude to our mentors, whose advice and encouragement enabled me to successfully handle the chances and problems that came throughout the project.

SIMRAN KISHAN KANOJIA

Contents

1. Introduction	6
2. Preliminaries	7
2.1 Basic Definitions and Results	7
2.2 Introduction to CIELAB (CIE L^*a^*b)	9
2.3 Generative Adversarial Networks (GAN) architecture	11
2.4 cGAN architecture	12
2.5 U-Net Model	13
2.6 PatchGAN	15
2.6 Loss Functions	17
3. Datasets and features	20
4. Learning Framework	22
5. Processing Pipeline	24
6. Training Results	32
7. Conclusion	38
Bibliography	39

Chapter 1

Introduction

Image colorization is a significant task in the field of computer vision, aiming to add color to grayscale images. The process involves understanding the context and description of the grayscale images to accurately predict and fill in appropriate colors. It finds applications in restoring historical photographs, enhancing visual aesthetics, and assisting in various domains like medical imaging, entertainment, and more. Traditional methods relied on manual effort or predefined color palettes, often lacking accuracy and requiring extensive human intervention. However, recent advancements in deep learning, particularly using Generative Adversarial Networks (GANs), have shown promising results in automating the image colorization process.

The objective of this project was to explore and implement a state-of-the-art approach using Conditional Generative Adversarial Networks (cGANs) for automatic image colorization. cGANs offer a compelling framework for image-to-image translation tasks i.e., pix2pix, leveraging conditional information to generate realistic colored versions of grayscale input images. The goal is to train a cGAN model and look whether it is capable of accurately inferring and applying colors to grayscale images while preserving the interpretable details and maintaining visual accuracy.

This report presents the methodology, experimentation, and outcomes of implementing cGANs for image recolorization. It details the dataset used, the architecture and training procedure of the cGAN model, the evaluation metrics employed, and an analysis of the colored output images. By leveraging the capabilities of cGANs, the project aims to contribute to the advancement of automated image colorization techniques, potentially addressing challenges in various applications that benefit from accurate and efficient recolorization of grayscale images.

Chapter 2

Preliminaries

2.1 Basic Definitions and Results

Definition 2.1.1: Neural Networks

A neural network is a method in artificial intelligence that teaches computers to process data in a way that is inspired by the human brain. It is a type of machine learning process, called deep learning, that uses interconnected nodes or neurons in a layered structure that resembles the human brain.

Definition 2.1.2: Generator

A Generator in GANs is a neural network that creates fake data to be trained on the discriminator. It learns to generate believable data. The generated examples/instances become negative training examples for the discriminator. It takes a fixed-length random vector carrying noise as input and generates a sample.

Definition 2.1.3: Discriminator

The Discriminator is a neural network that identifies real data from the fake data created by the Generator.

Definition 2.1.4: Generative Adversarial Networks

The Generative Adversarial Networks (GAN) is a machine learning model that uses deep learning to train two neural networks to compete against each other. The goal is to generate new, synthetic data that resembles some known data distribution.

Definition 2.1.5: Conditional Generative Adversarial Networks

The Conditional Generative Adversarial Network (cGAN) is a model used in deep learning, a derivative of machine learning. It enables more precise generation and discrimination of images to train machines and allow them to learn on their own. In other words, it involves the conditional generation of images by the generator model.

Definition 2.1.6: Image Recolorization

Image Recolorization with cGANs involves using conditional Generative Adversarial Networks to transform grayscale or black-and-white images into realistic and visually appealing colorized versions by leveraging a generator network trained to produce accurate color reconstructions based on the input grayscale images.

Definition 2.1.7: CIE LAB color space

The CIE LAB color space is a color model designed to mimic human vision accurately. It uses three channels: L for lightness, a for green to red, and b for blue to yellow. LAB offers perceptual uniformity and is independent of specific devices, making it ideal for color correction, analysis, and precise color matching in various industries.

Definition 2.1.8: Pix2Pix GAN

The Pix2Pix GAN is a general approach for image-to-image translation. It is based on the conditional generative adversarial network, where a target image is generated, conditional on a given input image. In this project, the Pix2Pix GAN changes the loss function so that the generated image is both plausible in the content of the target domain and is a plausible translation of the input image. It can transform images from one type to another, like turning sketches into realistic images or converting black-and-white photos into color.

2.2 Introduction to CIELAB (CIE L* a* b)

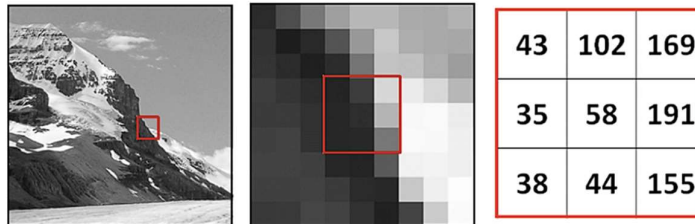
Image colorization is the process of assigning colors to a grayscale image to make it more aesthetically appealing and perceptually meaningful. As a typical technique, convolutional neural network (CNNs) has been well-studied and successfully applied to several tasks such as image recognition, image reconstruction, image generation, etc.

A CNN consists of multiple layers of small computational units that only process portions of the input image in a feed-forward fashion. Each layer is the result of applying various image filters, each of which extracts a certain feature of the input image, to the previous layer. Thus, each layer may contain useful information about the input image at different levels of abstraction.

2.2.1 Color representation

Rendering an image involves handling digital colors and understanding the fundamentals of grayscale images represented as pixel grids. Grayscale images are essentially grids of pixels. Moreover, the core logic of our neural network involves these foundational concepts.

Image to grids to pixel



Just like grayscale images, each layer in a color image has value from 0-255. The value 0 means that it has no color in that layer. If the value is 0 for all color channels, then the image pixel is black. A neural network creates a relationship between an input value and output value. In this project the network needs to find the traits that link grayscale images with colored ones. So, we should search for the features that link a grid of grayscale values to the three-color grids.

$$f \left(\begin{bmatrix} 93 & 92 & 83 & 77 & 77 \\ 92 & 77 & 77 & 77 & 92 \\ 92 & 77 & 83 & 77 & 92 \\ 77 & 77 & 77 & 92 & 92 \\ 77 & 77 & 92 & 92 & 92 \end{bmatrix} \right) = \begin{bmatrix} 83 & 92 & 83 & 77 & 77 \\ 99 & 99 & 77 & 77 & 92 \\ 99 & 77 & 83 & 77 & 92 \\ 77 & 77 & 77 & 95 & 92 \\ 77 & 77 & 95 & 92 & 92 \end{bmatrix} \begin{bmatrix} 93 & 92 & 83 & 69 & 69 \\ 92 & 69 & 69 & 77 & 92 \\ 92 & 69 & 83 & 77 & 92 \\ 69 & 69 & 77 & 92 & 92 \\ 77 & 77 & 92 & 92 & 92 \end{bmatrix} \begin{bmatrix} 83 & 92 & 83 & 77 & 77 \\ 83 & 77 & 77 & 77 & 92 \\ 92 & 77 & 83 & 75 & 85 \\ 75 & 77 & 75 & 85 & 85 \\ 75 & 75 & 85 & 85 & 85 \end{bmatrix}$$

2.2.2 Recolorization problem

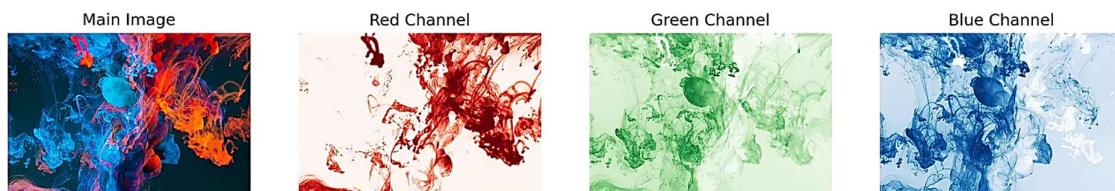
Our final output is a colored image. While, we have a grayscale image for the input and we want to predict two color layers, the a and b in Lab. To create the final color image, we'll include the L (grayscale image) we used for the input. The result will be creating a Lab image.

RGB and CIE $L^*a^*b^*$ (or just "Lab") are two different color spaces or ways of describing colors. When we load an image, we get a rank-3 (height, width, color) array with the last axis containing the color data for our image. These data represent color in RGB color space or LAB space. In RGB color space there are 3 numbers for each pixel indicating how much Red, Green, and Blue the pixel is. However, in LAB color space, we have again three numbers for each pixel but these numbers have different meanings. The first number (channel), L , encodes the Lightness of each pixel, and when we visualize this channel (the second image in the row below) it appears as a black-and-white image. The a and b channels encode how much green-red and yellow-blue each pixel is, respectively.

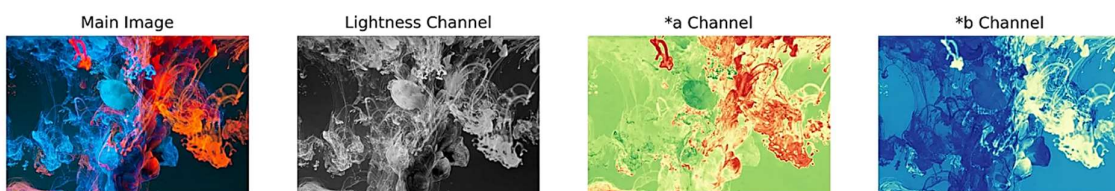
For this project, we use Lab color space instead of RGB to train the models because to train a model for recolorization, we should give it a grayscale image and hope that it will make it colorful. When using Lab, we can give the L channel to the model (which is the grayscale image) and want it to predict the other two channels (a , b) and after its prediction, we concatenate all the channels and we get our colorful image.

