RECOMMENDATION SYSTEM: RECOMMENDING NATURAL COLOURS FOR BLACK AND WHITE IMAGES

Major project report submitted in partial fulfilment of the requirement

for the degree of Bachelor of Technology

In

Computer Science and Engineering

By

AKSHAT **(191238)**

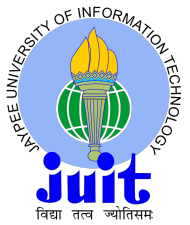
SHIVAM SHARMA **(191241)**

UNDER THE SUPERVISION OF

DR. SHWETA PANDIT (Supervisor)

&

DR. JAGPREET SIDHU (Co-Supervisor)



Department of Computer Science & Engineering and

Information Technology

Jaypee University of Information Technology, Waknaghat,

173234, Himachal Pradesh, INDIA

CERTIFICATE

Candidate Declaration

I hereby declare that the work presented in this report entitled **"RECOMMENDATION SYSTEM: RECOMMENDING NATURAL COLOURS FOR BLACK AND WHITE IMAGES"** in partial fulfilment of the requirements for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering** submitted in the department of Computer Science & Engineering and Information Technology, Jaypee University of Information Technology Waknaghat is an authentic record of my own work carried out over a period from July 2022 to May 2023 under the supervision of **Dr. Shweta Pandit (Assistant Professor SG) Department of Electronics and Communication Engineering & Dr. Jagpreet Sidhu (Assistant Professor SG) Department of Computer Science and Engineering**.

I also authenticate that I have carried out the above mentioned project work under the proficiency stream **"Data Science"**.

The matter embodied in the report has not been submitted for the award of any other degree or diploma.

Akshat (191238)
Shivam Sharma (191241)

This is to certify that the above statement made by the candidate is true to the best of my knowledge.

Supervisor	**Co-Supervisor**
Dr. Shweta Pandit	**Dr. Jagpreet Sidhu**
Assistant Professor (SG)	Assistant Professor (SG)
Department of ECE	Department of CSE

ACKNOWLEDGEMENT

Firstly, We express our heartiest thanks and gratefulness to almighty God for His divine blessing makes it possible for us to complete the project work successfully.

We are really grateful and wish my profound indebtedness to Supervisor **Dr. Shweta Pandit Assistant Professor SG (Department of ECE) & Dr. Jagpreet Sidhu Assistant Professor SG (Department of CSE)** Jaypee University of Information Technology, Waknaghat. Deep Knowledge & keen interest of my supervisors in the field of "**Data Science**" to carry out this project. Their endless patience, scholarly guidance, continual encouragement, constant and energetic supervision, constructive criticism, valuable advice, reading many inferior drafts and correcting them at all stages have made it possible to complete this project.

We would like to express our heartiest gratitude to **Dr. Shweta Pandit Assistant Professor SG (Department of ECE) & Dr. Jagpreet Sidhu Assistant Professor SG (Department of CSE)**, for their kind help to finish my project.

We would also generously welcome each one of those individuals who have helped us straightforwardly or in a roundabout way in making this project a win. In this unique situation, We might want to thank the various staff individuals, both educating and non-instructing, which have developed their convenient help and facilitated my undertaking.

Finally, we must acknowledge with due respect the constant support and patients of our parents.

Akshat (191238)

Shivam Sharma (191241)

TABLE OF CONTENT

LIST OF ABBREVIATIONS

Abbreviations	Full Form
CNN	Convolutional Neural Network
GAN	Generative Adversarial Network
cGAN	Conditional Generative Adversarial Network
LAB	L*: Lightness A*: Red/Green B*: Blue/Yellow
RGB	R*:Red G*:Green B*:Blue
SURF	Speeded Up Robust Features
PSNR	Peak Signal-to-Noise Ratio

LIST OF FIGURES

LIST OF GRAPHS

ABSTRACT

In the past ten years, the idea of automatic image colorization has attracted attention for a range of uses, including the restoration of old or damaged photos. This problem is extremely poorly presented since assigning colour information involves such a wide range of degrees of freedom. Recent developments in automatic colorization frequently use input images that share a common theme or data that has undergone extensive processing, like semantic maps. Using conditional adversarial networks, we attempt to fully broaden the colorization process and address image colorization issues. Landscape colour and grayscale images from the publicly accessible Kaggle dataset were used to train the network.

In this study, we perform a colorization task using the L*a*b* colour space. This differs from the widely used RGB colour space because the L*a*b* colour space features a separate channel for displaying image brightness while the remaining two channels, a* and b*, are used to fully represent the four distinct colours of human vision: red, green, blue, and yellow.

The network was trained using the generator and discriminator parts of the GAN network design. The discriminator assesses the effectiveness of the colorizations and gives feedback to the generator to help it perform better while the generator creates coloured images. The network can develop the ability to create more precise and aesthetically pleasing colorizations by repeatedly training the generator and discriminator.

In general, automatic image colorization has the potential to revolutionise the way we enhance and preserve historical photographs. It can also be a useful tool for designers and artists who need to produce coloured images rapidly. Additional study in this field is anticipated to produce even more remarkable findings and applications.

Chapter 01:
INTRODUCTION

1.1 Introduction

The "translation" of an input image into a corresponding output image can be used to describe a variety of image processing, computer graphics, and computer vision problems. The process of turning a grayscale image into a colourful RGB image is known as image colorization.

Convolutional Neural Networks (CNNs) have been utilised in the past to address a variety of applications involving image processing, and the community has already made major advancements in this field. Finding loss functions that motivate the CNN to do what we actually want it to is difficult in the picture colorization process because of the enormous amount of variability available when assigning colours to images.

Here, we use conditional GANs from the family of generative adversarial networks to try and convert a grayscale input image into a colour image. This design is composed of two smaller networks called Generator and Discriminator. The generator aims to generate outputs that are identical to real data, as suggested by its name. Identifying whether a sample originated from the generator's model distribution or the original data distribution is the task of the discriminator. Up until the generator can consistently produce data that the discriminator cannot classify, both subnetworks are trained concurrently. At this stage, the generator can provide a generalised mapping from a grayscale image to an RGB colour space image, which is known as Nash Equilibrium.

1.2 Objective

The problem of image colorization is to convert a high-dimensional input into a high-dimensional output. The input structure and the output structure are very closely linked in this pixel-wise regression problem. This implies that each pixel in the grayscale input image must get colour information in addition to having an output with the same spatial dimension as the input.

Here we try to used L*a*b* instead of RGB colour space this would reduce the output mapping from three colour spaces to only two which might help to reduce model parameters which makes training faster, efficient as well as effective minimisation of cost function.

We are trying to construct the GAN network architecture, which consists of a generator and a discriminator. The output mapping must meet the requirement that the generator would produce a colourful image using only the input latent noise and grayscale image as input data. By penalising picture structure on a patch-level scale, the discriminator architecture we are using is dubbed PatchGAN. This discriminator's job is to determine the truthfulness of each NxN patch in a picture. This makes it easier to create precise, accurate images.

In general, to accomplish successful and efficient training, our method for image colorization makes use of the advantages of the Lab* colour space and GAN network architecture, producing precise and accurate images.

1.3 Motivation

Early in the new millennium, grayscale colorization models initially emerged. In 2002, Welsh et al. unveiled a texture synthesis-based colorization method. By comparing brightness and texture data from an existing colour image to the grayscale image to be coloured, the colourized grayscale images were created.

The colorization problem was given a fresh formulation by Levin et al. in 2004. An inverse technique was used to build the cost function, penalising pixels that differed from a weighted average of their neighbouring pixels. Both of these suggested options required a substantial amount of user input, which rendered the alternatives less than ideal.

A colorization method was proposed in light of the comparison of the differences in colorization between convolutional neural networks and GAN. The models in the study learn a loss function in addition to the mapping from input to output picture.

Our main objective is to create a custom conditional GAN network model i.e. easy to train, optimise, save & light weight to deploy. We would also create an optimised dataflow pipeline and use various techniques such as spectral normalisation, label smoothing etc. Finally, we would also compare the results of different training techniques and hyperparameters such as batching, learning rate etc.

1.4 Language Used

- Python
- HTML5
- CSS3
- JavaScript

1.5 Libraries Used:

- Numpy
- Pandas
- Matplotlib
- Tensorflow
- Kaggle
- Tensorflow_io
- Seaborn
- Tensorboard
- ReactJS
- FastAPI
- Nginx
- Netlify
- Pickle

1.6 Deliverables/Outcomes

Through this project, we hope to develop a custom conditional GAN network model that is simple to train, optimised, easy to save, and deploy.

A GAN is made up of two smaller networks called the generator and discriminator. Multilayer perceptron architecture is used by both the generator and discriminator. Convolutional neural networks (CNNs) are used for both the generator and discriminator since colorization is a form of image translation problem.

Additionally, we would present data on the effects of different hyperparameter tuning on image colorization and develop an optimised preprocessing and flow pipeline. We offer a self-contained model that can be used with Tensorflow serving, Tensorflow.js, or even just Tensorflow to be hosted on any cloud backend.

Additionally, we intend to compare the effectiveness of our model to other cutting-edge image colorization techniques, including those that are based on deep learning and conventional image processing methods. This would allow us to prove our proposed model's superiority and offer a comparative examination of its capabilities.

Chapter 02:
LITERATURE SURVEY

2.1 Welsh et al. 2002

By comparing brightness and texture data from an existing colour image to the grayscale image to be coloured, they developed a texture synthesis-based colorization process that coloured grayscale images.

The majority of exemplar-based colorization techniques make the assumption that reference colour pixels with similar neighbourhood or intensities should be used to assign the colour to the target grayscale pixels. Welsh et al. suggested a colorization method that merely relies on transferring colour in accordance with the brightness values that are obtained by averaging nearby pixels in the target image and those in the reference image.

Even when the correlation is with the same brightness value and equal neighbourhood statistical variables, mismatching may occur in a different location of the colour image. As a result, some methods consider each pixel's higher-level content in addition to brightness. For instance, Irony et al. effectively used textural data and a supervised classifier to colour a classification image.

In order to identify matching pixels between grayscale and colour images, Gupta et al. devised a technique that makes use of a cascade feature matching methodology. Simply said, the major goal of these exemplar-based approaches is to accurately transfer colour by identifying the reference image's and target image's best-matched pixels (areas).

2.2 Levin et al. 2004

They proposed a fresh approach to the colorization issue. An inverse technique was used to build the cost function, penalising pixels that differed from a weighted average of their neighbouring pixels. Both of these suggested options required a substantial amount of user input, which rendered the alternatives less than ideal.

However, colorization offers a considerable barrier due to the fact that it is an expensive and time-consuming process. For instance, while colouring a still image, an artist may frequently begin by separating the image into sections before assigning each section a different colour. Unfortunately, automatic segmentation algorithms frequently identify fuzzy or complicated region boundaries inaccurately, such as the line dividing a subject's hair from her face.

Therefore, the task of manually designing intricate borders between sections is typically left to the artist. Additionally, tracking of areas across multiple shot frames is necessary for colorization. Current tracking algorithms usually require a lot of user input because they can't reliably track non-rigid regions.

A neural network can be trained on a large sample of coloured images to segment images and track non-rigid regions automatically. This approach would considerably reduce the quantity of user input required and speed up the colorization process. The drawback is that in order to adequately train the neural network, a large amount of data would be required, which is usually difficult to obtain in some sectors, such as historical photos or priceless artworks. Despite these challenges, using a deep learning approach to colourize presents exciting possibilities for additional research in the field.

2.3 Phillip Isola et al. 2018

In recent years, conditional adversarial networks have gained popularity as a method for resolving image-to-image translation issues. These networks learn a loss function that can be used to train the mapping in addition to learning to map input images to output images. With this method, a variety of activities can be handled using a single generic strategy, doing away with the requirement for distinct loss formulas for various tasks.

This method's capacity to resolve issues like colouring photographs, reconstructing objects from edge maps, and synthesising photos from label maps, which were previously challenging or impossible to handle, is one of its key benefits. The extensive usage of our technology by artists and other internet users, who have submitted their own testing with our technology and further illustrated its adaptability and simplicity, has also been facilitated by the publication of the pix2pix programme linked with this study.

Due to conditional adversarial networks' effectiveness, it is possible that hand engineering of mapping functions won't be necessary in the future, and that adequate results could be reached with less parameter fiddling. Therefore, by making a variety of tasks easier and more accessible for both researchers and practitioners, this technology has the potential to fundamentally alter how we approach image-to-image translation challenges.

