CSE 676 Deep Learning Final Project

Sankalp Mehani (sankalpm), Deepthi D'Souza (deepthid)

July 5, 2023

1 Problem Statement

This project aims to develop a Deep Convolutional Generative Adversarial Network (DCGAN) model for automatic colorization of greyscale images. Colorization enhances the visual appeal and interpretability of greyscale images, with applications in image restoration, historical photo colorization, and artistic rendering. The DCGAN model will be trained on paired greyscale and color image datasets, enabling it to generate realistic and visually coherent colorized outputs. By evaluating the model's performance using objective metrics and human evaluations, this project aims to improve its accuracy and explore the potential impact of architectural choices, loss functions, and training strategies. This project has the potential to advance computer vision applications and enhance the way we perceive and interact with visual content.

2 The Dataset

The dataset used for this project is the popular Cifar-10 dataset. 50,000 images were downloaded to comprise the training set and 10,000 images for the test set. These RGB images were converted to the LAB color space which consists of three channels: L for lightness, A for green-red axis, and B for blue-yellow axis.
We utilized the L channel as the input to the generator network. The L channel represents the grayscale version of the original image and contains information about the image's brightness and contrast. By using the L channel as the generator input, we enabled the model to focus solely on learning the colorization process without being concerned with the brightness and contrast aspects.
The AB channels, representing the color information in the LAB color space, are used as the target for the discriminator network. Along with the input image's L channel, the AB channels are sent to the discriminator for comparison. The discriminator's role is to determine the authenticity of the generated colorization by assessing how well the AB channels match the ground truth color information.