2.2.3 RBG vs L^*a^*b

Red, Green, and Blue channels of an image



Lightness, a , and b channels of Lab color space for an image

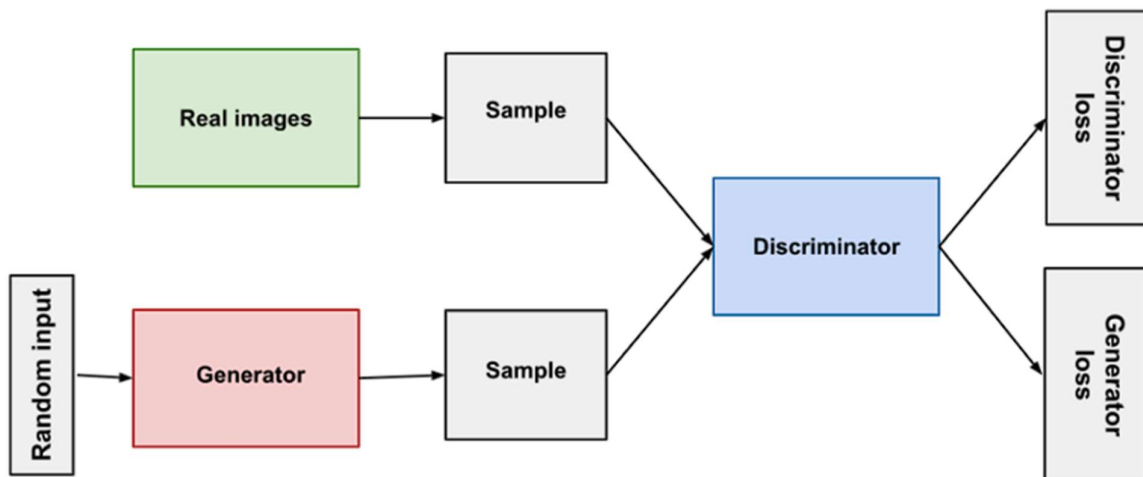


2.3 Generative Adversarial Networks (GAN) architecture

The architecture of a GAN comprises two primary components. The generator is a neural network that produces data instances, and the discriminator aims to ascertain their authenticity. The discriminator model determines whether a data instance appears authentic (i.e., potentially belonging to the original training data) or counterfeit. The generator model endeavours to deceive the discriminator and trains on more data to yield credible results.

This architecture is adversarial because the generator and discriminator operate in opposition with contrasting aims—one model strives to emulate reality while the other endeavours to identify counterfeits. These two components train concurrently, refining their capabilities over time. They can learn to recognize and replicate intricate training data such as images, audio, and video.

GAN architecture

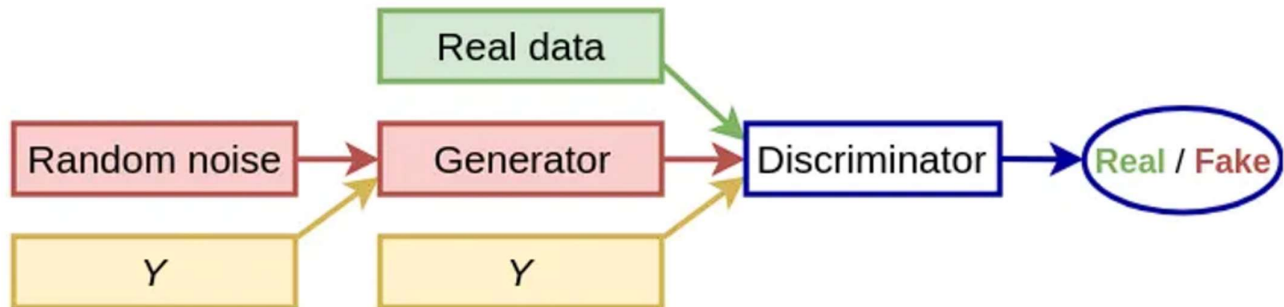


2.4 Conditional Generative Adversarial Networks (cGAN) architecture

The overall architecture of a cGAN is similar to that of a Deep Convolution GAN (DCGAN) but with minor modifications. The Discriminator architecture in a cGAN is similar to that of a Deep Convolution GAN, consisting of several convolutional layers, batch normalization layers, and Leaky ReLU activation functions. However, in a cGAN, the Discriminator receives an additional input, the conditioning information, labels, and the generated image. This extra input allows the Discriminator to consider both the image's realism and the consistency of the conditioning information when evaluating the generated image.

The Generator architecture in a cGAN is similar to that of a DCGAN, consisting of several transpose convolutional layers, batch normalization layers, and ReLU activation functions. However, in a cGAN, the Generator receives an additional input, the conditioning information, and the random noise used as the latent code. This input allows the Generator to generate images consistent with the conditioning information.

cGAN architecture



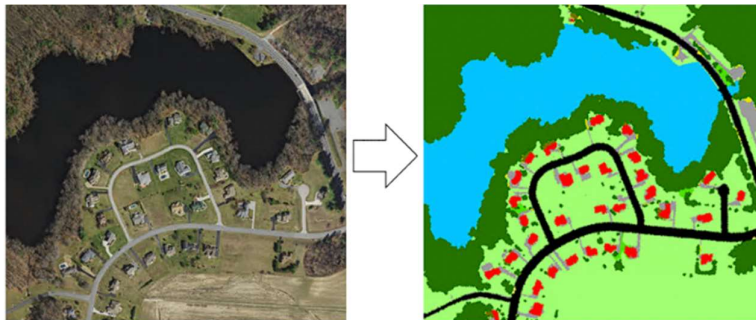
Where, Y is the conditioning information

2.5 U-Net Model

A U-Net is a specific type of generator architecture that is used for image-to-image translation tasks, like for this project objective (Image Recolorization using cGAN). It's an architecture developed by Olaf Ronneberger et al. for Biomedical Image Segmentation in 2015 at the University of Freiburg, Germany.

Semantic segmentation, also known as pixel-based classification, is an important task in which we classify each pixel of an image as belonging to a particular class. Let's say in Geographic Information Systems (GIS), segmentation can be used for land cover classification or for extracting roads or buildings from satellite imagery.

Semantic segmentation



U-Net has been one of the most popularly used approaches in any semantic segmentation task today. It is a fully convolutional neural network that is designed to learn from fewer training samples. It is an improvement over the existing FCN - Fully convolutional networks for semantic segmentation.

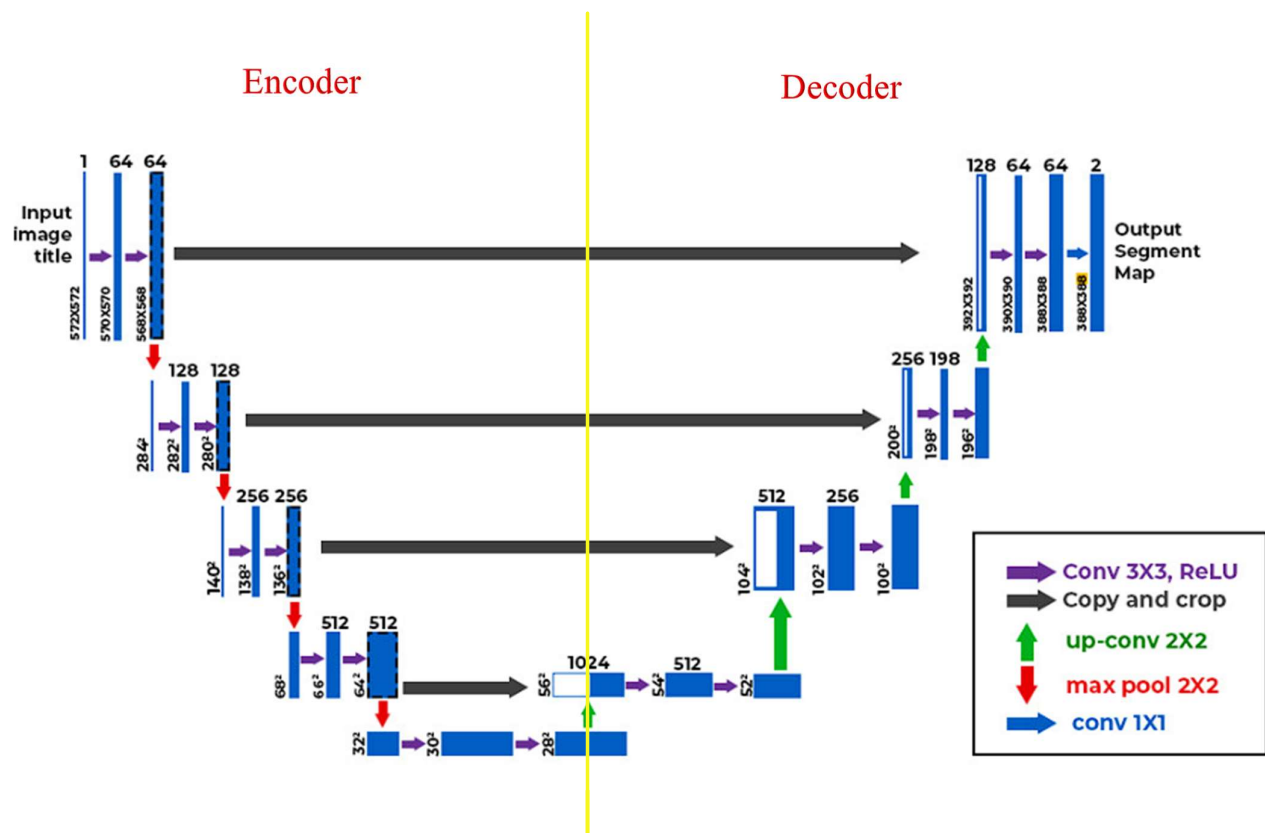
2.5.1 U-Net Architecture

U-Net is composed of an encoder and a decoder, which are mirror images of each other. The encoder compresses the input image into a lower-dimensional feature representation, while the decoder expands the feature representation back into an output image. The encoder and decoder are connected by skip connections, which help to preserve spatial information and improve the quality of the generated image.