Additionally, conditional adversarial networks have demonstrated excellent results by producing stunning images despite the lack of sufficient training data. This is due to the fact that neural networks can generate precise output images after learning from a sparse set of paired training data. As a result, problems with image-to-image translation can be resolved in a variety of situations, including those where it is difficult or expensive to collect substantial amounts of annotated data.

2.4 Hong-an Li et al. 2022

Image colour rendering is currently receiving a lot of attention as a significant area of image processing. Image colour rendering based on neural networks has gradually grown in popularity as deep learning has progressed. due to the manual nature of traditional colour rendering techniques and their strict requirements for reference photos. Furthermore, the colour rendering effect is not perfect when the image's structure and colour are complicated.

Deep learning-based colour rendering techniques 87 can be quickly implemented in a real-world production setting, eliminating the drawbacks of the older techniques. The image can be automatically produced in accordance with the model utilising the neural network model and the associated dataset training model, free from the influence of humans or other variables.

In order to achieve colour rendering, Larsson employed the super-column model to break down the colour and saturation of the image and the convolutional neural network to take into account the brightness of the image as input. Iizuka used the fusion layer in the convolutional neural network to combine the low-dimensional feature and global feature of the image, generating the colour of the image and processing images of any resolution.

To manage the multi-mode uncertainty in colour rendering and retain the colour diversity, Zhang created an appropriate loss function. However, up-sampling is used to make the image size consistent when the grayscale image characteristics are retrieved using the aforementioned method, which results in the loss of image information.

Additionally, the rendering effect is limited due to the network structure's inability to accurately extract and comprehend the complex features of the image. To achieve the image transformation, Isola enhanced conditional generative adversarial networks.

By learning the mapping relationship between a grayscale image and a colour image, for instance, the suggested pix2pix model can achieve conversion between different images. However, the training instability of generative adversarial networks (GAN) based on the pix2pix model is a drawback. Additionally, producing robust images is a weakness of the present deep learning-based image rendering techniques.

Gabor filters are effective in extracting texture information from images at all scales and directions, and they can somewhat mitigate the effects of noise and light fading. As a result, we suggest an improved pix2pix colour rendering technique based on the Gabor filter for robust images.

These are this paper's primary contributions:

1) The enhanced pix2pix model not only achieves good visual effects and image rendering on autopilot, but also more stable training and improved image quality.
2) The Gabor filter was included to improve the stability of the images produced by the model.
3) Experimental metrics demonstrate that the proposed method performs better for a robust image.

Chapter 03:
SYSTEM DESIGN & DEVELOPMENT

3.1 Problem Definition

The conversion of an input image into a corresponding output image is a typical problem in image processing, graphics, and vision. Despite the fact that the challenge of converting pixels to pixels is typically the same, these challenges are generally overcome by means that are application-specific.

Conditional adversarial nets are a general strategy that appear to be successful for a variety of these problems. The restoration of old or damaged photos is one application for which automatic image colorization has drawn interest. This problem is very poorly posed because assigning colour information involves such high degrees of freedom. Recent advances in automatic colorization use complex input arguments, such as semantic maps..

Convolutional neural networks (CNNs) are currently the industry standard workhorse powering a variety of image prediction applications because of significant advancements made by the community in this area. Despite CNNs learning to minimise a loss function, which evaluates the output's quality, producing good losses still requires a lot of manual labour.

The findings will be biassed if we employ a simple strategy and instruct the CNN to minimise the Euclidean distance between predicted and ground truth pixels. This is because blurring, which reduces Euclidean distance by averaging all realistic outputs, is the cause of this. For instance, identifying the loss functions that cause the CNN to produce clear, realistic images is still a work in progress and frequently necessitates specialised knowledge. Instead, it would be ideal if we could simply define a general target and then have a loss function that is appropriate for achieving this goal generated automatically.

In a conventional GAN, noise data generated at random serves as the generator's input. This method, however, cannot be used to solve the automatic colorization problem because the inputs to our issue are grayscale images rather than noise. This problem was resolved using conditional generative adversarial networks, a subset of GAN. cGANs are capable of processing conditional data (in this case, a grayscale image), which they use to perform mapping and cost function minimization.

3.2 Our Solution

The computer vision community has been interested in the problem of image colorization for a while now. Despite the recent major improvements, there is always potential for growth, especially when it comes to totally automating the process. In order to solve this problem, we'll build a dataflow pipeline in this project that combines image preprocessing stages with a cGAN architecture and trains it on a publicly available dataset. Additionally, we will look at different picture representation formats and contrast how well they work in the context of colourizing images.

We will look into the picture preprocessing techniques required for our task first. Because it significantly affects the accuracy and calibre of the results, image preprocessing is a crucial step in the pipeline for colourizing images. The preparation techniques we may consider include scaling the input images to a standard size, carrying out colour space conversions, and employing data augmentation techniques to increase our training dataset.

Following the preprocessing of the images, we can begin building the dataflow pipeline. We generate colourized images for the training and evaluation stages of the pipeline using a cGAN architecture. A cGAN is a type of generative model that uses adversarial training to generate images that are comparable to the training data.

When building our dataflow pipeline, we must consider a number of hyperparameters because they could have an impact on how well our model works. The hyperparameters that we might consider include the learning rate, batch size, and number of epochs. We can also experiment with other convolutional layers or residual blocks, for instance, to see which model design best achieves our aim.

In our project, the choice of image representation format is essential. Despite being the most popular format for displaying coloured images, RGB might not always be the best choice for our objective. One option that we might consider is the LAB colour space, which divides an image's luminance (brightness) and chrominance (colour) components. By separating the brightness information from the colour information, we might be able to improve the accuracy of our colorization model.

In conclusion, the objective of this research is to fully automate the image colorization process by creating a dataflow pipeline that combines image pretreatment techniques, a cGAN architecture, and training it on a dataset that is openly accessible. To determine which model architecture and hyperparameters are ideal for our purpose, we will explore other options. We will also compare the performance of other image representation formats, including RGB and LAB, to determine which one best serves our goal. With the aid of this project, we want to make significant progress towards fully automating the image colorization process and improving the accuracy and quality of the output.

3.3 Dataset used:

Landscape colour & grayscale images dataset

3.4 Dataset Features

3.4.1 Type of Dataset

This Dataset contains 7129 colourful RGB images & 7129 grayscale images of landscapes in jpg image format.

3.4.2 Description of the dataset

Dataset Description: This dataset consists of streets, buildings, mountains, glaciers, trees etc and their corresponding grayscale image in two different folders. The main objective of creating this dataset is to create a neural network that can colourized grayscale landscape images.

Total no of files: 14,300 (14.3k) images including both colourful and grayscale.

3.5 Algorithm / Pseudo code of the Project Problem

A conditional GAN's objective could be described as

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

where *G* seeks to reduce the goal and *D* tries to maximise it, i.e.
G = arg minG maxD LcGAN (G, D).*

We contrast the unconditional form with a discriminator that does not observe x:

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

Previous research has shown that it is advantageous to pair the GAN target with a more traditional loss, like L2 distance. The discriminator's job is unaffected, but the generator needs to mislead the discriminator and be relatively similar to the ground truth output in an L2 sense in order to be effective. We also take into account using L1 distance because L1 encourages less blurring.

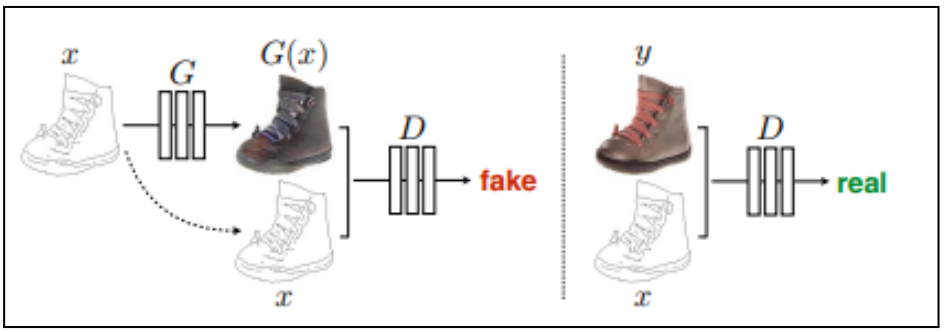
$$\mathcal{L}_{L1}(G) = \mathbb{E}_{x,y,z}[\|y - G(x, z)\|_1].$$

Our ultimate goal is

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

Generic artificial neural networks (GANs) are generative models that identify a mapping, G: z y, from an input picture y to a random noise vector z. A mapping from the observed picture x and the random noise vector z to y, denoted by G: (x, z) y, is learned using conditional GANs in contrast.

An adversarially trained discriminator named "D" is instructed to be as skilled as possible at identifying the generator's "fakes". G has been trained to produce images that are indistinguishable from "real" photographs. In Figure, this training approach is displayed.



Fig(3.1) Methodology

3.6 Network Architecture

3.6.1 Generator U-Net:

The image colorization problem has seen widespread application of encoder-decoder networks. They are created by downsampling the data gradually through a sequence of layers, starting with the input image, until it reaches the bottleneck layer. The input image's most abstract representation can be found in the bottleneck layer. The image is then recreated by running the bottleneck representation through a number of upsample layers in reversal of the previous step.

Each piece of data must pass through each tier of this technique, including the bottleneck layer, which is a disadvantage. Low-level data required for the colorization of images may be lost as a result of this. Since a lot of low-level information is exchanged between the input and output in many picture translation issues, it is preferable to transmit this information directly over the network.

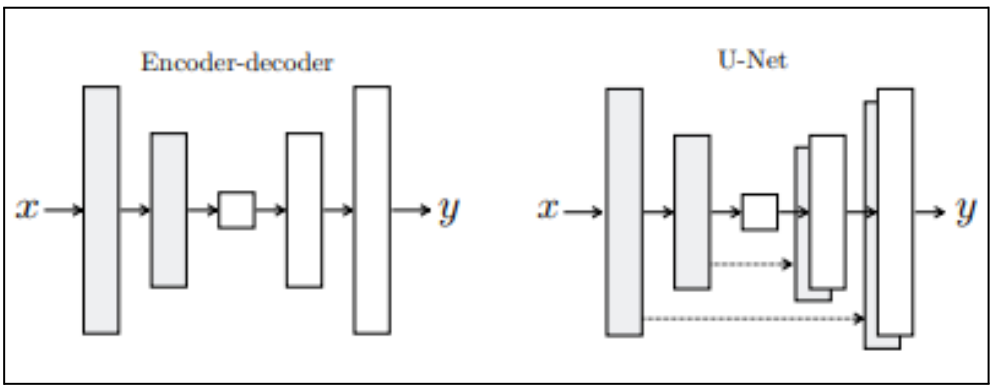
To address this issue, skip links are added to the encoder-decoder network. The U-shaped shape of the skip links is where the term "U-Net" originates. The deep neural network architecture known as the U-Net generator combines the encoder-decoder network and skip connections. The skip allows the generator to quickly transfer low-level information from the input image to the output image.

The two main parts of the U-Net generator are the encoder and decoder. The encoder and encoder-decoder networks both use a series of convolutional layers to gradually downsample the data in the encoder. Although the decoder component also has a number of upsampling layers, unlike the encoder-decoder network, the size of the data is gradually increased. The skip connections are added between each layer i and layer n-i, where n is the total number of layers. All layers i and n-i channels are concatenated together in a skip connection.

By integrating skip connections, the U-Net generator can more effectively capture the low-level information included in the input image. The network is able to transfer this data without passing it through any layers—including the bottleneck layer—directly to the output image through the use of skip connections. The final image is hence more accurate and finely detailed.

The U-Net generator has been successfully used to colourize images in addition to other image translation tasks. The output image's accuracy and quality have shown a substantial improvement over earlier encoder-decoder network topologies. Because it has fewer parameters than other generative models, the U-Net generator is also quicker and more efficient.

In conclusion, the deep neural network architecture known as the U-Net generator combines the encoder-decoder network and skip connections. The skip connections enable the network to transmit low-level data directly from the input image to the output image. The U-Net generator has proven to be a significant advance over previous encoder-decoder network topologies in terms of output image quality and accuracy. When compared to other generative models, it has a very small number of parameters, which makes it quicker and more efficient. It has been successfully used for a variety of picture translation tasks, including image colorization.