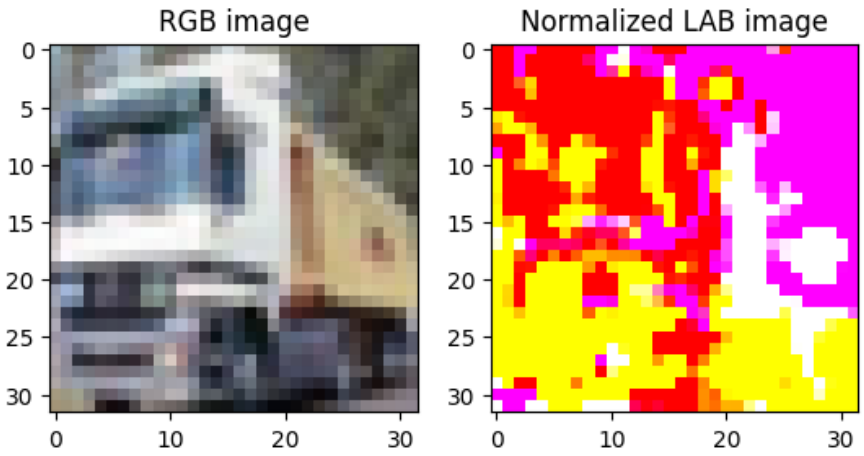

Figure 1: RGB image in LAB color space

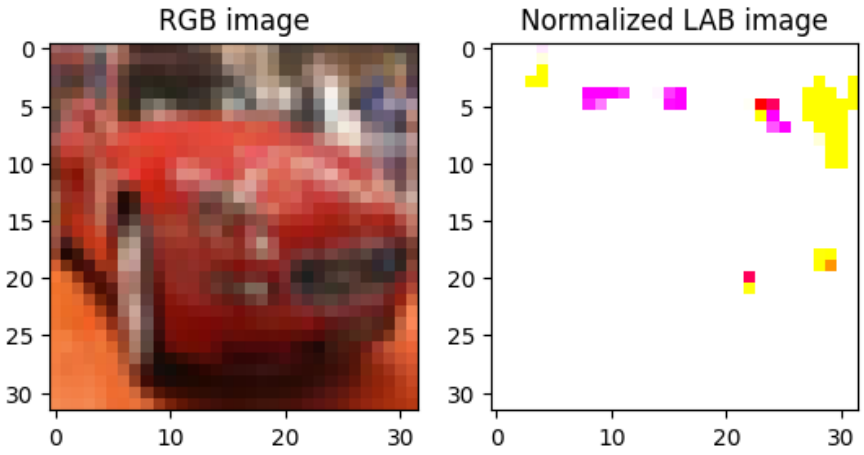

Figure 2: RGB image in LAB color space

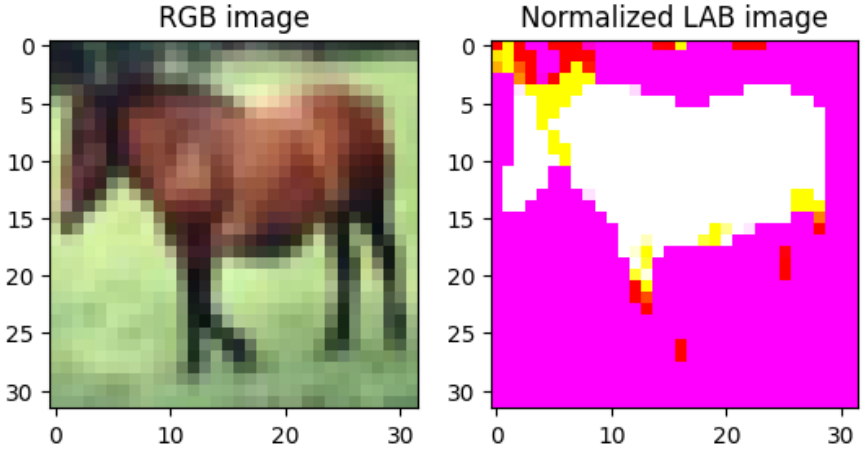

Figure 3: RGB image in LAB color space

3 Architecture 1

3.1 Generator and Discriminator

Two crucial components of a DCGAN architecture are the two sub-models: Generator and Discriminator.
Generator: The Generator transforms greyscale images into colorized versions using deep convolutional neural networks. By learning from paired greyscale and color images, it generates visually appealing and realistic colorizations.
Discriminator: The Discriminator is a binary classifier that distinguishes between real color images and colorized images generated by the Generator. It is trained adversarially with the Generator to accurately classify real and generated color images, improving the Generator's colorization capabilities.
The optimizer function for both of these sub-models is the Adam optimizer function. BCELoss is used as the loss criterion.

```python
class GeneratorModel(tf.keras.Model):
    def __init__(self):
        super(GeneratorModel, self).__init__()

        self.conv1 = Conv2D(32, (3, 3), padding='same', activation='relu', input_shape=(IMAGE_SIZE, IMAGE_SIZE, 1))
        self.conv2 = Conv2D(64, (3, 3), padding='same', strides=2, activation='relu')
        self.batchnorm1 = BatchNormalization()
        self.conv3 = Conv2D(128, (3, 3), padding='same', activation='relu', strides=2)
        self.conv4 = Conv2D(128, (3, 3), padding='same', activation='relu')
        self.batchnorm2 = BatchNormalization()
        self.conv5 = Conv2D(256, (3, 3), padding='same')
        self.activation = Activation('relu')
        self.batchnorm3 = BatchNormalization()
        self.upsampling1 = UpSampling2D(size=(2, 2))
        self.conv6 = Conv2D(128, (3, 3), padding='same', activation='relu')
        self.batchnorm4 = BatchNormalization()
        self.upsampling2 = UpSampling2D(size=(2, 2))
        self.conv7 = Conv2D(64, (3, 3), padding='same', activation='relu')
        self.batchnorm5 = BatchNormalization()
        self.conv8 = Conv2D(32, (3, 3), padding='same', activation='relu')
        self.conv9 = Conv2D(2, (3, 3), padding='same')
        self.batchnorm6 = BatchNormalization()
        self.activation2 = Activation('tanh')

    def call(self, inputs):
        x = self.conv1(inputs)
        x = self.conv2(x)
        x = self.batchnorm1(x)
        x = self.conv3(x)
        x = self.conv4(x)
        x = self.batchnorm2(x)
        x = self.conv5(x)
        x = self.activation(x)
        x = self.batchnorm3(x)
        x = self.upsampling1(x)
        x = self.conv6(x)
        x = self.batchnorm4(x)
        x = self.upsampling2(x)
        x = self.conv7(x)
        x = self.batchnorm5(x)
        x = self.conv8(x)
        x = self.conv9(x)
        x = self.batchnorm6(x)
        x = self.activation2(x)
        return x
```

Figure 4: Generator Model Architecture

```python
# Define the Autoencoder Generator Model Architecture
class AutoencoderGeneratorModel(Model):
    def __init__(self):
        super(AutoencoderGeneratorModel, self).__init__()

        # Define input layer
        self.inputs = Input(shape=(IMAGE_SIZE, IMAGE_SIZE, 1))

        # Define downsampling layers
        self.downstack = [
            downsample(32, 4, apply_batchnorm=False),
            downsample(64, 4),
            downsample(128, 4),
            downsample(256, 4),
            downsample(256, 4)
        ]

        # Define upsampling layers
        self.upstack = [
            upsample(256, 4, apply_dropout=True),
            upsample(128, 4),
            upsample(64, 4),
            upsample(32, 4),
        ]

        # Define output layer
        initializer = tf.random_uniform_initializer(0, 0.02)
        self.output_layer = Conv2DTranspose(2, 3, strides=2, padding='same', kernel_initializer=initializer, activation='tanh')

    def call(self, image):

        # Downsample the image
        skips = []
        for downsample_image in self.downstack:
            image = downsample_image(image)
            skips.append(image)

        # Reverse the skips list and exclude the last element
        skips = reversed(skips[:-1])

        # Upsample the image
        for upsample_image, skip in zip(self.upstack, skips):
            image = upsample_image(image)
            image = tf.concat([image, skip], axis=-1)

        # Apply the output layer
        image = self.output_layer(image)
        return image
```