The encoder is the first half of the architecture and is usually a pre-trained classification network like VGG/ResNet where one can apply convolution blocks followed by a maxpool downsampling to encode the input image into feature representations at multiple different levels.

The decoder is the second half of the architecture. The goal is to semantically project the discriminative features (lower resolution) learnt by the encoder onto the pixel space (higher resolution) to get a dense classification. The decoder consists of upsampling and concatenation followed by regular convolution operations.

U-Net architecture



Where, Blue boxes represent multi-channel feature maps, while white boxes represent copied feature maps. The arrows of different colors represent different operations.

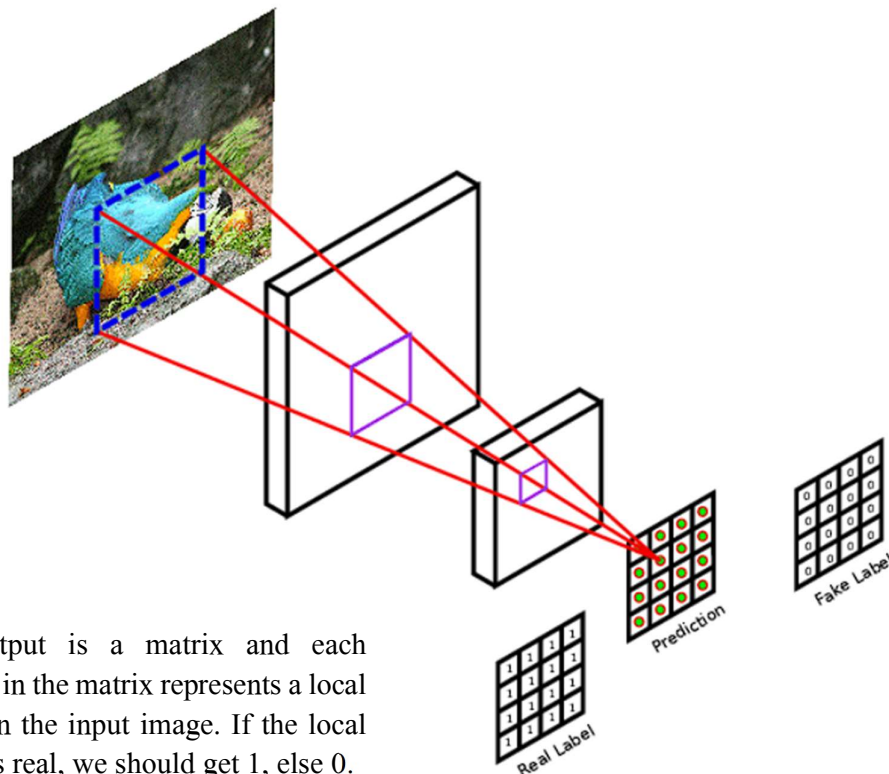
2.6 PatchGAN

PatchGAN is a type of discriminator for GAN which only penalizes structure at the scale of local image patches. The PatchGAN discriminator tries to classify if each $N \times N$ patch in an image is real or fake. This discriminator is run convolutionally across the image, averaging all responses to provide the ultimate output of D .

For example, imagine an image of a forest. A traditional discriminator would focus on the big picture and try to determine whether the image as a whole is real or fake. However, a PatchGAN discriminator would break down the image into smaller pieces, such as tree leaves or patches of grass, and try to determine whether those smaller components are real or fake. This approach allows PatchGAN to identify small and intricate details that a traditional discriminator might miss.

Another unique characteristic of PatchGAN is that it assumes independence between pixels separated by more than a patch diameter. This means that it treats each patch as a separate entity and does not take into account the context of neighbouring patches. While this may seem like a limitation, it can actually be beneficial in capturing textures or styles where the overall structure is not as important.

PatchGAN discriminator



The output is a matrix and each element in the matrix represents a local region in the input image. If the local region is real, we should get 1, else 0.

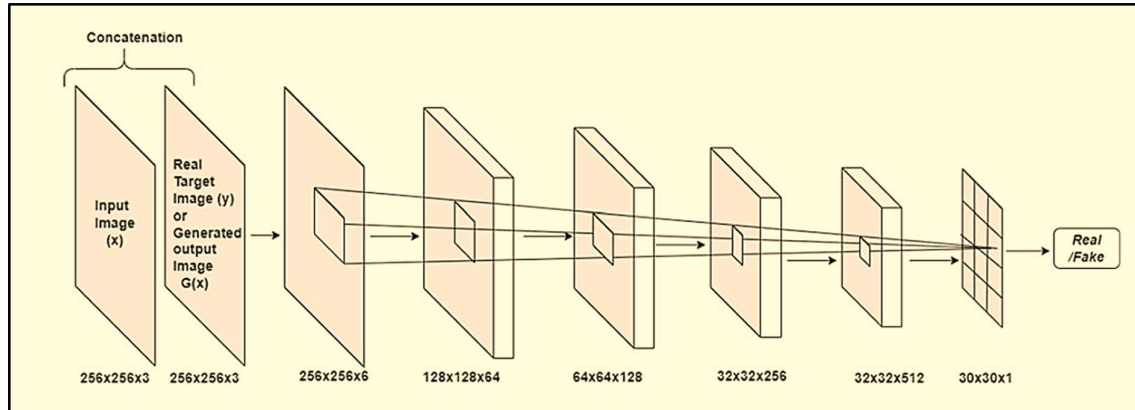
2.6.1 PatchGAN Architecture

PatchGAN operates by dividing an image into smaller patches and evaluating each patch individually to provide feedback to the generator in a cGAN. It works in following manner,

- 1) Patch Level assessment: - Instead of analysing the entire image at once, PatchGAN processes small local patches of the input image using convolutional layers.
- 2) Convolutional operations: - These layers slide over the input image, extracting features from each patch. Strided convolutions enable the network to cover the entire image with overlapping patches.
- 3) Feature extraction: - It then extracts and processes information from each patch, learning representations specific to local regions of the image.
- 4) Output layer: - The final layer of PatchGAN produces a grid of outputs, where each output corresponds to the assessment of a specific patch-sized region in the input image.
- 5) Realism evaluation: - Each output value represents the probability that the corresponding patch looks realistic or belongs to the real dataset. A sigmoid activation function is typically used to produce values between 0 and 1, indicating realism probabilities.
- 6) Feedback to the Generator: - The realism probabilities of individual patches provide detailed feedback to the generator in the cGAN. This feedback guides the generator to produce more realistic and coherent local structures and textures in the generated images.
- 7) Adversarial training: - It operates in an adversarial manner within the cGAN framework. It competes with the generator, aiming to distinguish between real and generated patches, while the generator aims to generate patches that deceive the discriminator into perceiving them as real.

By evaluating multiple local patches individually, PatchGAN enhances the discriminative power of the cGAN, providing more detailed feedback to the generator. This process contributes to generating higher-quality and visually coherent outputs in image-to-image translation tasks.

PatchGAN architecture



2.7 Loss Functions

2.7.1 Adversarial Loss (GAN Loss)

The loss function in a cGAN typically consists of two components: the generator loss and the discriminator loss. The generator loss measures how well the generator is able to fool the discriminator, while the discriminator loss measures how well the discriminator is able to distinguish between real and generated samples.

1. Generator loss -

The generator loss is usually defined as the negative log-likelihood of the discriminator's output, given the generated sample and the conditioning input. The generator tries to minimize this loss, which encourages it to generate samples that are classified as real by the discriminator.

Generator Loss ($L(G)$) is given by,

$$\mathcal{L}_{GAN}(G, D) = \mathbb{E}_{x,y}[\log D(x, y)]$$

2. Discriminator loss -

The discriminator loss measures the binary cross-entropy between the true labels (real or fake) and the discriminator's predictions. The discriminator tries to maximize this loss, which encourages it to produce accurate predictions.