Fig(3.2) Generator Methodology

3.6.2 Discriminator PatchGAN:

The problem of obtaining high-frequency information has been resolved using a discriminator architecture known as PatchGAN. Because it only penalises structure at the patch size, the PatchGAN discriminator can determine the validity of each of the N N patches in a picture. The total number of responses is then used to calculate the PatchGAN discriminator's final output.

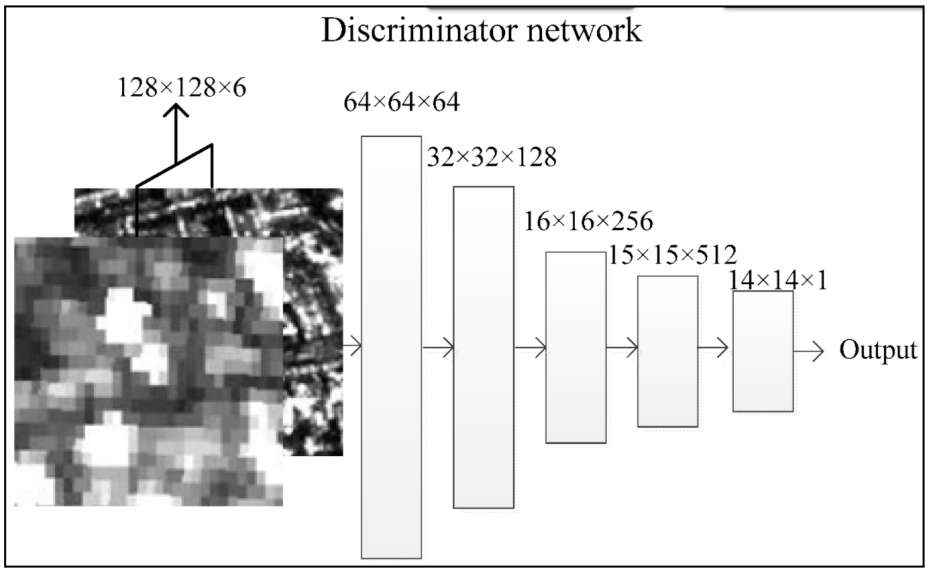
The PatchGAN discriminator provides a number of advantages over traditional discriminators used in image colorization problems. It has the advantage of being efficient computationally. Because it only penalises structure at the patch size, the PatchGAN discriminator uses significantly less compute than traditional discriminators.

Another advantage of the PatchGAN discriminator is that it is better at detecting high-frequency characteristics. Traditional discriminators frequently fail to collect high-frequency information, which reduces the clarity of the image. The local picture patch organisation, on the other hand, is the main focus of the PatchGAN discriminator, allowing it to better capture high-frequency information.

Images have been successfully coloured using the PatchGAN discriminator, among other image translation tasks. It has proven to have a significant improvement in output image quality and accuracy over conventional discriminators. Due to the PatchGAN discriminator's relatively low parameter count when compared to other discriminators, it is also quicker and more efficient.

In conclusion, the PatchGAN discriminator is an architecture designed to punish structure at the patch size. This makes it possible for it to more

precisely gather high-frequency data and assess the reliability of each of the N × N patches in a picture. The PatchGAN discriminator is computationally efficient and better at capturing high-frequency information than traditional discriminators used to address image colorization problems. It has shown a notable improvement over traditional discriminators in terms of the output image's quality and accuracy, and it has been effectively applied to a number of image translation applications, including image colorization.



Fig(3.3) Discriminator Methodology

3.7 Flow graph of the Major Project Problem

The fundamental problem with the project at hand is image colorization, and using a flow diagram, we can understand how it functions. Both the raw grayscale image and the desired output image in its original colours are shown at the beginning of the process. The generator then takes the input image and transforms it into the desired output image while maintaining the original image by using latent information from the image.

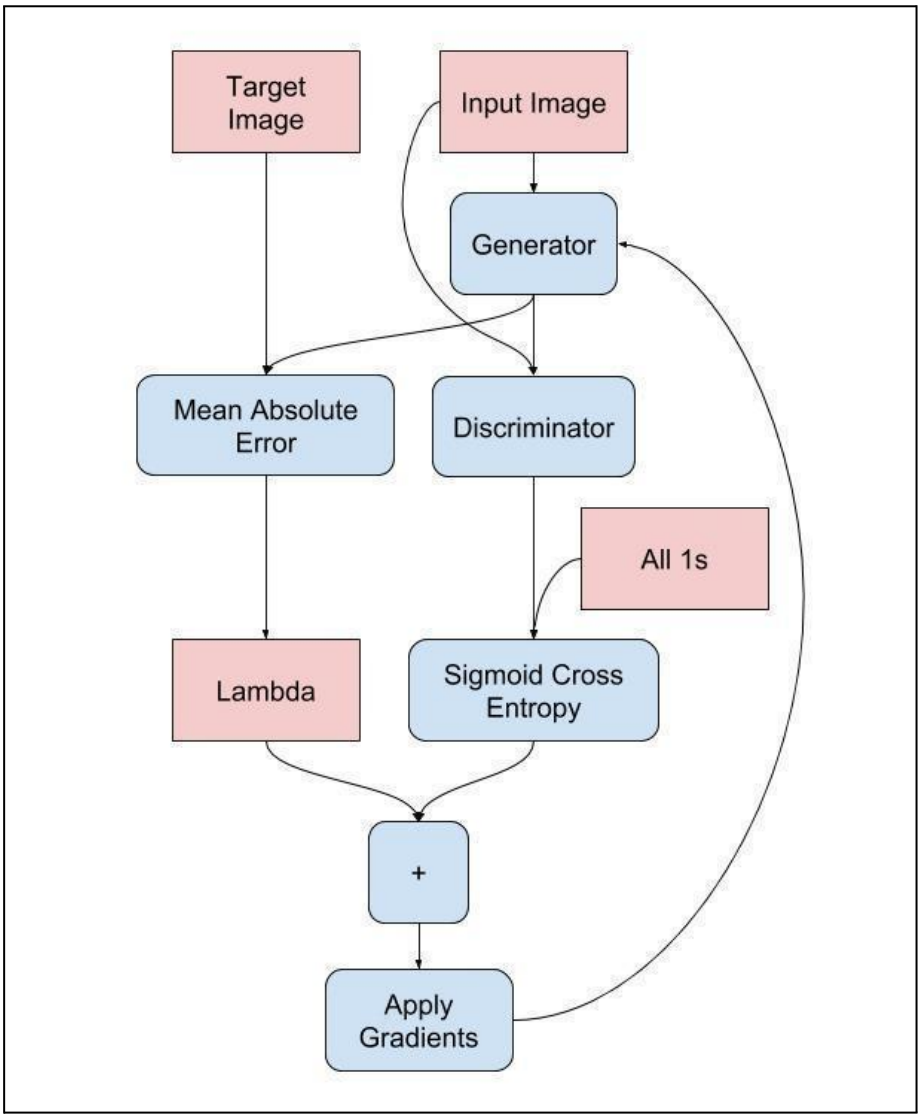
Utilising an absolute mean error calculator to determine the mean absolute error between the two photos is the next step in the process. This error, which is measured in pixels, reveals how accurately and realistically the coloured images produced by the generator are.

A predicted image from the generator is given to the discriminator, who decides if it is authentic or fake. The discriminator computes the output for each image and compares it to a threshold value to determine if the image is authentic or fake. The generator can alter its workflow and create fake images less frequently in the future if the discriminator determines that the image is a fake and notifies the generator of this finding.

The final output error is calculated by adding the outputs of the sigmoid entropy calculator, the absolute error, and the sigmoid entropy. Given the discriminant output, the sigmoid entropy calculator's results reveal that the discriminator shouldn't be able to distinguish between real and fake photos.

Overall, the workflow of this image colorization project is to improve the accuracy and realism of the output of the generator. The generator takes the input image and transforms it into an accurate and realistic coloured image while keeping the original grayscale image. The discriminator evaluates the generator's output and provides feedback to help it streamline its processes and produce future images that are more accurate and lifelike.

This image colorization project can also be applied in the real world to increase the visual appeal of product images, improve medical imaging, and repair old photographs. By adding colour, black-and-white images can come to life, becoming more appealing and accessible to modern viewers. For example, by adding colour to old photographs, we would be better able to imagine the past and feel more familiar with and aware about our cultural heritage. Similar to this, colouring medical images can improve patient outcomes by assisting professionals in identifying and diagnosing a range of health problems. As a result, this project's process has the potential to have a substantial impact on various industries, making it a valuable tool for many professions.



Fig(3.4) Flow Chart

3.8 Screenshots of the project's various stages

3.8.1 Preprocessing functions for images:

1) **Load image:** Reads and decodes a jpeg file from a given path as tensor variable.

```
@tf.function()
def load_img(target_img_path):
  target_img = tf.io.read_file(target_img_path)
  target_img = tf.io.decode_jpeg(target_img, channels=3)
  target_img = tf.image.resize(target_img, (IMG_WIDTH, IMG_HEIGHT))
  target_img = tf.cast(target_img, tf.float32)

  return target_img
```

2) **Resize:** Resizes the given image tensor according to the provided height and width.

```
@tf.function()
def resize(target_img, height, width):
  target_img = tf.image.resize(
      target_img, (width, height),
      method=tf.image.ResizeMethod.NEAREST_NEIGHBOR
  )

  return target_img
```

Fig(3.5) Image Resizing

3) **Jitter:** Adds random cropping and rotation to given image tensor.

```
@tf.function()
def random_crop_rotate(target_img, width, height):
  cropped_img = tf.image.random_crop(target_img, (width, height, 3))
  target_img = tf.image.random_flip_left_right(cropped_img)
  return target_img
```

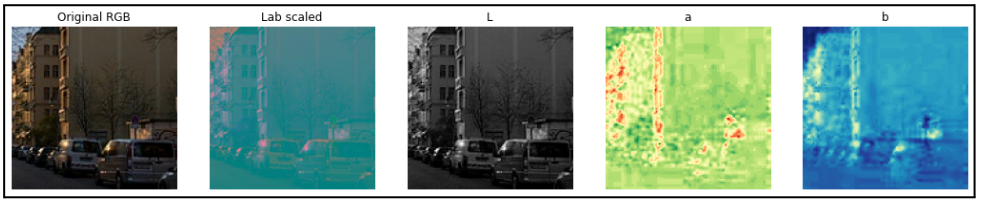
Fig(3.6) Adding Random Jitter

4) **RGB to LAB:** Converts an RGB image (tensor) into LAB colorspace image for both visualisation and neural network.

```
@tf.function()
def rbg2lab(target_img, isnormalized = True, normalize_lab = False, comp = ''):
  if not isnormalized:
    target_img = target_img/255.0

  # Takes RGB Image in Normalized Form
  target_img = tfio.experimental.color.rgb_to_lab(target_img)
  if normalize_lab:
    tf.Assert(tf.reduce_any(tf.equal(comp, ['vis', 'net'])), data=[comp], name='Lab_Normalization_Error')
    if comp == 'vis':
      target_img = (target_img + [0, 128, 128]) / [100., 255., 255.] # 0-1 range
    else:
      target_img = target_img / [50., 127.5, 127.5] + [-1, 0., 0.]   # -1 to 1 range
  return target_img
```

5) **LAB to RGB:** Converts an LAB colorspace image into RGB image (tensor) for both visualisation and neural network.

```
@tf.function()
def lab2rgb(lab_img, isnormalized = False, comp = ''):
  if isnormalized:
    tf.Assert(tf.reduce_any(tf.equal(comp, ['vis', 'net'])), data=[comp], name='Lab_Normalization_Error')
    if comp == 'vis':
      lab_img = lab_img * [100.,255., 255.] + [0, -128, -128]; # from 0-1 range
    else:
      lab_img = (lab_img + [1.,0., 0.]) * [50., 127.5, 127.5]; # from -1 to 1 range

  # Take LAB Image in Unnormalized Form.
  rgb = tfio.experimental.color.lab_to_rgb(lab_img)
  return rgb
```



Fig(3.7) LAB to RGB Conversion

3.8.2 Data Pipelining:

1) **Splitting Dataset:** Splits dataset based on given train, evaluation and test sizes.

```
train_dataset_files = dataset_files.skip(test_size + eval_size)
test_dataset_files = dataset_files.take(test_size)
eval_dataset_files = dataset_files.take(eval_size)
```

2) **Interleaving Data:** Maps *map_func* across this dataset, and interleaves the results.

```
train_dataset = train_dataset_files.interleave(
    lambda tar_img : tf.data.Dataset.from_tensors(load_train_img(tar_img, comp = 'net')),
    num_parallel_calls=tf.data.AUTOTUNE, deterministic=False
)
eval_dataset = eval_dataset_files.interleave(
    lambda tar_img : tf.data.Dataset.from_tensors(load_train_img(tar_img, comp = 'net')),
    num_parallel_calls=tf.data.AUTOTUNE, deterministic=False
)
test_dataset = test_dataset_files.interleave(
    lambda tar_img : tf.data.Dataset.from_tensors(load_test_img(tar_img, comp = 'net')),
    num_parallel_calls=tf.data.AUTOTUNE
)
```

3) **Optimising Dataflow:** Using *tf.data* API to build a highly efficient and fast tensorflow input pipeline.

```
dataset_files = dataset_files.shuffle(BUFFER_SIZE, reshuffle_each_iteration=False)
```

```
train_dataset = train_dataset.cache().batch(BATCH).prefetch(tf.data.AUTOTUNE)
test_dataset = test_dataset.cache().batch(BATCH).prefetch(tf.data.AUTOTUNE)
eval_dataset = eval_dataset.cache().batch(BATCH).prefetch(tf.data.AUTOTUNE)
```

3.8.3 Model:

1) **Base Layer:**

Down Sampling Layer:

Data input is mapped into a low dimensional latent representation using this layer.

```
def downsample(filters, kernel_size, batch_normalization = True, activation = True):
  pipeline = tf.keras.Sequential()

  pipeline.add(tfa.layers.SpectralNormalization(
      tf.keras.layers.Conv2D(
        filters=filters, kernel_size=kernel_size,
        strides=2, padding='same',
        use_bias = False
      ), 10
  ))

  if batch_normalization:
    pipeline.add(tf.keras.layers.BatchNormalization())

  if activation:
    pipeline.add(tf.keras.layers.LeakyReLU())

  return pipeline
```

Fig(3.8) Down Sampling

Up Sampling Layer:

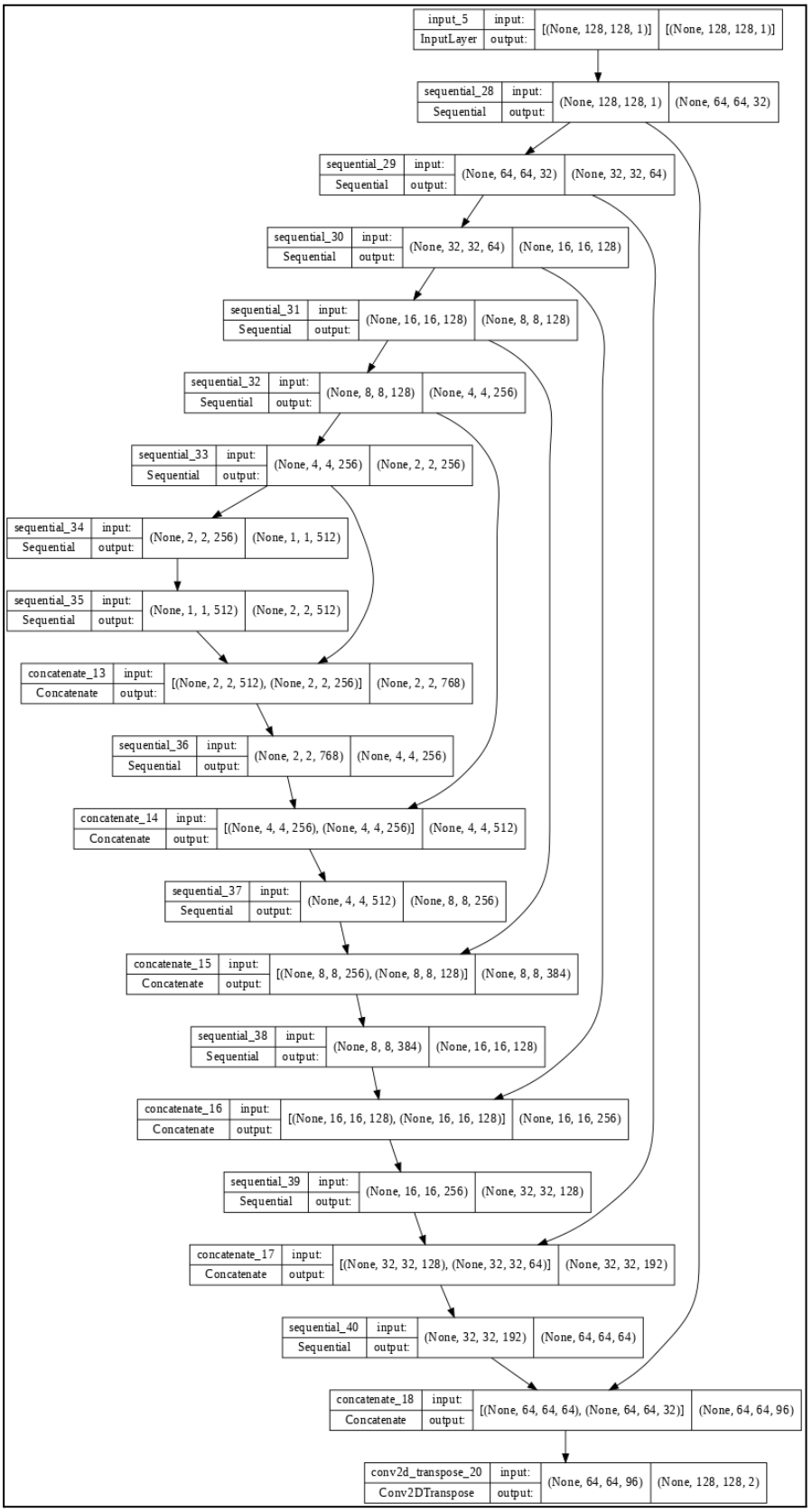
This layer serves as a mapping between input data and the desired output in a low-dimensional latent form.

```
def upsample(filters, kernel_size, dropout=False, activation = True):
  pipeline = tf.keras.Sequential()

  pipeline.add(tfa.layers.SpectralNormalization(
      tf.keras.layers.Conv2DTranspose(
          filters, kernel_size, strides=2,
          padding = 'same',
          use_bias = False
      ), 10
  ))

  if dropout:
    pipeline.add(tf.keras.layers.Dropout(0.5))

  if activation:
    pipeline.add(tf.keras.layers.ReLU())

  return pipeline
```

Fig(3.9) Up Scaling

2) Generator:

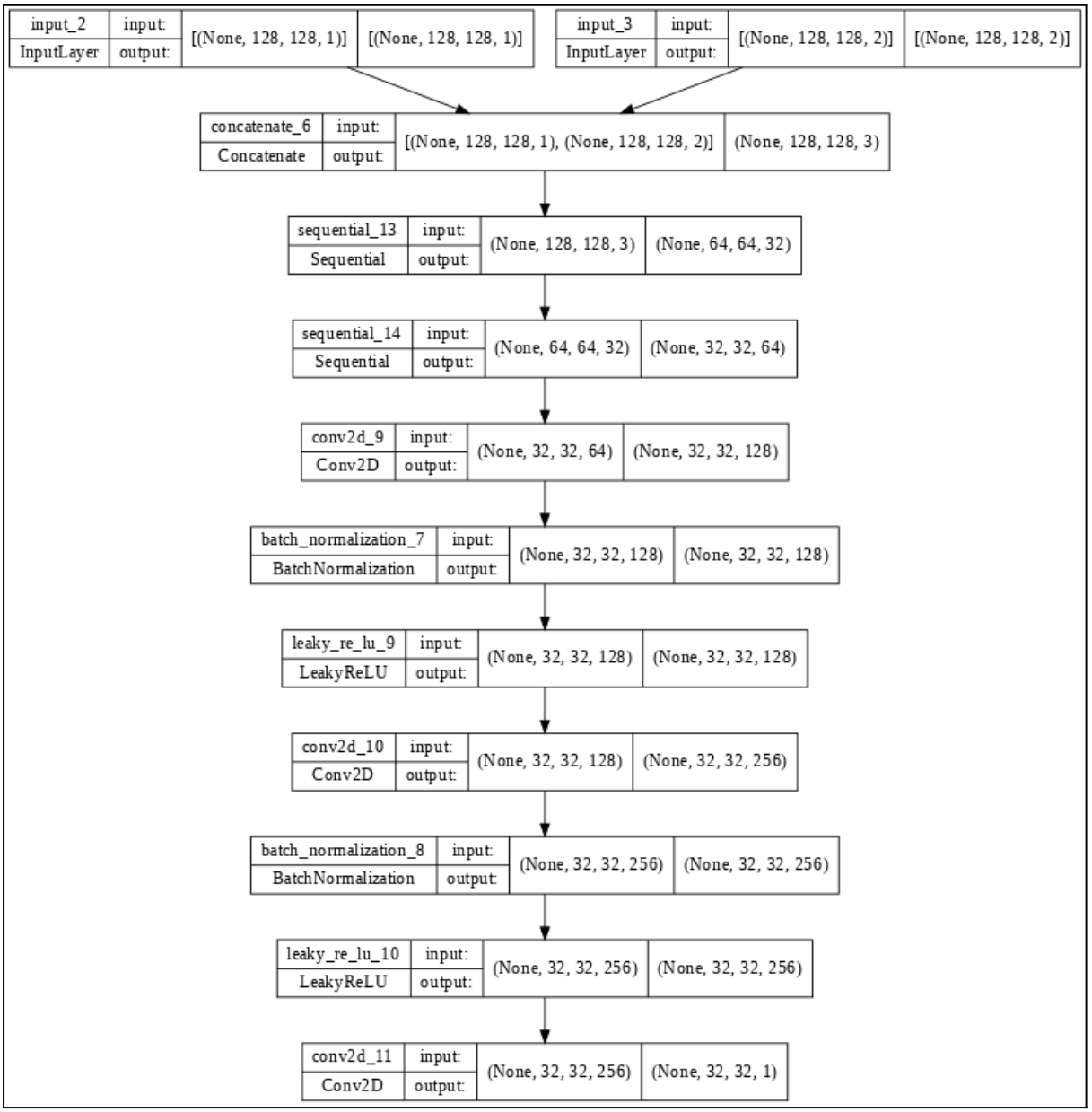
```
def Generator():
  inputs = tf.keras.layers.Input(shape=[128, 128, 1])

  downstack = [                     # output
        downsample(32, 4, False), # (64, 64, 32)
        downsample(64, 4),        # (32, 32, 64)
        downsample(128, 4),       # (16, 16, 64)
        downsample(128,4),        # (8, 8, 128)
        downsample(256,4),        # (4, 4, 128)
        downsample(256,4),        # (2, 2, 256)
        downsample(512,4),        # (1, 1, 512)
  ]

  upstack = [                       # output
        upsample(512, 4, True),   # (2, 2, 512)
        upsample(256, 4, True),   # (4, 4, 256)
        upsample(256, 4, True),   # (8 , 8, 256)
        upsample(128, 4, True),   # (16, 16, 128)
        upsample(128, 4),         # (32, 32, 128)
        upsample(64,  4),         # (64, 64, 64)
  ]

  last = tf.keras.layers.Conv2DTranspose(
      2, 4, strides=2,
      padding='same',
      activation = 'tanh')        # (128, 128, 2)

  x = inputs
  skips = []
  for down in downstack:
    x = down(x)
    skips.append(x)
  skips = reversed(skips[:-1])

  for id, (up, skip) in enumerate(zip(upstack, skips)):
    x = up(x)
    x = tf.keras.layers.Concatenate()([x, skip])

  x = last(x)
  return tf.keras.Model(inputs, x)
```



Fig(3.10) Generator

3) Discriminator:

```
def Discriminator():
  inp = tf.keras.layers.Input(shape=(128,128,1))
  tar = tf.keras.layers.Input(shape=(128,128,2))

  x = tf.keras.layers.concatenate([inp, tar])

  down1 = downsample(32, 4, batch_normalization = False)(x)       # (64, 64, 32)

  down2 = downsample(64, 4, )(down1)                              # (32, 32, 64)

  #down3 = downsample(256, 4, )(down2)

  conv1 = tf.keras.layers.Conv2D(128, 4, strides = 1, use_bias = False, padding = 'same') (down2)
  batchnorm1 = tf.keras.layers.BatchNormalization()(conv1)
  leaky_relu1 = tf.keras.layers.LeakyReLU()(batchnorm1)           # (32, 32, 128)

  #down4 = downsample(256, 4, )(down3)

  #zpad1 = tf.keras.layers.ZeroPadding2D()(down3)

  conv2 = tf.keras.layers.Conv2D(256, 4, strides = 1, use_bias = False, padding='same')(leaky_relu1)
  batchnorm2 = tf.keras.layers.BatchNormalization()(conv2)
  leaky_relu2 = tf.keras.layers.LeakyReLU()(batchnorm2)           # (32, 32, 256)

  last = tf.keras.layers.Conv2D(1, 4, strides=1, padding='same')(leaky_relu2)
                                                                  # (32, 32, 1)
  return tf.keras.Model(inputs=[inp, tar], outputs=last)
```