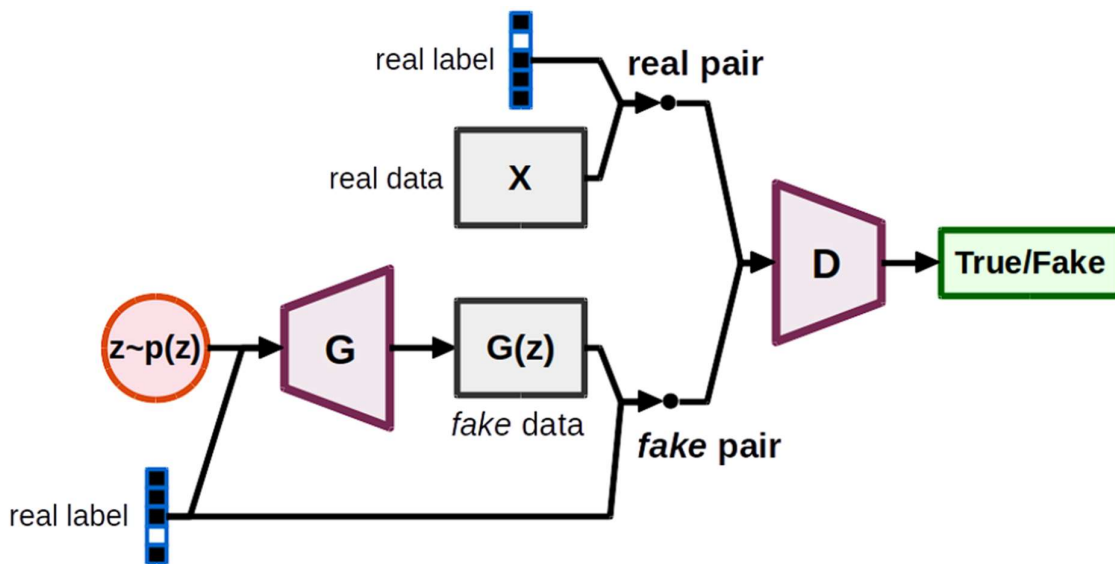
Discriminator Loss ($L(D)$) is given by,

$$\mathcal{L}_{GAN}(G, D) = \mathbb{E}_{x,y}[\log D(x, y)] + \mathbb{E}_x[\log(1 - D(x, G(x)))]$$

where,

- G represents the generator network.
- D represents the discriminator network.
- x is the input data (let's say a grayscale image for recolorization).
- y is the corresponding ground truth data (example, a colored image).

Conditional GAN architecture



2.7.2 Pixel-wise Loss (Reconstruction Loss)

These loss functions operate on individual pixels, comparing the predicted value at each pixel with the actual value provided in the ground truth data. The goal is to minimize the difference across all pixels, which in turn improves the accuracy of the model's predictions.

1. L1 Loss (Mean Absolute Error) –

This computes the average absolute differences between corresponding pixels of the generated ($G(x)$) and ground truth (y) images.

L1 loss is given by,

$$\mathcal{L}_{L1}(G) = \frac{1}{N} \sum_{i=1}^N |G(x_i) - y_i|$$

where,

- $G(x_i)$ represents the colorized image generated by the model for input x_i
- y_i is the ground truth colorized image corresponding to x_i
- N is the total number of pixels in the images being compared.

2. L2 Loss (Mean Squared Error) –

This measures the average squared differences between corresponding pixels of the generated ($G(x)$) and ground truth (y) images.

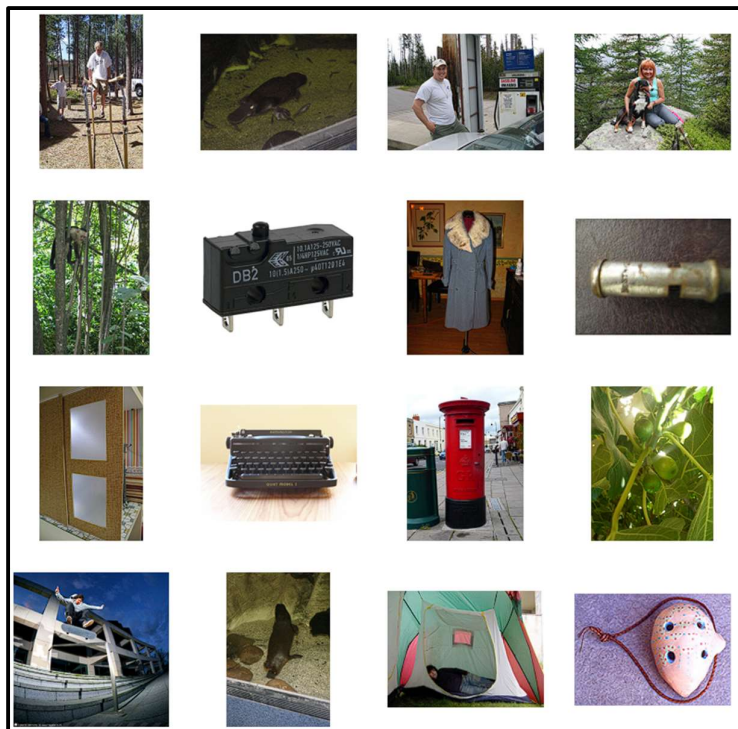
Both L1 and L2 Losses aim to minimize the pixel-wise differences between the generated and ground truth images during training. L1 Loss tends to be more robust to outliers, while L2 Loss emphasizes larger errors due to squaring differences. These losses guide the model to produce colorized images that closely resemble the ground truth images.

Chapter 3

Dataset and Features

3.1 Dataset

This project utilizes a subset of the vast ImageNet dataset, renowned for its extensive collection of images arranged according to the WordNet hierarchy. ImageNet encompasses millions of labelled images categorized into over 100,000 synsets (a set of synonyms that share a common meaning) within WordNet's structure. Primarily comprised of nouns, these synsets capture diverse concepts. Although ImageNet serves as a rich resource for supervised learning, our focus lies on image colorization which includes transforming grayscale images into their colored counterparts. While the "image-to-image translation with conditional adversarial network" paper leveraged the entire 1.3 million-image ImageNet dataset, this project evolves around on training and evaluating models using a smaller subset of 5000 images from this expansive collection. We assume 80% (4000 images) of data as the train set and 20% (1000 images) as the validation set.



3.2 Feature Extraction

Feature extraction plays a pivotal role in cGANs for colorization as it directly influences the quality of the generated images. In this process, the generator is trained to craft data that mirrors genuine data, essentially learning the underlying data distributions and extracting crucial features. Conversely, the discriminator's task involves discerning fake data, leading it to recognize essential features within authentic data. Both the generator and discriminator networks utilize convolutional layers to extract these features, aiming to capture local and hierarchical representations within the input image. These convolutional layers employ filters to identify edges, textures, and patterns, crucial for image understanding. These identified features traverse through successive layers of the network, undergoing transformations and amalgamations to ultimately produce the final output image.

```
3  def __init__(self):
4      super(Generator, self).__init__()
5      # Define the layers for feature extraction and image generation
6      self.encoder = nn.Sequential(
7          # Convolutional layers for feature extraction
8          nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1),
9          nn.ReLU(),
10         # More convolutional layers
11         # ...
12     )
13     self.decoder = nn.Sequential(
14         # Transposed convolutional layers for image generation
15         nn.ConvTranspose2d(64, 3, kernel_size=3, stride=1, padding=1),
16         nn.Tanh() # Output layer with tanh activation
17     )
18
19     def forward(self, x):
20         # Feature extraction in the encoder
21         features = self.encoder(x)
22         # Image generation in the decoder
23         output = self.decoder(features)
24         return output
25
26 # Discriminator for distinguishing real vs. fake images
27 class Discriminator(nn.Module):
28     def __init__(self):
29         super(Discriminator, self).__init__()
30         # Define the layers for feature extraction and classification
31         self.conv_layers = nn.Sequential(
32             # Convolutional layers for feature extraction
33             nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),
34             nn.LeakyReLU(0.2),
35             # More convolutional layers
36             # ...
37         )
38         self.fc_layers = nn.Sequential(
39             # Fully connected layers for classification
40             nn.Linear(64 * 256 * 256, 1), # Example linear layer
41             nn.Sigmoid() # Output layer with sigmoid activation
42         )
43
44     def forward(self, x):
45         # Feature extraction in convolutional layers
46         features = self.conv_layers(x)
47         # Flatten features for fully connected layers
48         features = torch.flatten(features, 1)
49         # Classification in fully connected layers
50         output = self.fc_layers(features)
51         return output
52
```

Chapter 4

Learning Framework

4.1 Proposed methodology

The objective is training a conditional Generative Adversarial Network (cGAN) to produce a colored image from a grayscale input generated by Lab. This is achieved by employing a UNET architecture as the generator and utilizing PatchGAN for the discriminator.

The generator network operates with two inputs: a random noise vector and the grayscale image, leveraging this information to create a colored rendition of the input. Meanwhile, the discriminator network receives both the original grayscale image and the generated colored image, aiming to distinguish the authenticity of the generated image by classifying it as real or fake.

4.2 Feature Extraction

During training, the discriminator sees both real images and generated images, and it develops an internal representation of the features that distinguish real images from fake ones. This internal representation can be thought of as a feature extractor, as it extracts and represents the important features of the input images.

4.3 Image Recolorization

In our project, we trained and evaluated our models using 5000 samples from the dataset. After splitting the data, we trained our model for 45 epochs. In the result part we show how increasing numbers of epochs affected directly the output and step by step the gray inputs become colorful.

4.4 Pre-trained Model

We used a pre-trained model for classification as the backbone of the generator's downsampling path in a cGAN for colorization can be a good choice as it can provide a strong foundation of features to build upon. When using a pre-trained model, the model has already learned to recognize various visual features from large amounts of data. These features can be used as a starting point for the generator in a cGAN, allowing it to quickly learn the mapping between grayscale image and color image. This can be particularly useful when the training data set for colorization is small or limited.

The pre-trained model can provide a rich set of features that would be difficult to learn from scratch. It is important to keep in mind that the pre-trained model i.e., ResNet18 was designed for a different task (classification) and its features may not be optimal for colorization, so we add fine-tuning of the pre-trained model to adapt it to the colorization task by removing the last two layers (GlobalAveragePooling and a Linear layer for the ImageNet classification task) and uses this backbone to build a U-Net with the needed output channels (2 in our case) and with an input size of 256x256.