Fig(3.11) Discriminator

3.8.4 Training:

1) **Losses:**

- **Generator Loss:** This is the GAN loss that penalises possible structure difference between the generator output and target image

```
@tf.function()
def generator_loss(disc_gen_output, gen_target_img, target_img):
  labels = tf.ones_like(disc_gen_output)
  gan_loss = binary_loss(labels, disc_gen_output)
  struct_loss = tf.reduce_mean(tf.abs(target_img - gen_target_img))
  total_loss = gan_loss + (LAMBDA_STRUCT * struct_loss)

  return total_loss, gan_loss, struct_loss
```

Fig(3.12) Generator Loss

- **Discriminator Loss:** This is the GAN loss that penalises wrong predictions made by the discriminator.

```
@tf.function()
def discriminator_loss(disc_real_output, disc_gen_out):
  labels = tf.ones_like(disc_real_output)
  real_loss = binary_loss_smooth(labels, disc_real_output)

  labels = tf.zeros_like(disc_gen_out)
  fake_loss = binary_loss_smooth(labels, disc_gen_out)

  return real_loss + fake_loss
```

Fig(3.13) Discriminator Loss

2) **Optimizers:** Optimizers use gradient descent algorithms to reduce residual errors and help converge the cost function to minima.With various learning rates, we employed the Adam optimizer for the generator and discriminator.

```
generator_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0004)
discriminator_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001)
```

A lower learning rate is used for the discriminator so that only useful gradients are propagated back and there is no oscillation during the convergence of gradient descent.

GAN Model:

This model combines both the generator and discriminator, calculates losses and matrices, performs gradient descent steps and displays results.

```
class GAN(tf.keras.Model):
  def __init__(self, generator, discriminator, **kwargs):
    super(GAN, self).__init__(**kwargs)
    self.generator = generator
    self.discriminator = discriminator

  def compile(self, generator_optimizer, discriminator_optimizer, loss_fn, metric_fn):
    super(GAN, self).compile()
    self.generator_optimizer = generator_optimizer
    self.discriminator_optimizer = discriminator_optimizer
    self.loss_fn = loss_fn
    self.metric_fn = metric_fn

  def call(self, input, training = False):
    fake_tar_img = self.generator(input, training = training)
    return fake_tar_img
```

```python
def train_step(self, images):
  input_img, target_img = images
  with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
    fake_tar_img = self(input_img, training = True)
    disc_real_output = self.discriminator([input_img, target_img], training = True)
    disc_fake_output = self.discriminator([input_img, fake_tar_img], training = True)

    total_gen_loss, gan_loss, struct_loss = self.loss_fn['generator_loss'](disc_fake_output, fake_tar_img, target_img)
    disc_loss = self.loss_fn['discriminator_loss'](disc_real_output, disc_fake_output)

  generator_gradients = gen_tape.gradient(total_gen_loss, generator.trainable_variables)
  discriminator_gradients = disc_tape.gradient(disc_loss, discriminator.trainable_variables)

  self.generator_optimizer.apply_gradients(zip(generator_gradients, generator.trainable_variables))
  self.discriminator_optimizer.apply_gradients(zip(discriminator_gradients, discriminator.trainable_variables))

  self.metric_fn['total_gen_loss_tracker'].update_state(total_gen_loss)
  self.metric_fn['disc_loss_tracker'].update_state(disc_loss)

  return {
      'Avg_Total_Gen_Loss': self.metric_fn['total_gen_loss_tracker'].result(),
      'Disc_Loss': self.metric_fn['disc_loss_tracker'].result(),
  }
```

```python
def test_step(self, images):
  input_img, target_img = images
  fake_tar_img = self(input_img, training = True)

  disc_real_output = self.discriminator([input_img, target_img], training = True)
  disc_fake_output = self.discriminator([input_img, fake_tar_img], training = True)

  total_gen_loss, gan_loss, struct_loss = self.loss_fn['generator_loss'](disc_fake_output, fake_tar_img, target_img)
  disc_loss = self.loss_fn['discriminator_loss'](disc_real_output, disc_fake_output)

  self.metric_fn['total_gen_loss_tracker'].update_state(total_gen_loss)
  self.metric_fn['disc_loss_tracker'].update_state(disc_loss)

  return {
      'Avg_Total_Gen_Loss': self.metric_fn['total_gen_loss_tracker'].result(),
      'Disc_Loss': self.metric_fn['disc_loss_tracker'].result(),
  }
```

```python
def get_config(self):
  return {'generator': self.generator, 'discriminator': self.discriminator}

@property
def metrics(self):
  return [self.metric_fn['total_gen_loss_tracker'], self.metric_fn['disc_loss_tracker']]
```

Fig(3.14) Gan Architecture

3.9 User Interface

3.9.1 Model Deployment & Communication:

1) **Exporting TensorFlow model:**

Exporting a TensorFlow model in the SavedModel format is a crucial step in putting it into production. As a language-independent format for saving machine learning models, SavedModel is supported by TensorFlow and enables models to be loaded and used in a variety of programming languages. When a model is exported in the SavedModel format, TensorFlow maintains the trained model's graph, variables, and metadata describing the input and output signatures.

Since the SavedModel format is adaptable and extensible, changing the model or adding new features is a breeze. While the model's structure is saved in a protobuf file, the weights of the model are kept in a set of variables. The input and output tensors of the model, as well as their data types and formats, are also described in the protobuf file. The metadata for the saved model, which is kept separately in a file, includes details like the model's training parameters and the version of TensorFlow that was used to construct the SavedModel.

Once the model has been exported in the SavedModel format, it can be loaded and used in other programming languages like Python, C++, Java, and Go. Since different frameworks or programming languages are used in real-world settings, this makes it simple to deploy the model there. Additionally, the SavedModel format is made specifically for delivering models in applications in the real world.

2) **Build a web API:**

To deploy a TensorFlow model, you must construct a web API that can answer questions and make predictions. An assortment of protocols and building blocks known as a web API can be used to develop web-based services and applications that can communicate with one another. You can use well-known web frameworks like Flask, FastAPI, or Django to build a web API. These frameworks provide a quick and efficient way to create a RESTful API that can accept input data, process it using a deployed TensorFlow model, and output model predictions..

Your TensorFlow model may be made available as a web service by developing a web API, making it simple for other applications or services to use. Users can create, read, update, and delete resources using HTTP methods when using a RESTful API. When deploying a TensorFlow model, the API will be in charge of taking input data in a specific format, preprocessing it, feeding it to the deployed model, and delivering the model's predictions in the output format.

There are many benefits to creating a web API to launch your TensorFlow model. First of all, it makes it possible for you to swiftly integrate your model into other applications or services without having to provide users direct access to the model itself. This can be quite useful when you need to hide your model from end users due to security or intellectual property issues.

It can be easier to update and maintain your model over time using a well-designed API. By separating the model from the application code, you may make changes to the underlying model without having to rewrite the entire programme. This can expedite the procedure while lowering the likelihood of making errors or defects.

3) **Deploying the API on Nginx:**

Once a web framework has been used to create a web API that can accept requests and return predictions, the next step is to deploy the web API on a web server like Nginx. Nginx, a well-liked open-source web server, is praised for its dependability, scalability, and speed. Nginx is used to deliver online applications, including web APIs, in many production settings.

When you install your API on Nginx, you can use a reverse proxy to forward incoming requests to your API. Between a client and an API server, there is a server known as a "reverse proxy" that routes incoming requests depending on the URL of the request or other criteria to the correct server.

4) **Configuring Nginx:**

Setting up Nginx to send requests to your API is necessary for deploying your TensorFlow model in a production environment. The Nginx configuration file contains directives that describe how Nginx ought to respond to incoming requests and communicate with your API server.

To configure Nginx to forward requests to your API, you must make changes to the configuration file. The location directive specifies the URL endpoint for your API.

In addition to the location directive, you might also need to set the server

block and the proxy_pass directive in the Nginx configuration file. While the server block specifies the IP address and port number that Nginx should listen to for incoming requests, the proxy_pass directive routes incoming requests to the API server. You might also need to specify additional parameters, such as SSL certificates and caching, depending on your specific use case. Test your deployment after setting up Nginx by sending requests to the API endpoint and ensuring that the responses are correct.

5) **Testing the deployment:**

Testing the deployment of a TensorFlow model on an API server is an essential step in the deployment procedure. Testing demonstrates that the deployed model is functioning as expected and that the API server is configured properly to manage incoming requests and produce the desired outcomes.

To test the deployment and see if the response contains the expected predictions, you may make test requests to the API endpoint. You may use a variety of tools, such as web browser extensions and command-line utilities like curl and Postman, to send queries to the API. Testing the API under various conditions is essential, including delivering different input data types, searching for errors and exceptions, and testing edge situations.

When evaluating the TensorFlow model's deployment on an API server, it is critical to take the system's performance and scalability into account. This involves looking for any bottlenecks or potential performance concerns and evaluating how the system responds to the increasing number of incoming requests.

3.9.2 Frontend

1. About ReactJS:

ReactJS, more often known as React, is an open-source JavaScript library developed by Facebook for single-page user interfaces (UIs) and applications. Since it makes it possible to create reusable UI components and manage application state effectively, React is a preferred option among developers worldwide.

One of React's primary features is its component-based design. Developers may create unique UI elements that can be utilised across several pages or even whole projects. This reduces the amount of code required to construct an application and makes it simpler to maintain and update.

Another feature utilised by React is a virtual DOM (Document Object Model), which is a condensed version of the real DOM. Thanks to the virtual DOM, React is incredibly efficient and quick because it only updates the components that need to be updated. This results in a speedier website load time and a better user experience.

The state of an application may also be managed via React. React provides a simple and obvious way to manage the state of an application using props and state. State controls a component's internal data, whereas props are used to convey data between components.

React also provides a declarative UI development approach. The user interface (UI) may be designed by developers, while React takes care of the rest. As a result, there is a lower possibility of defects and mistakes, and understanding the code is also made easier. Many libraries and technologies that are compatible with React have been developed with the help of the large and active React community.

2. Control Flow:

2.1. User Upload Image:

The user selects an image file from their local device and uploads it to the website server. Once submitted, the image is then momentarily stored on the server.

2.2. The image preview is shown on the website:

A preview of the image is displayed on the website after it is posted so that the user can confirm that it was uploaded properly. The picture file is rendered on the page in this preview using HTML and CSS.

2.3. On clicking the submit button, three background processes occur:

The provided picture is moved from its temporary place on the server to a web hosting platform where it is saved and made accessible to other users and operations.

2.3.1. First image is hosted on an online hosting platform:

The provided picture is moved from its temporary place on the server to a web hosting platform where it is saved and made accessible to other users and operations.

2.3.2. Then this hosted image is processed through our API & colourized by the model:

The hosted image is processed using an API, and then colorization is added using a machine learning model. In this process, computer vision algorithms are used to find and anticipate the image's most likely colour applications.

2.3.3. The B&W image is replaced by the coloured image in the

preview:

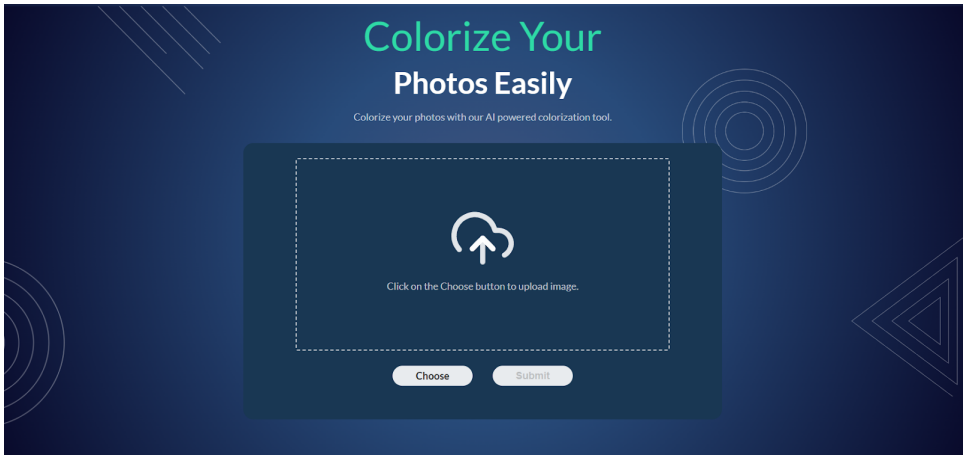
After the colorization process is complete, the black and white (B&W) image that was visible in the preview is replaced with the newly coloured image. This action, which dynamically updates the preview picture without requiring a page reload, is performed using JavaScript.

2.4. The download button is available to download the coloured image:

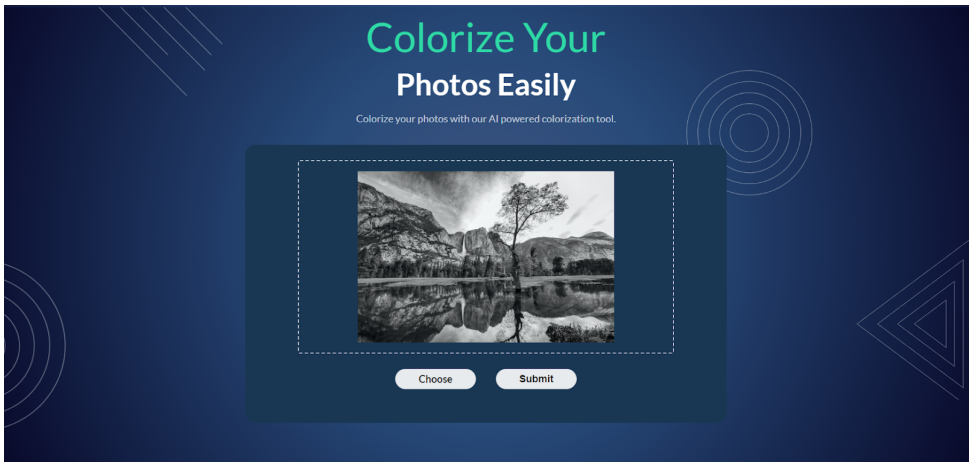
The colourized image may be downloaded by the user by clicking a download button once it has been displayed. This button initiates a process that downloads the colourized image from the internet hosting platform to the user's local device.

3) Functioning of the WebApp:

Step-1) User clicks on the choose button to upload the desired B&W image.

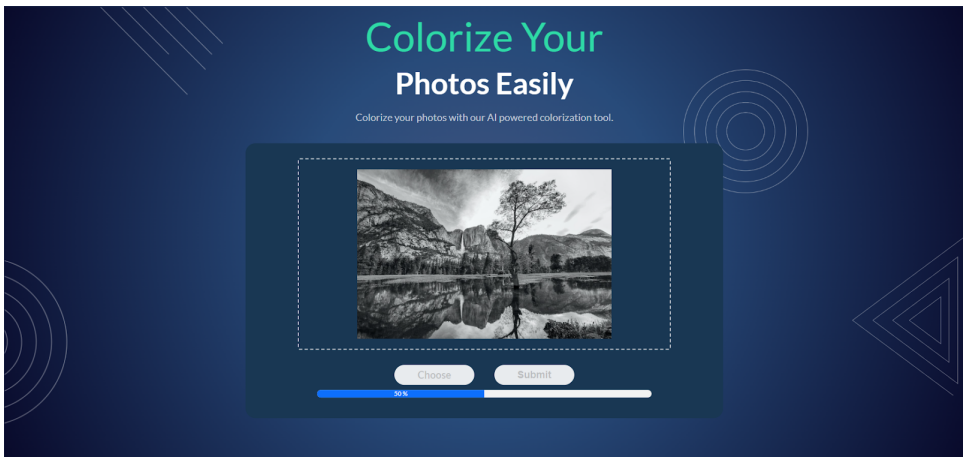


Step-2) The uploaded image preview is shown on the website, Now the user can either replace the current image or click on the submit button to begin the process of Colorization.

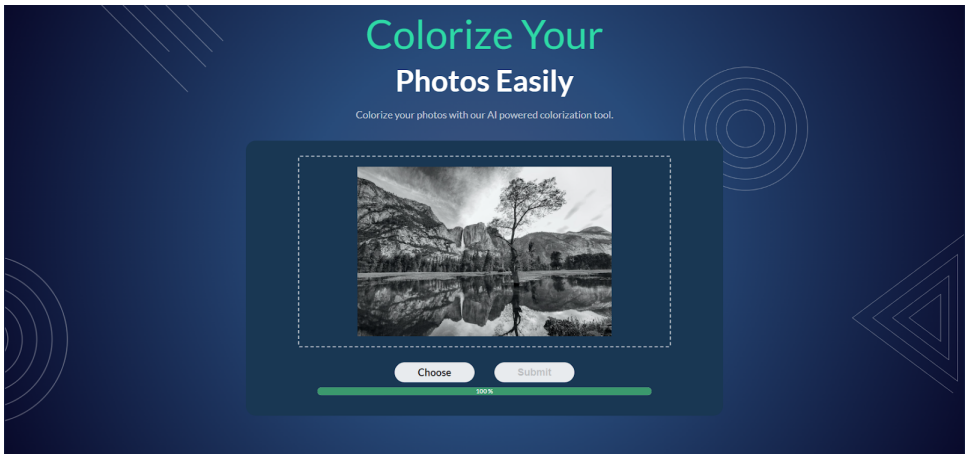


Step-3) Once the user clicks on the submit button, the four step

colorization process begins which includes hosting of the image, processing the image through model pipeline, actual colorization of image, returning the coloured image to the web portal.

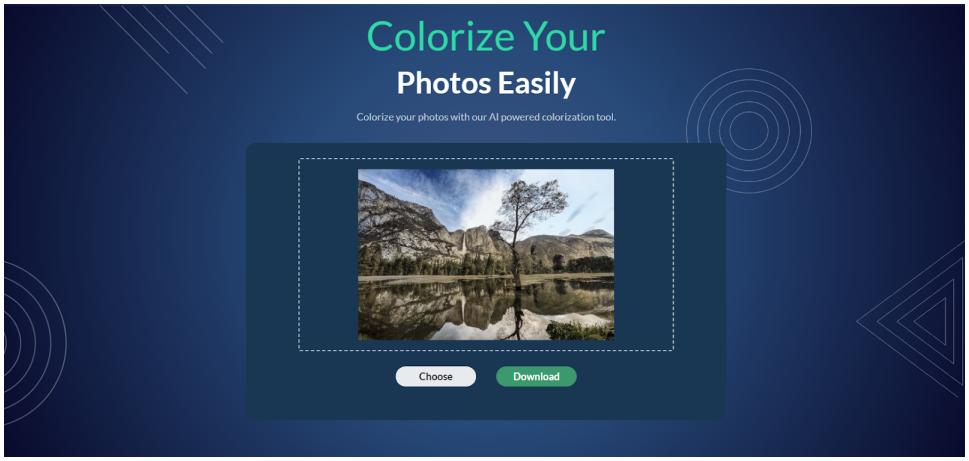


The progress of the process can be seen visually on the progress bar.



Step-4) After the colorization process is completed, the colourized image

preview is automatically displayed on the website.The download button is now available



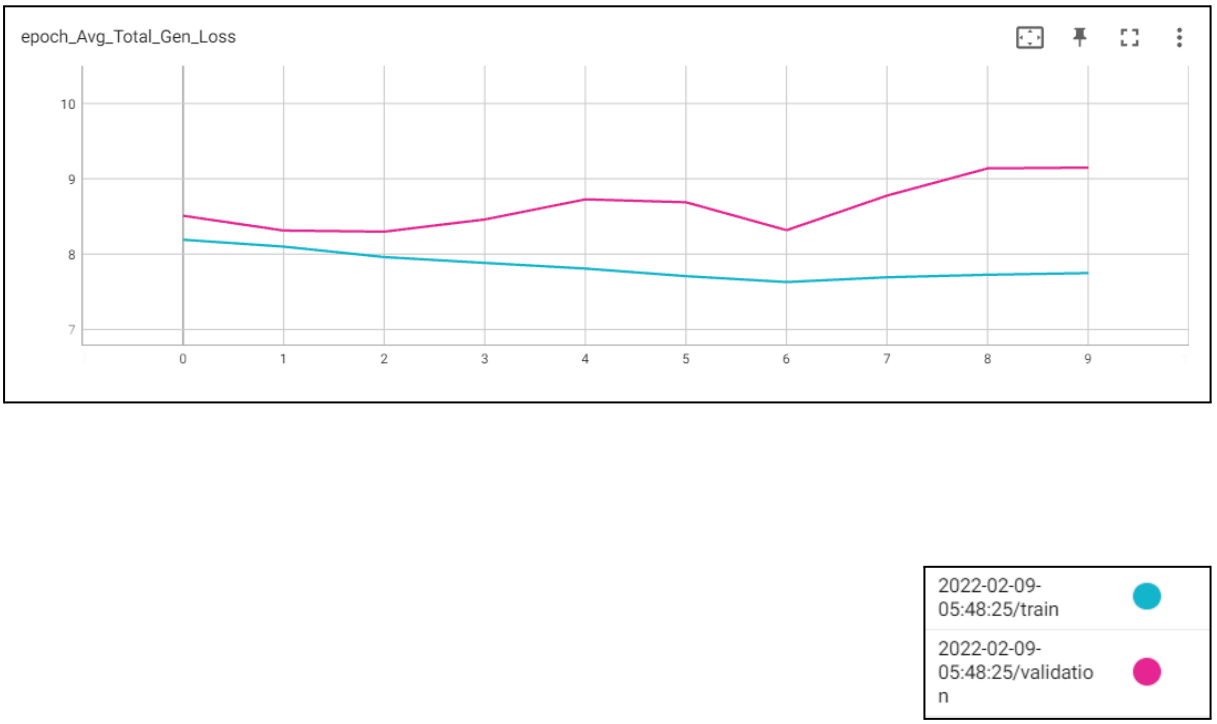
Step-5) Users can click on the download button to save the colourized image locally on their system.

CHAPTER 04:

EXPERIMENTS & RESULT ANALYSIS

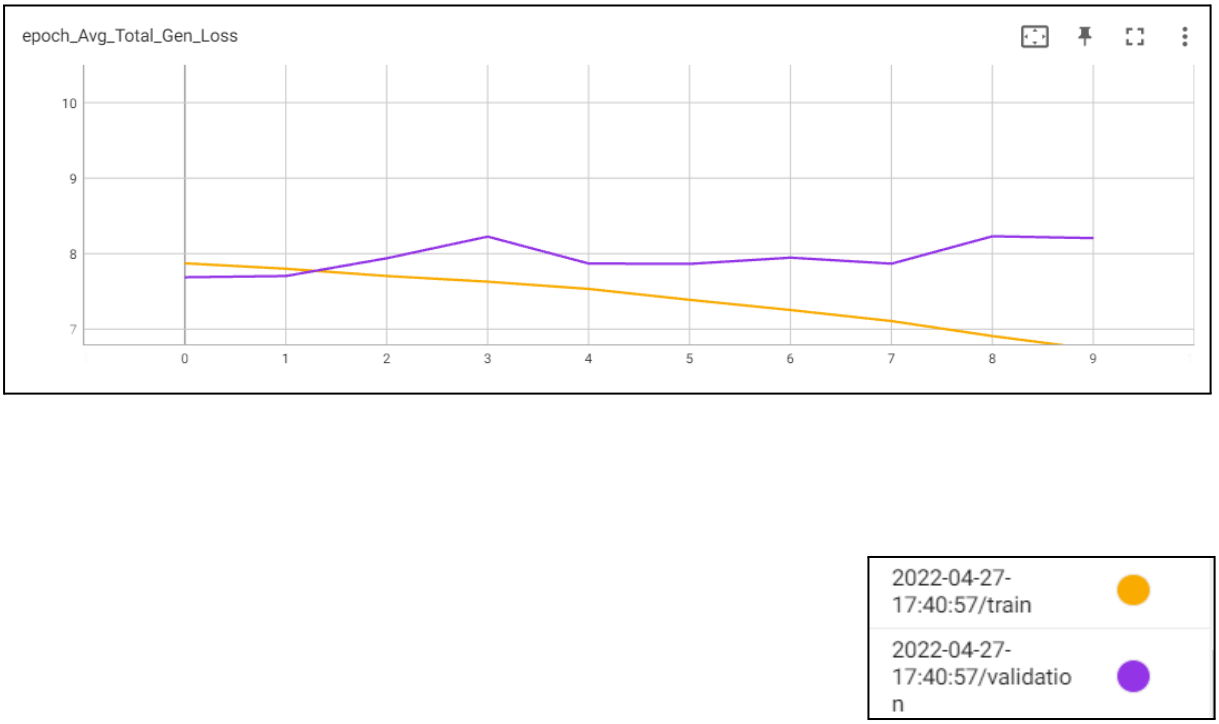
Discussion on the Results Achieved (All graphs are loss v/s epochs)

This is the resulting graph when image colorization task is performed on RGB images, the overfitting is due to greater no. of model parameters as RGB images have three colour feature maps.