4.5 Model Evaluation (Losses)

We used the 'vanilla' GAN loss for our project and registered some constant tensors as the 'real' and 'fake' labels. Then when we call this module, it makes an appropriate tensor full of zeros or ones (according to what we need at the stage) and computes the loss.

Also, we use L1 loss as well and compute the distance between the predicted two channels and the target two channels and multiply this loss by a coefficient which in our case ($\lambda = 100$) to balance the two losses and then add this loss to the adversarial loss. Then we call the backward method of the loss.

Chapter 5

Processing pipeline

5.1 Transforming colored images to grayscale

Since the dataset taken for this project contained colored images throughout, there was a need to transform those images to grayscale images. For that we used Lab color space instead of RGB to train the models because to train a model for recolorization, we should give it a grayscale image and hope that it will make it colorful. When using Lab, we can give the L channel to the model (which is the grayscale image) and want it to predict the other two channels (a, b) and after its prediction, we concatenate all the channels and we get our colorful image.

So, we resize the images and flipping horizontally (flipping only if it is training set), and read the RGB image. Then convert it to Lab color space and separate the first (grayscale) channel and the color channels. Then define the data loaders.

```
1 SIZE = 256
2 class ColorizationDataset(Dataset):
3     def __init__(self, paths, split='train'): # Determining the transformations based on the split type
4         if split == 'train':
5             # For training data: resizing and adding random horizontal flips for augmentation
6             self.transforms = transforms.Compose([
7                 transforms.Resize((SIZE, SIZE), Image.BICUBIC), # Resize the image
8                 transforms.RandomHorizontalFlip(), # A little data augmentation!
9             ])
10        elif split == 'val':
11            # For validation data: resizing without augmentation
12            self.transforms = transforms.Resize((SIZE, SIZE), Image.BICUBIC)
13
14        self.split = split # Storing the split type
15        self.size = SIZE # Storing the image size
16        self.paths = paths # Storing paths to images
17
18    def __getitem__(self, idx):
19        # Loading the image using its path and converting it to RGB
20        img = Image.open(self.paths[idx]).convert("RGB")
21        img = self.transforms(img)
22        img = np.array(img) # Converting the image to a NumPy array
23        img_lab = rgb2lab(img).astype("float32") # Converting RGB to L*a*b
24        img_lab = transforms.ToTensor()(img_lab) # Converting to PyTorch tensor
25        L = img_lab[[0], ...] / 50. - 1. # Extracting the L channel and normalizing it to values between -1 and 1
26        ab = img_lab[[1, 2], ...] / 110. # Extracting the ab channels and normalizing them to values between -1 and 1
27
28        return {'L': L, 'ab': ab}
29
30    def __len__(self): # Returning the total number of images in the dataset
31        return len(self.paths)
32
33 # Function to create data loaders for the colorization dataset
34 def make_data loaders(batch_size=16, n_workers=0, pin_memory=True, **kwargs):
35     # Creating an instance of the ColorizationDataset class with provided arguments
36     dataset = ColorizationDataset(**kwargs)
37     # Creating a DataLoader to load data in batches for training/validation
38     dataloader = DataLoader(dataset, batch_size=batch_size, num_workers=n_workers,
39                             pin_memory=pin_memory)
40     return dataloader
```


5.2 Implementing the UNET Autoencoder (generator)

The generator network architecture in the pix2pix cGAN utilizes a U-Net structure, which consists of a U-shaped encoder-decoder architecture. Leaky ReLU activation functions are used in the encoder with a slope of 0.2, while the decoder employs regular ReLU activations. Batch normalization layers follow most convolutional layers, excluding the initial C64 layer. The encoder part of the network is constructed with the following sequence:

C64-C128-C256-C512-C512-C512-C512

where,

- C64 implies convolutional layer with 64 filters or channels
- C128 implies convolutional layer with 128 filters or channels
- C256 implies convolutional layer with 256 filters or channels
- C512 implies convolutional layer with 512 filters (repeated multiple times)

This U-Net which we built has 8 layers down, so if we start with a 256 by 256 image, in the middle of the U-Net we will get a 1 by 1 ($256 / 2^8$) image and then it gets up-sampled to produce a 256 by 256 image (with two channels).

The decoder section involves upsampling the previously downsampled images and establishing skip connections, connected to the U-Net model. Convolutional transpose layers with batch normalization and optional dropout layers are employed for upsampling. Dropout serves as a regularization technique to prevent overfitting, enhance training stability, and encourage robust feature learning within GANs. In this project, dropout is used solely within the Generator for introducing noise. The decoder block of the generator network follows this architecture:

CD512-CD512-CD512-C512-C256-C128-C64

where,

- CD512 implies convolutional layer followed by downsampling with 512 filters (repeated multiple times)
- C512 implies convolutional layer with 512 filters
- C256 implies convolutional layer with 256 filters
- C128 implies convolutional layer with 128 filters
- C64 implies convolutional layer with 64 filters

After the last layer in the decoder, a convolution is applied to map to the number of output channels (2 channels in colorization task), followed by a Tanh function. As an exception to the above notation of encoder-decoder architecture, Batch- Norm is not applied to the first C64 layer in the encoder. All ReLUs in the encoder are leaky, with slope 0.2, while ReLUs in the decoder are not leaky. The U-Net architecture is identical except with skip connections between each layer i in the encoder and layer $n-i$ in the decoder, where n is the total number of layers. The skip connections concatenate activations from layer i to layer $n - i$. This changes the number of channels in the decoder. The modified U-Net decoder is as follows:

CD512-CD1024-CD1024-C1024-C1024-C512-C256-C128.

Following the last decoder layer, a convolutional operation is applied to adjust the number of output channels (two channels in this task). The final layer uses the hyperbolic tangent (tanh) activation function due to its range of -1 to 1, suitable for probability-like predictions, as well as its smoothness and non-linearity.

The generator network requires two inputs: a random noise vector (z) and a condition, which for image colorization is the grayscale image (x). Utilizing these inputs, the generator (G) network produces a colorized image.

```

1 # Custom U-Net block module
2 class UnetBlock(nn.Module):
3     def __init__(self, nf, ni, submodule=None, input_c=None, dropout=False,
4                 innermost=False, outermost=False):
5         super().__init__()
6         self.outermost = outermost # Flags for identifying outermost and innermost blocks
7         if input_c is None: input_c = nf # Set input channels to nf if not provided
8         # Downward convolutional layer, ReLU activation, and batch normalization
9         downconv = nn.Conv2d(input_c, ni, kernel_size=4, stride=2, padding=1, bias=False)
10        downrelu = nn.LeakyReLU(0.2, True)
11        downnorm = nn.BatchNorm2d(ni)
12        uprelu = nn.ReLU(True) # Upward ReLU activation
13        upnorm = nn.BatchNorm2d(nf) # Upward batch normalization
14
15        if outermost:
16            # Upward convolution for the outermost block, Tanh activation for output
17            upconv = nn.ConvTranspose2d(ni * 2, nf, kernel_size=4, stride=2, padding=1)
18            down = [downconv]
19            up = [uprelu, upconv, nn.Tanh()]
20            model = down + [submodule] + up # Define the model layers for outermost block
21        elif innermost:
22            # Upward convolution for innermost block without downsampling
23            upconv = nn.ConvTranspose2d(ni, nf, kernel_size=4, stride=2, padding=1, bias=False)
24            down = [downrelu, downconv]
25            up = [uprelu, upconv, upnorm]
26            model = down + up # Define the model layers for innermost block
27        else:
28            # Upward convolution for intermediate blocks, optional dropout
29            upconv = nn.ConvTranspose2d(ni * 2, nf, kernel_size=4, stride=2, padding=1, bias=False)
30            down = [downrelu, downconv, downnorm]
31            up = [uprelu, upconv, upnorm]
32            if dropout: up += [nn.Dropout(0.5)] # Optional dropout layer
33            model = down + [submodule] + up # Define the model layers for intermediate blocks
34        self.model = nn.Sequential(*model) # Define the sequential model for the block
35
36    def forward(self, x):
37        if self.outermost:
38            return self.model(x) # Return the model output for the outermost block
39        else:
40            return torch.cat([x, self.model(x)], 1) # Concatenate input and block output for other blocks
41
42 # U-Net architecture
43 class Unet(nn.Module):
44     def __init__(self, input_c=1, output_c=2, n_down=8, num_filters=64):
45         super().__init__()
46         # Initialize the innermost block
47         unet_block = UnetBlock(num_filters * 8, num_filters * 8, innermost=True)
48         # Create n_down - 5 blocks for the intermediate layers
49         for _ in range(n_down - 5):
50             unet_block = UnetBlock(num_filters * 8, num_filters * 8, submodule=unet_block, dropout=True)
51         out_filters = num_filters * 8
52         # Create 3 blocks for the upward path
53         for _ in range(3):
54             unet_block = UnetBlock(out_filters // 2, out_filters, submodule=unet_block)
55             out_filters //= 2
56         # Create the outermost block for the final output
57         self.model = UnetBlock(output_c, out_filters, input_c=input_c, submodule=unet_block, outermost=True)
58
59    def forward(self, x):
60        return self.model(x) # Return the output from the U-Net model

```

5.3 Implementing the PatchGAN (Discriminator)

For constructing the discriminator, we use PatchGAN which takes in both the grayscale image and colored image and evaluates whether the colored image is a valid transformation of the grayscale input. The PatchGAN has an effective receptive field of 70x70. Every single cell in the output array tells the probability of a 70x70 patch being real or fake. The discriminator structure will follow the pattern build of