Graph(4.1) Avg. Total Gen. Loss

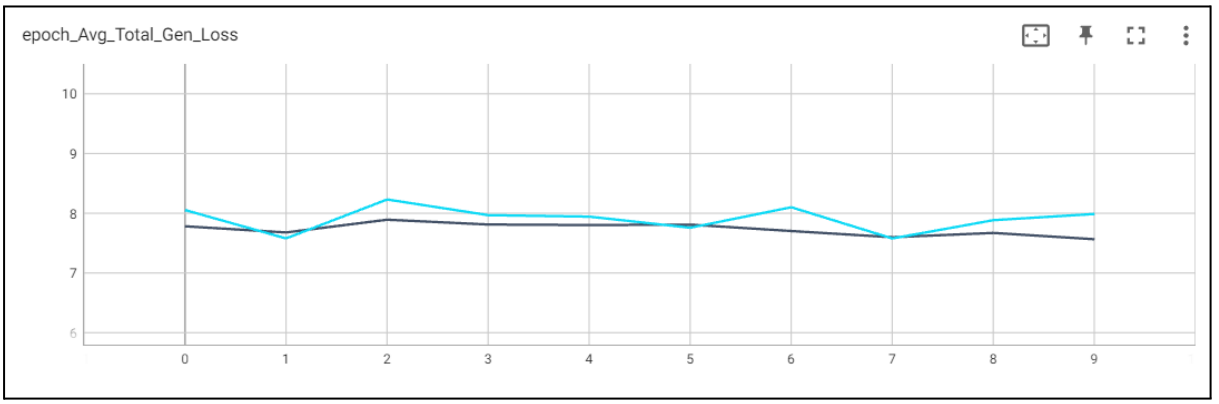
Reduction in overfitting when using L*a*b* colour space images which have two output feature map a*b*.

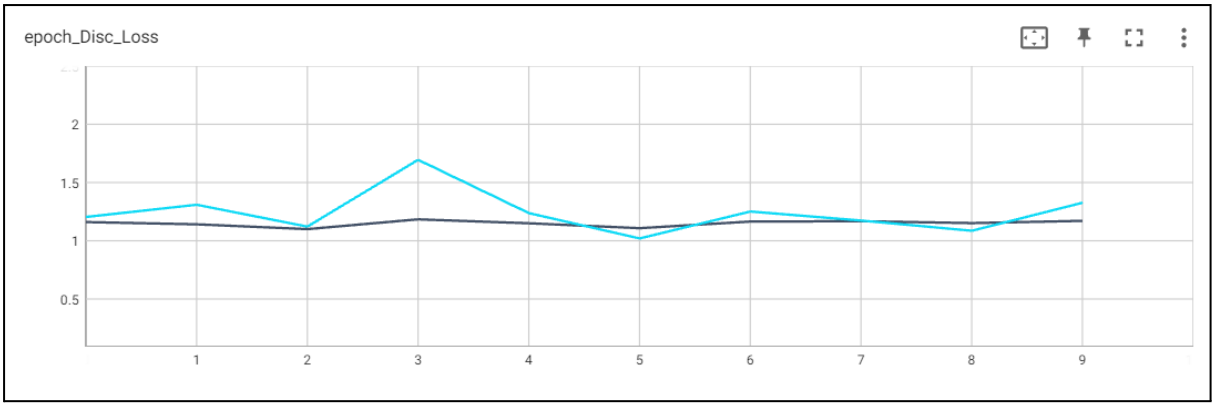


Graph(4.2) Optimised Avg. Total Gen. Loss

Resultant graph after performing hyper parameter tuning.

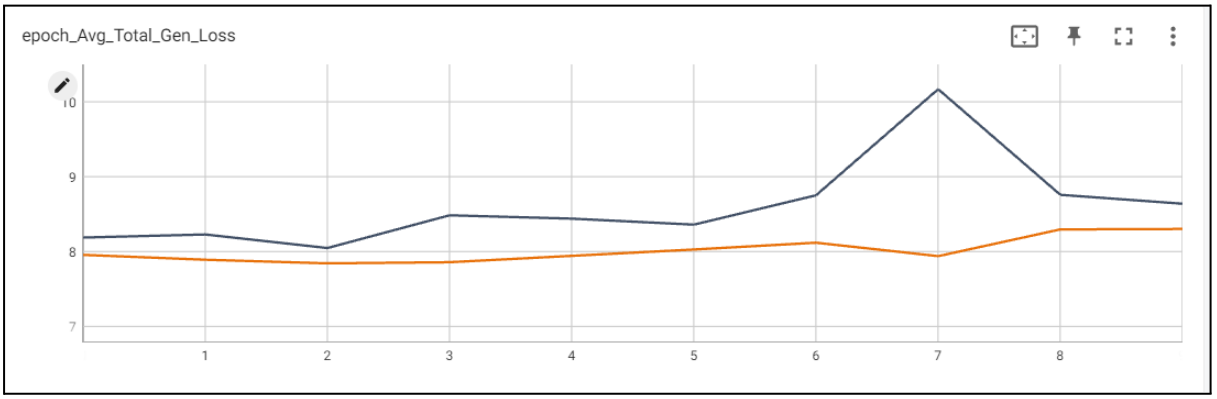
- Batch size = **128**
- Label smoothing = **0.1**
- LAMBDA (Fraction importance to image structural loss) = **100**

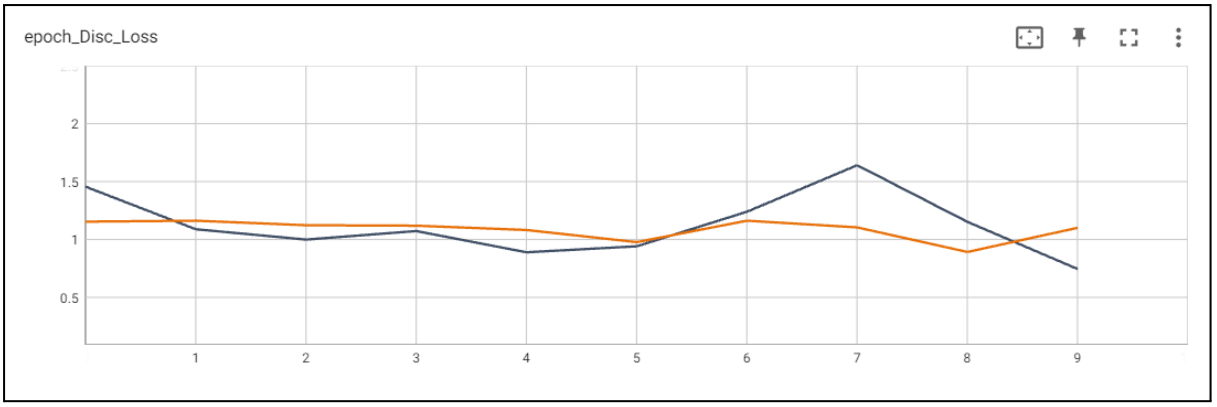




Graph(4.3) Disc.. Loss

The final graph after applying Spectral Normalisation, which stabilises the training of GANs by limiting the layer's spectral norm and regulates the Lipschitz factor of the layer.





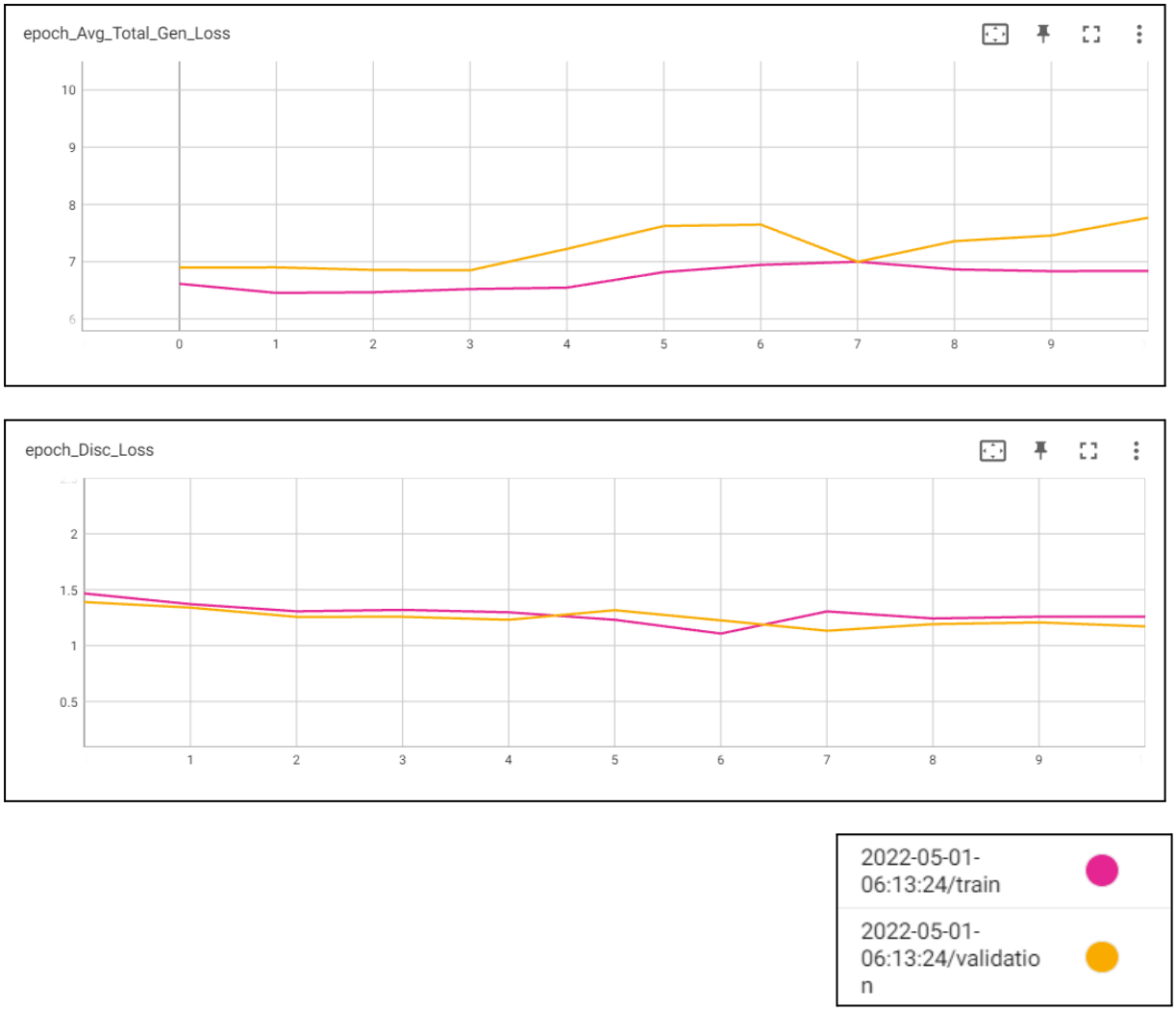
Graph(4.4) Avg. Total Gen. Loss using Spectral Normalisation

Resultant graph after setting advanced hyper parameters.

- Power iterations of Spectral Normalisation. (Improves the approximation of weight normalisation.) = **10**

- Beta_1 of Adam optimizer (Lower momentum helps in stable training and prevents gradient oscillations. = **0.5**

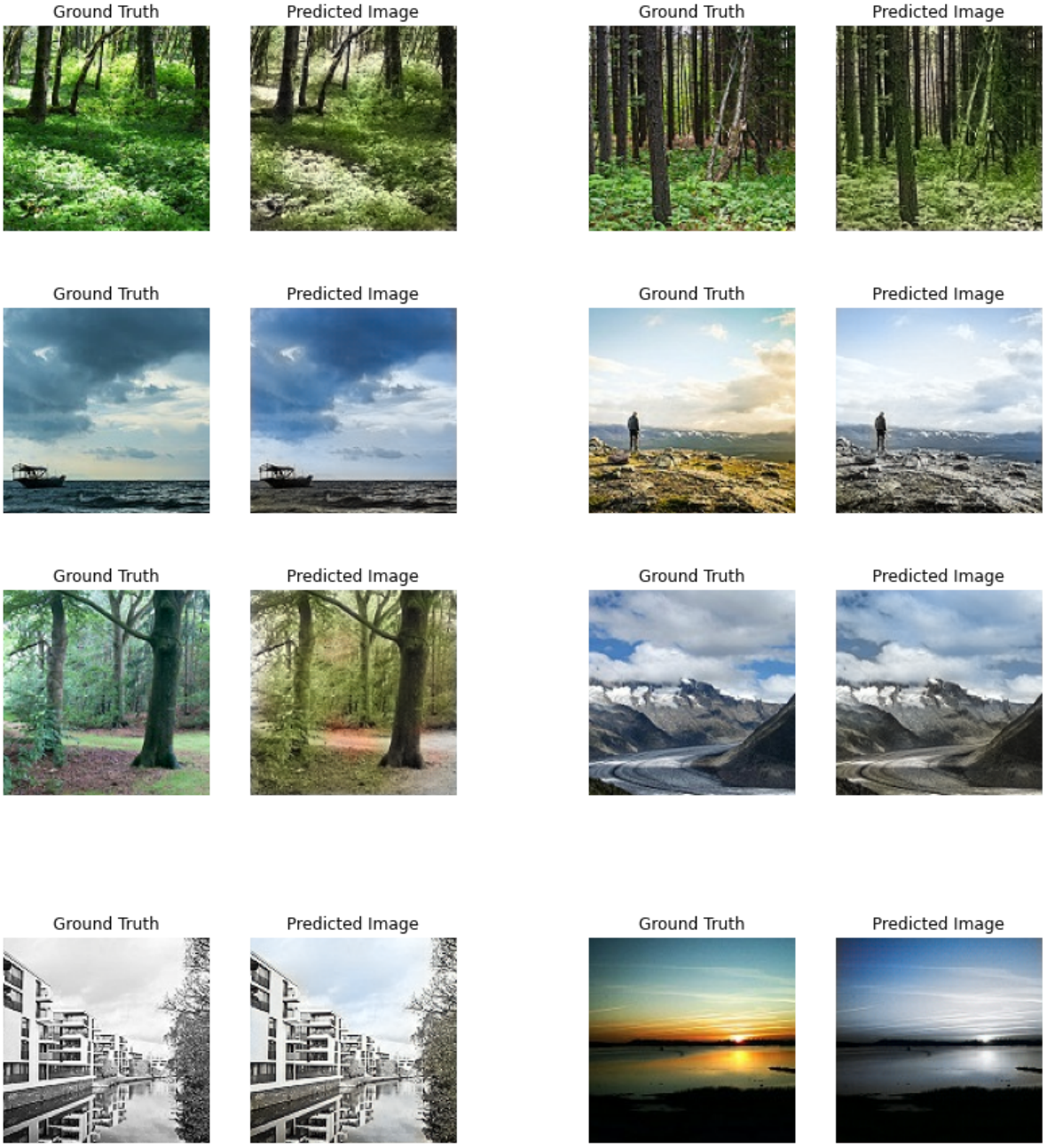
- Learning rate of Adam optimizer (Smaller & different earning rates for both the generator and discriminator help in model convergence as suggested by results) = **0.0004** & **0.001** respectively.

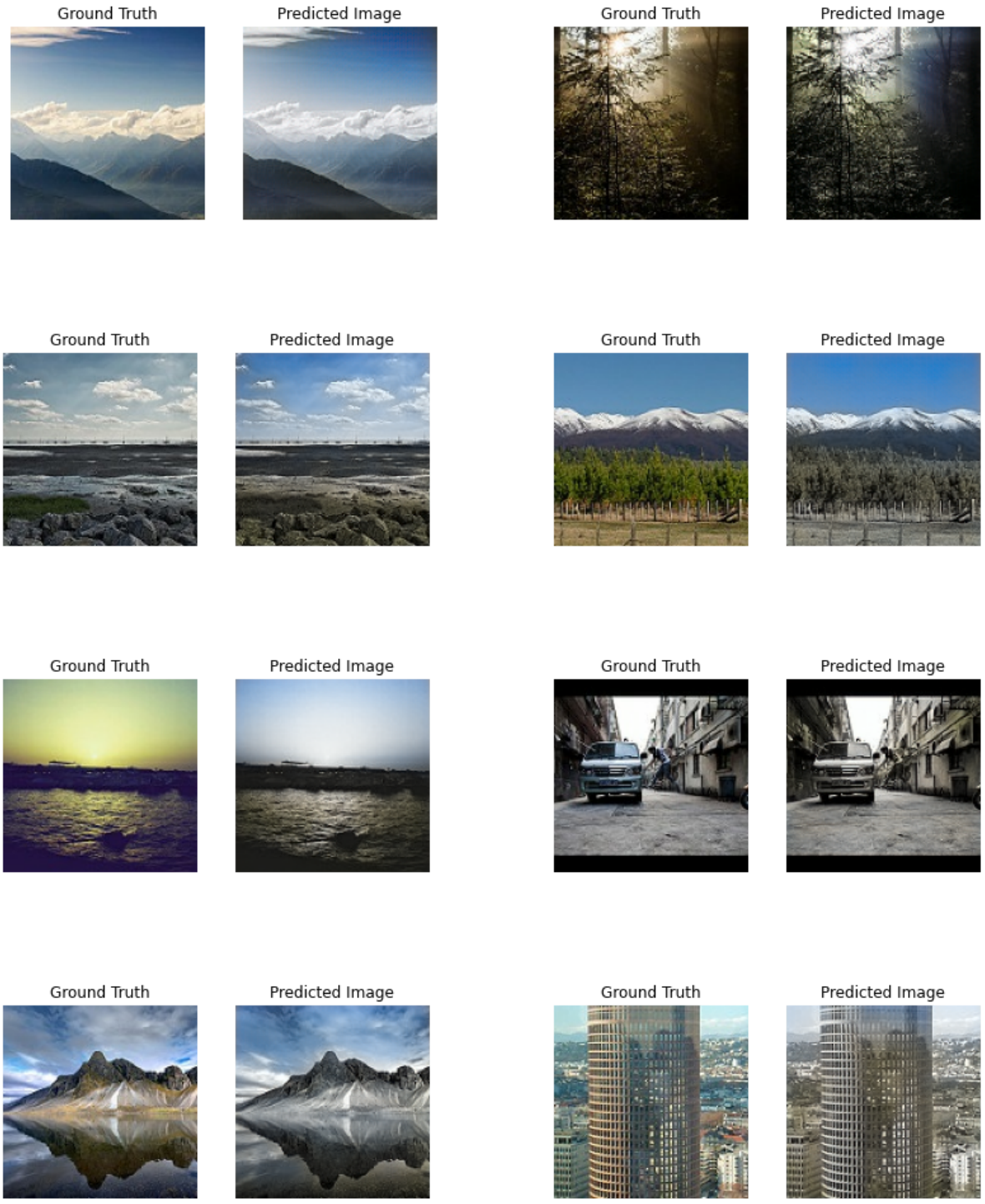


Graph(4.5) Resultant disc. graph after setting advanced hyper parameters

CHAPTER 05:
CONCLUSION

5.1 Image Colorization Results





Fig(5.1) Results

5.2 Application of the Major Project

- Old photos restoration and enhancement.
- To convert X-Rays and ultrasounds to coloured images.
- In CCTVs to improve night vision.
- Satellite imagery translation to 2D Maps.

5.3 Limitation of the Major Project

- It is challenging to create very high resolution, crisp, and colourful images..
- Model reaches Nash equilibrium but is quite unstable.
- Images that are highly detailed & diverse in respect of colours may have some artefacts in patches.

5.4 Future Work

Therefore, our goal and possible areas would be to produce visually better, more colourful images and use better quantitative matrices like peak signal-to-noise ratio (PSNR), which will enable a much more robust process of performance. GANs have been known to be very difficult to train because it requires finding a Nash equilibrium.

REFERENCES

1) P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, "Image-to-Image Translation with Conditional Adversarial Networks," in IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 5967-5976.

2) K. Nazeri, E. Ng, and M. Ebrahimi, "Image Colorization," in 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2018, pp. 2462-2466.

3) T. Welsh, M. Ashikhmin, and K. Mueller, "Transferring colour to grayscale images," in ACM Transactions on Graphics (TOG), vol. 21, no. 3, pp. 277-280, 2002.

4) A. Levin, D. Lischinski, and Y. Weiss, "Colorization using optimization," in ACM Transactions on Graphics (TOG), vol. 23, no. 3, pp. 689-694, 2004.

5) P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, "Image-to-image translation with conditional adversarial networks," 2016, arXiv:1611.07004.

6) M. Mirza and S. Osindero, "Conditional generative adversarial nets," 2014, arXiv:1411.1784.

7) S. Ioffe and C. Szegedy, "Batch normalisation: Accelerating deep network training by reducing internal covariate shift," in Proceedings of the International Conference on Machine Learning (ICML), 2015, pp. 448-456.

8) A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," in International Conference on Learning Representations (ICLR), 2016.

9) C. Li and M. Wand, "Precomputed real-time texture synthesis with Markovian generative adversarial networks," in European Conference on Computer Vision (ECCV), 2016, pp. 702-716.

10) D. Pathak, P. Krahenbuhl, J. Donahue, T. Darrell, and A. A. Efros, "Context encoders: Feature learning by inpainting," in IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 2536-2544.

11) R. Zhang, P. Isola, and A. A. Efros, "Colourful image colorization," in European Conference on Computer Vision (ECCV), 2016, pp. 649-666.

12) I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in Advances in Neural Information Processing Systems (NIPS), 2014, pp. 2672-2680.

13) A. B. L. Larsen, S. K. Sønderby, and O. Winther, "Autoencoding beyond pixels using a learned similarity metric," in International Conference on Machine Learning (ICML), 2016, pp. 1558-1566.

14) O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. "Imagenet large scale visual recognition challenge." International Journal of Computer Vision, vol. 115, no. 3, pp. 211-252, 2015.

15) M. Mathieu, C. Couprie, and Y. LeCun. "Deep multi-scale video prediction beyond mean square error." ICLR, 2016.

16) J. Johnson, A. Alahi, and L. Fei-Fei, "Perceptual Losses for Real-Time Style Transfer and Super-Resolution," in European Conference on Computer Vision (ECCV), 2016, pp. 694-711.

17) R. Huang, S. Zhang, T. Li, and R. He, "Beyond Face Rotation: Global and Local Perception GAN for Photorealistic and Identity Preserving Frontal View Synthesis," in IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 2439-2448.

18) J. Yim, H. Jung, B. Yoo, C. Choi, D. Park, and J. Kim, "Pixel-level Domain Transfer," in Conference on Neural Information Processing Systems (NeurIPS), 2017, pp. 1704-1713.

19) A. Shrivastava, T. Pfister, O. Tuzel, J. Susskind, W. Wang, and R. Webb, "Learning from Simulated and Unsupervised Images through Adversarial Training," in IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 2242-2251.

20) L. Gatys, A. Ecker, and M. Bethge, "Texture Synthesis Using Convolutional Neural Networks," in Advances in Neural Information Processing Systems (NIPS), 2015, pp. 262-270.

21) J. Kim, J. Lee, and K. Mu Lee, "Deeply-Recursive Convolutional Network for Image Super-Resolution," in IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 1637-1645.

22) C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, et al., "Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network," in IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 105-114.

23) J. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks," in IEEE International Conference on Computer Vision (ICCV), 2017, pp. 2223-2232.

24) S. Reed, Z. Akata, X. Yan, L. Logeswaran, B. Schiele, and H. Lee, "Generative Adversarial Text to Image Synthesis," in International Conference on Machine Learning (ICML), 2016, pp. 1060-1069.

25) J. Wang, X. Yang, C. Liu, Y. Huang, Z. Liu, and W. Xu, "High-Resolution Image Synthesis and Semantic Manipulation with Conditional GANs," in IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018, pp. 8798-8807.