C64-C128-C256-C512

We implement a model by stacking blocks of Conv-BatchNorm-LeakyReLU to decide whether the input image is fake or real. Besides, the first and last blocks do not use normalization and the last block has no activation function (It is embedded in the loss function).

```
1 # Patch Discriminator module
2 class PatchDiscriminator(nn.Module):
3     def __init__(self, input_c, num_filters=64, n_down=3):
4         super().__init__()
5         # Initialize the model as a list to hold layers
6         model = [self.get_layers(input_c, num_filters, norm=False)] # Start with the initial convolutional layers
7         # Create downsampling blocks for the discriminator
8         model += [self.get_layers(num_filters * 2 ** i, num_filters * 2 ** (i + 1),
9                                s=1 if i == (n_down-1) else 2) # Adjust stride for the last block in the loop
10                for i in range(n_down)]
11        # Add the final output layer without normalization or activation
12        model += [self.get_layers(num_filters * 2 ** n_down, 1, s=1, norm=False, act=False)]
13        # Create a sequential model using the defined layers
14        self.model = nn.Sequential(*model)
15
16    # Helper function to create convolutional layers with optional normalization and activation
17    def get_layers(self, ni, nf, k=4, s=2, p=1, norm=True, act=True):
18        layers = [nn.Conv2d(ni, nf, k, s, p, bias=not norm)] # Convolutional layer without bias if using normalization
19        if norm:
20            layers += [nn.BatchNorm2d(nf)] # Add batch normalization if specified
21        if act:
22            layers += [nn.LeakyReLU(0.2, True)] # Add LeakyReLU activation if specified
23        return nn.Sequential(*layers) # Return the sequential layers
24
25    def forward(self, x):
26        return self.model(x) # Forward pass through the model to get the discriminator's output
```

5.4 Analysing the loss function

The loss functions for a standard Generative Adversarial Network (GAN) and a Conditional Generative Adversarial Network (cGAN) are similar, but with some important differences. The cGAN loss function includes the additional condition (Label) as input to both the generator and discriminator, allowing the cGAN to take into account the additional information specified by the condition when learning to generate new images. The L1 loss alone leads to reasonable but blurry results. The cGAN alone ($\lambda = 0$) gives much sharper results but introduces visual artifacts on certain applications. Adding both terms together ($\lambda = 100$) reduces these artifacts.

Loss formula for GAN –

$$\mathcal{L}_{GAN}(G, D) = \mathbb{E}_y[\log D(y)] + \mathbb{E}_{x,z}[\log(1 - D(G(x, z)))]$$

Loss formula for cGAN –

$$\mathcal{L}_{cGAN}(G, D) = \mathbb{E}_{x,y}[\log D(x, y)] + \mathbb{E}_{x,z}[\log(1 - D(x, G(x, z)))]$$

Loss of L1 –

$$\mathcal{L}_{L1}(G) = \mathbb{E}_{x,y,z}[\|y - G(x, z)\|_1]$$

Our final objective is –

$$G^* = \arg \min_G \max_D \mathcal{L}_{cGAN}(G, D) + \lambda \mathcal{L}_{L1}(G)$$

This is a handy class we can use to calculate the GAN loss of our final model. In the **init** we decide which kind of loss we're going to use (which will be "vanilla" in our project) and register some constant tensors as the "real" and "fake" labels. Then when we call this module, it makes an appropriate tensor full of zeros or ones (according to what we need at the stage) and computes the loss.

```
1 class GANLoss(nn.Module):
2     def __init__(self, gan_mode='vanilla', real_label=1.0, fake_label=0.0):
3         super().__init__()
4         self.register_buffer('real_label', torch.tensor(real_label))
5         self.register_buffer('fake_label', torch.tensor(fake_label))
6         if gan_mode == 'vanilla':
7             self.loss = nn.BCEWithLogitsLoss()
8         elif gan_mode == 'lsgan':
9             self.loss = nn.MSELoss()
10
11     def get_labels(self, preds, target_is_real):
12         if target_is_real:
13             labels = self.real_label
14         else:
15             labels = self.fake_label
16         return labels.expand_as(preds)
17
18     def __call__(self, preds, target_is_real):
19         labels = self.get_labels(preds, target_is_real)
20         loss = self.loss(preds, labels)
21         return loss
```

For our final models, we provide noise only in the form of dropout, applied on several layers of our generator at both training and test time. Despite the dropout noise, we observe only minor stochasticity in the output of our nets.

5.5 Model Initialisation

This is our approach to model initialization. We'll set the weights, key hyperparameters in the article, with the following values:

- Mean = 0.0
- Standard Deviation = 0.02

```
1 def init_weights(net, init='norm', gain=0.02):
2
3     def init_func(m):
4         classname = m.__class__.__name__
5         if hasattr(m, 'weight') and 'Conv' in classname:
6             if init == 'norm':
7                 nn.init.normal_(m.weight.data, mean=0.0, std=gain)
8             elif init == 'xavier':
9                 nn.init.xavier_normal_(m.weight.data, gain=gain)
10            elif init == 'kaiming':
11                nn.init.kaiming_normal_(m.weight.data, a=0, mode='fan_in')
12
13            if hasattr(m, 'bias') and m.bias is not None:
14                nn.init.constant_(m.bias.data, 0.0)
15        elif 'BatchNorm2d' in classname:
16            nn.init.normal_(m.weight.data, 1., gain)
17            nn.init.constant_(m.bias.data, 0.)
18
19    net.apply(init_func)
20    print(f"model initialized with {init} initialization")
21    return net
22
23 def init_model(model, device):
24     model = model.to(device)
25     model = init_weights(model)
26     return model
```

Main Model –

- **Step1:** This class brings together all the previous parts and implements a few methods to take care of training our complete model.
- **Step2:** In the init our generator and discriminator using the previous functions and classes which defined and also initialize them with `init_model` function.
- **Step 3:** using module's forward method (only once per iteration (batch of training set)) and store the outputs in `fake_color` variable of the class.
- **Step4:** first train the discriminator by using `backward_D` method in which we feed the fake images produced by generator to the discriminator and label them as fake.

- **Step5:** Then feeding a batch of real images from training set to the discriminator and label them as real also adding up the two losses for fake and real and take the average and then call the backward on the final loss.
- **Step6:** Training the generator in backward_G method feed the discriminator the fake image and try to fool it by assigning real labels to them and calculating the adversarial loss.

We used minibatch SGD and apply the Adam solver, with a learning rate of 0.0002, and momentum parameters $\beta_1 = 0.5$, $\beta_2 = 0.999$. These parameters are obtained by a research paper.

```

1  class MainModel(nn.Module):
2      def __init__(self, net_G=None, lr_G=2e-4, lr_D=2e-4,
3                  beta1=0.5, beta2=0.999, lambda_L1=100.):
4          super().__init__()
5
6          self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
7          self.lambda_L1 = lambda_L1
8
9          if net_G is None:
10             self.net_G = init_model(Unet(input_c=1, output_c=2, n_down=8, num_filters=64), self.device)
11         else:
12             self.net_G = net_G.to(self.device)
13             self.net_D = init_model(PatchDiscriminator(input_c=3, n_down=3, num_filters=64), self.device)
14             self.GANcriterion = GANLoss(gan_mode='vanilla').to(self.device)
15             self.L1criterion = nn.L1Loss()
16             self.opt_G = optim.Adam(self.net_G.parameters(), lr=lr_G, betas=(beta1, beta2))
17             self.opt_D = optim.Adam(self.net_D.parameters(), lr=lr_D, betas=(beta1, beta2))
18
19         def set_requires_grad(self, model, requires_grad=True):
20             for p in model.parameters():
21                 p.requires_grad = requires_grad
22
23         def setup_input(self, data):
24             self.L = data['L'].to(self.device)
25             self.ab = data['ab'].to(self.device)
26
27     > def forward(self): ...
28
29     > def backward_D(self): ...
30
31     > def backward_G(self): ...
32
33     > def optimize(self):
34         self.forward()
35         self.net_D.train()
36         self.set_requires_grad(self.net_D, True)
37         self.opt_D.zero_grad()
38         self.backward_D()
39         self.opt_D.step()
40
41         self.net_G.train()
42         self.set_requires_grad(self.net_D, False)
43         self.opt_G.zero_grad()
44         self.backward_G()
45         self.opt_G.step()

```



5.5 Model Training


We proceed with model training using six distinct sets of images.

- Each training epoch consumes approximately 1 to 2 minutes when executed on Colab, a notable aspect of the computational time involved, provided NVIDIA GeForce GTX 1650 (4 GB GDDR6 dedicated) was used to perform this task.
- To accommodate the lengthy training duration, we perform training for 30 epochs, providing a substantial training period.
- Throughout each step of the training process, discernible changes and adaptations within the model become evident.
- Progressively training the model for over 50 epochs unveils a gradual improvement, yielding increasingly meaningful outcomes.
- For achieving optimal or near-perfect results, an extended training duration of 100 epochs becomes necessary, ensuring comprehensive learning and refinement within the model's performance.

```
1 results = []
2 show=False
3 def train_model(model, train_dl, epochs, display_every=1):
4     data = next(iter(val_dl)) # getting a batch for visualizing the model output after fixed intervals
5     for e in range(epochs):
6         print(e)
7         loss_meter_dict = create_loss_meters() # function returning a dictionary of objects to
8         i = 0                                     # log the losses of the complete network
9         for data in tqdm(train_dl):
10             model.setup_input(data)
11             model.optimize()
12             update_losses(model, loss_meter_dict, count=data['L'].size(0)) # function updating the log objects
13             i += 1
14             if i in [1,5,10,50,100,249]:
15                 visualize(model, data, save=False) # function displaying the model's outputs
16                 show=True
17             else:
18                 show=False
19             if show:
20                 print(f"\nEpoch {e+1}/{epochs}")
21                 print(f"Iteration {i}/{len(train_dl)}")
22                 results.append(log_results(loss_meter_dict,show=show)) # function to print out the losses
23
24
25 model = MainModel()
26 train_model(model, train_dl, 45)
```

model initialized with norm initialization
model initialized with norm initialization
0

100%  50/50 [00:57<00:00, 1.60s/it]



Chapter 6

Training Results

6.1 Performance of our Main Model

Our model has been trained using the ImageNet-1000 dataset. In order to prevent the network from being biased toward particular color tones, we arbitrarily chose a portion of the dataset without placing any restrictions on the image categories. For every model, the input image size was set at 256 by 256 pixels. All models are trained using PyTorch software packages accelerated on GPU hardware (NVIDIA GeForce GTX 1650 system, T4 GPUs Google Colab) utilizing the Adam optimizer. Initially, we evaluate the primary model that was trained for 45 epochs using 4000 training data and 1000 validation using the ImageNet-1000 dataset. To add a little data augmentation, we also rotated the training photos randomly.

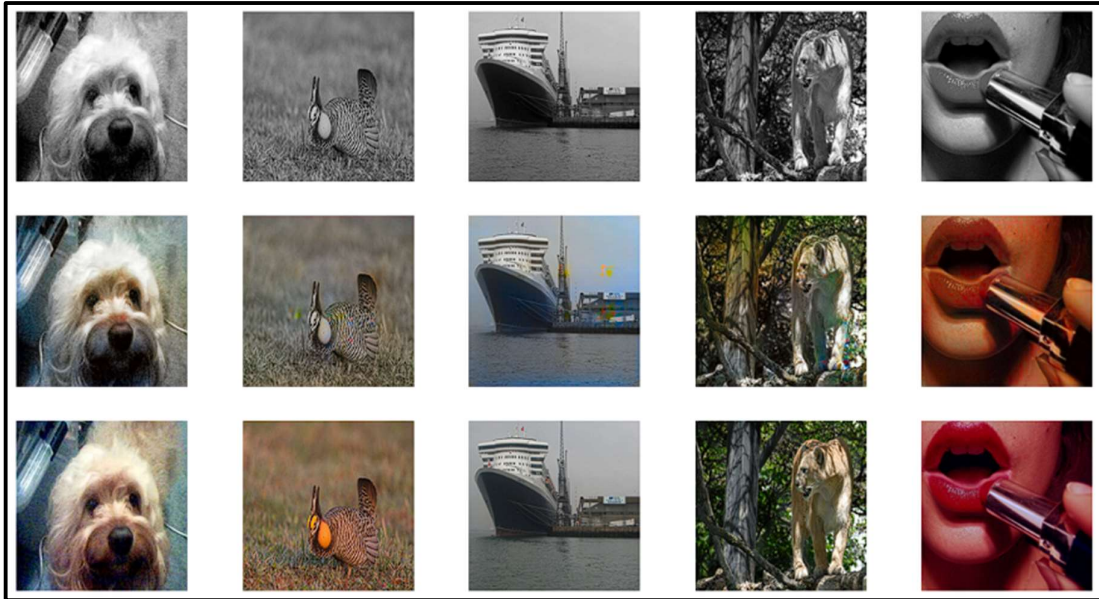
Evaluation of the main model for 45 epochs



In a cGAN, both the generator and discriminator networks undergo training across multiple epochs to understand the relationship between grayscale and color images. Extending the number of epochs enables the network to grasp more deeper connections, leading to improved colorization accuracy. Training for more epochs or using additional data typically results in more realistic outcomes. However, due to resource constraints, our training is limited to 45 epochs in this project.

By the 45th epoch, the model showed competence in distinguishing and colorizing certain objects. However, it struggled with smaller objects and deeper details, exhibiting limitations in accuracy.

Result of training a set of images for 45 Epochs

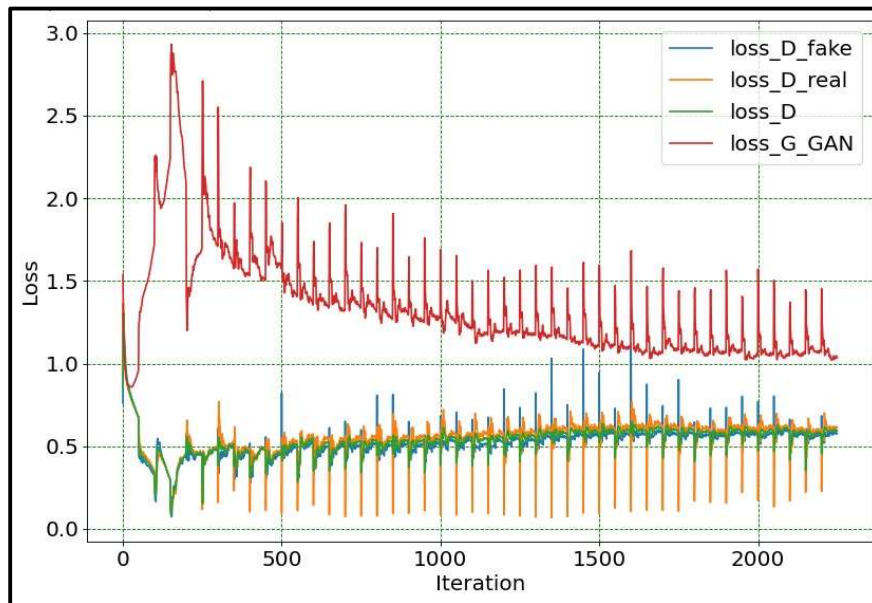


Trained our model in 45 epochs with 4000 training images and 1000 images for validation. The first row represents grayscale, the second row is generated image by our model, and the third row is a ground truth image.

To measure the accuracy of colorization we need human test which means that the human brain should not be able to discriminate between real and generated images. In the illustration above we can easily realize that some of the generated images are not real. Also, it is hard to guess that two of generated images (third and fourth image from left to right second row) are fake or real. Therefore, by training this model in more epochs and also by increasing the data we can get better results.

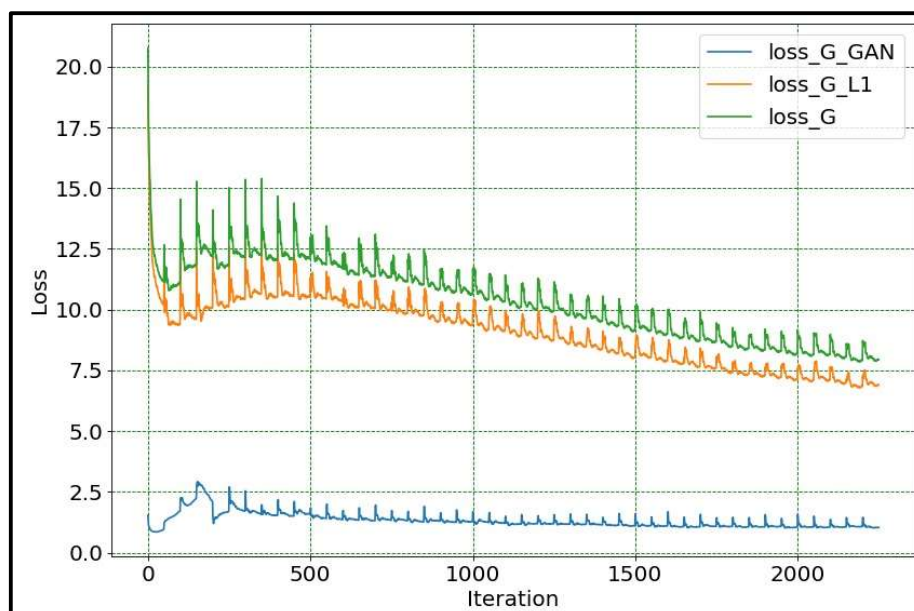
To get better outcomes, training the model for longer and adding more data can make the model pick up smaller details and improve how realistic the colorizations look.

GRAPH 1 - The Discriminator and G_GAN loss for 45 epochs

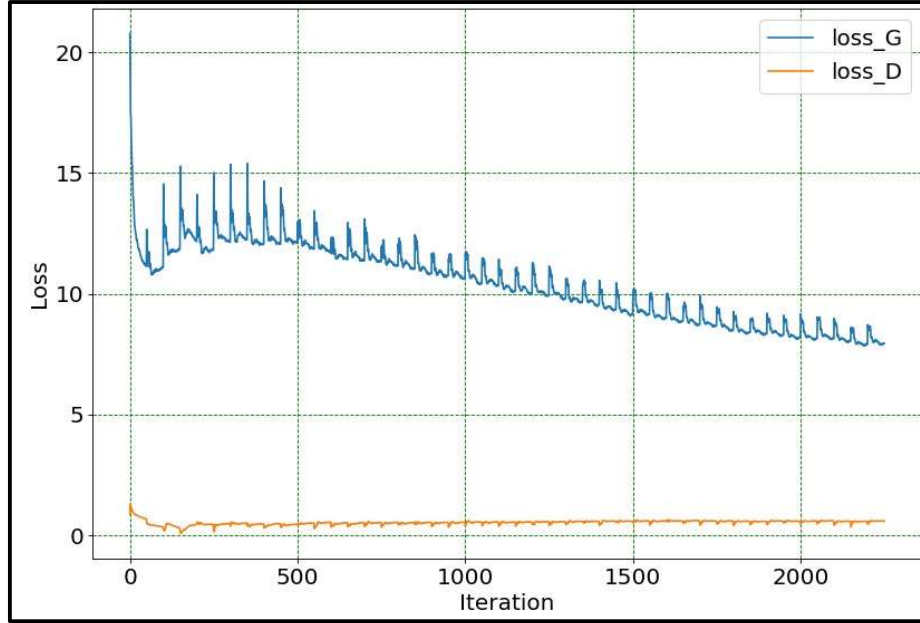


During the implementation of cGAN for 45 epochs, we calculate the GAN losses of our final model. Moreover, to monitor the learning process of our models, model losses were plotted with respect to the number of iterations. In the above graph (GRAPH 1), we can see the discriminator losses which contain fake loss and real loss. By averaging the fake and real loss we can get the loss of the discriminator. Also, we can see that the `loss_G_GAN` which we defined in equation is reduced when the number of iterations is increased.

GRAPH 2 - The Generator loss for 45 epochs



GRAPH 3 - Generator and Discriminator losses for 45 epochs



It can be seen that by increasing the number of iterations the loss_G, which we defined in equation,

$$G^* = \arg \min_G \max_D \mathcal{L}_{cGAN}(G, D) + \lambda \mathcal{L}_{L1}(G)$$

is getting close to loss_D. Besides, in the GRAPH 3 we can see the final gan and discriminator loss. Due to the limited resources as mentioned before, it was not possible to train the model for more epochs. However, we can realize the reduction of the GAN loss after 4000 iterations which is equal to epoch 16.

6.2 Testing of our Main Model using COCO 2017 dataset

Using the COCO (Common Objects in Context) 2017 dataset to test our main model of cGAN's colorization performance provides a strong evaluation method. This dataset, similar to ImageNet, helps assess how well the model adds color to images. Testing on COCO showcases the model's adaptability and accuracy across diverse scenes and objects, offering insights into its effectiveness beyond its training data.

Testing of main cGAN model on COCO 2017 dataset



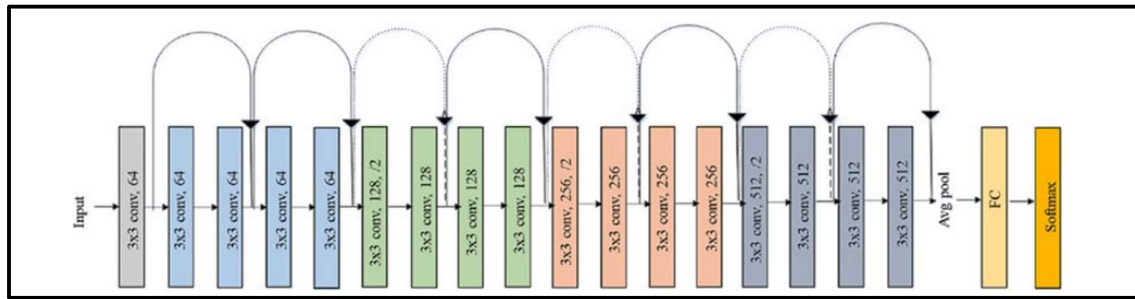
Test results of our model with the COCO 2017 dataset. The first row is grayscale, the second row is generated image by the model, and the third row is the ground truth image.

6.3 Pre-trained ResNet18 performance on ImageNet

6.3.1 Fine tuning ResNet-18

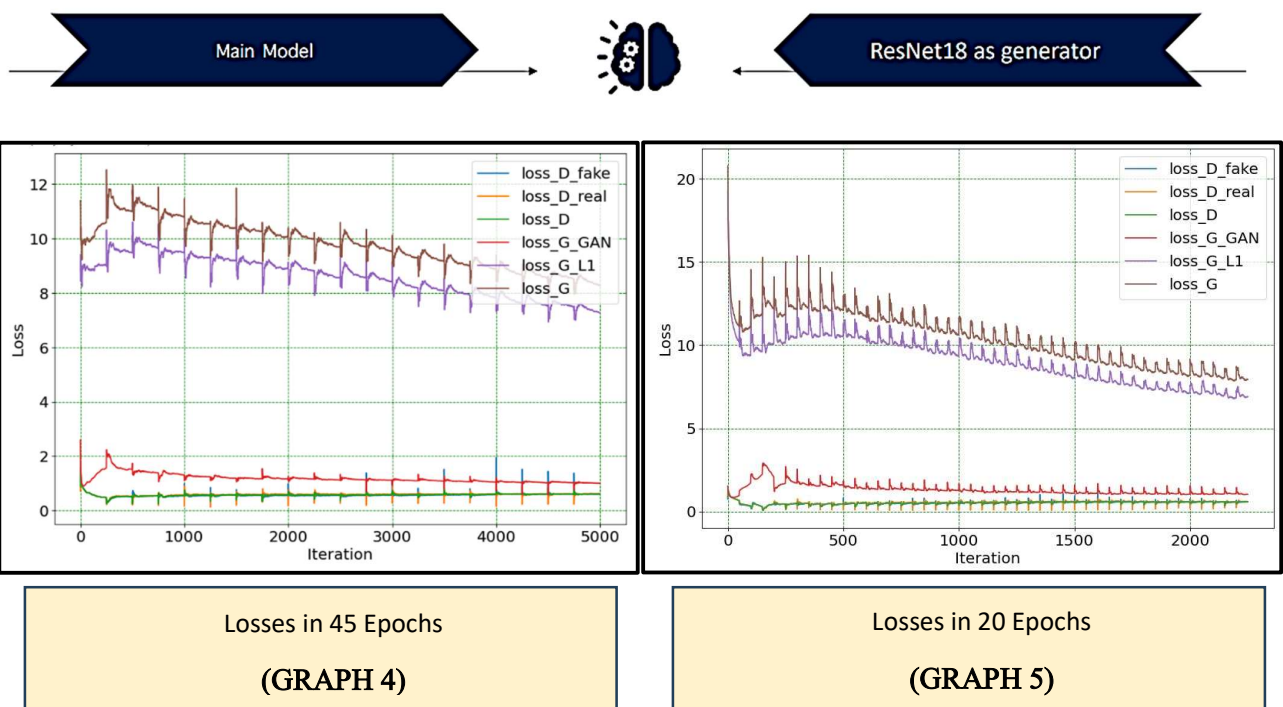
- We removed the last two layers: -
GlobalAveragePooling and a Fully Connected Layer
- Used this backbone to build a U-Net as the generator: -
Two output channels and 1 input size of 256x256
- Trained the generator only with L1 loss in 20 epochs (not depends on Discriminator)
- Did the same training process with new generator.

ResNet Architecture



6.3.2 Comparison with main model

We used a pre-trained ResNet18 as the backbone of our U-Net. It quickly learns the mapping between grayscale images and color images. As a result, using a pre-trained model as the backbone of the generator's down-sampling path in a cGAN for colorization can be a useful approach. In fact, we use a pre-trained ResNet18 as the backbone of the U-Net and to accomplish the second stage of pretraining, we train the U-Net on our training set with only L1 Loss in 20 epochs. Then we move to the combined adversarial and L1 loss, and trained it for 20 epochs which we can see the losses. By comparing GRAPH 4 and GRAPH 5 we can get that the pre-trained model has better performance because in fewer number iterations gets less loss.



Chapter 7

Conclusion

This project was centred on developing a sophisticated deep learning approach specifically customised for the automated recolorization of grayscale images transformed from the input data by Lightness in Lab color space. To achieve this, a hybrid architecture combining a cGAN (conditional Generative Adversarial Network) with a U-Net generator was employed. The strength of this approach lies in the combination of these two frameworks.

The cGAN's adversarial training mechanism enables the network to learn the delicate details of colorization by discriminating between real and generated images. Meanwhile, the U-Net generator's unique architecture, with its encoding-decoding structure, excels in capturing both high-level features and fine-grained details within images.

The U-Net's ability to effectively comprehend and reconstruct high-level features alongside minute image details makes it a reliable choice for image-to-image translation tasks such as colorization. However, the success of this methodology is dependent upon two critical factors: -

1. **Quality of Training Data**
2. **Network Architecture Design**

These two aspects, the quality of training data and the complexity of the network design, all together influence the success and proficiency of the recolorization process. Achieving exceptional results in this context demands meticulous attention to dataset selection and the thoughtful crafting of the network's architecture.

In a nutshell,

- cGAN shows promising performance, delivering reasonable results even with small datasets and limited resources.
- Implementing a pretrained model as the generator in cGAN enhances performance, leading to improved outcomes.
- With bigger dataset and a greater number of training epochs the results will be more realistic. (which needs more recourses)

Bibliography

- https://www.researchgate.net/publication/360254530_Two_Decades_of_Colorization_and_Decolorization_for_Images_and_Videos
- <https://subscription.packtpub.com/book/data/9781789537147/1/ch01lv1sec03/transforming-color-space--rgb---lab#:~:text=The%20brightness%20of%20the%20input,to%20touch%20the%20color%20channels>
- <https://deeplai.org/machine-learning-glossary-and-terms/per-pixel-loss-function>
- <https://towardsdatascience.com/colorizing-black-white-images-with-u-net-and-conditional-gan-a-tutorial-81b2df111cd8>
- <https://theailearner.com/tag/patchgan/>
- <https://machinelearningmastery.com/how-to-implement-pix2pix-gan-models-from-scratch-with-keras/>
- <https://www.analyticsvidhya.com/blog/2022/01/imagine-your-world-with-generative-adversarial-networks/>