Figure 5: Autoencoder Generator Model Architecture

```python
class DiscriminatorModel(Sequential):
    def __init__(self):
        super(DiscriminatorModel, self).__init__()

        self.add(Conv2D(32, (3, 3), padding='same', activation='relu', input_shape=(IMAGE_SIZE, IMAGE_SIZE, 3)))
        self.add(Conv2D(32, (3, 3), padding='same', activation='relu'))
        self.add(AveragePooling2D(pool_size=(2, 2)))
        self.add(Dropout(0.25))

        self.add(Conv2D(64, (3, 3), padding='same', activation='relu'))
        self.add(Conv2D(64, (3, 3), padding='same', activation='relu'))
        self.add(AveragePooling2D(pool_size=(2, 2)))
        self.add(Dropout(0.25))

        self.add(Flatten())
        self.add(Dense(512))
        self.add(LeakyReLU(0.2))
        self.add(BatchNormalization())
        self.add(Dropout(0.5))
        self.add(Dense(1))
        self.add(Activation('sigmoid'))
```

Figure 6: Discriminator Model Architecture

3.2 Training

The Generator model was trained for 50 epochs. The loss metric was observed to drop from 38.12 to 17.09, indicating a significant reduction. Training the model for more epochs coupled with fine tuning the hyperparameters, is expected to yield even better results.

```
Epoch 1: gen loss: 37.37885284423828, disc loss: 5.242197513580322
Epoch 2: gen loss: 23.28597068786621, disc loss: 4.678247451782227
Epoch 3: gen loss: 22.862367630004883, disc loss: 4.427259922027588
Epoch 4: gen loss: 22.481117248535156, disc loss: 4.305530548095703
Epoch 5: gen loss: 22.25856590270996, disc loss: 4.262806415557861
Epoch 6: gen loss: 22.099830627441406, disc loss: 4.248347282409668
Epoch 7: gen loss: 21.977008819580078, disc loss: 4.239887237548828
Epoch 8: gen loss: 21.86307144165039, disc loss: 4.23980188369751
Epoch 9: gen loss: 21.75882339477539, disc loss: 4.23942756652832
Epoch 10: gen loss: 21.673133850097656, disc loss: 4.239718914031982
```

Figure 7: Training the model for 10 epochs

Generator loss represents how effectively the generator network is able to fool the discriminator by generating colorized images that are visually similar to real images. Throughout the training process, the generator loss steadily decreases. This decrease indicates that the generator is progressively improving its ability to generate more convincing colorizations.

Discriminator loss measures the ability of the discriminator network to correctly classify between real and generated colorized images. Similar to the generator loss, we observe a consistent decrease in the discriminator loss during training. This reduction signifies that the discriminator is becoming more proficient at distinguishing between real and generated images.

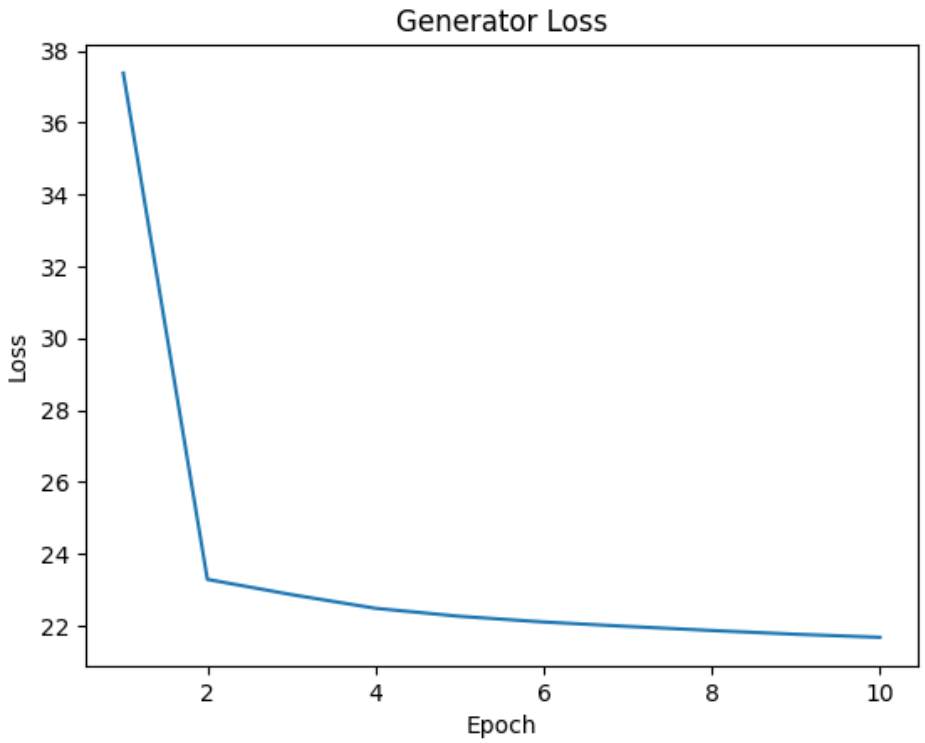

Figure 8: Generator loss after 10 epochs

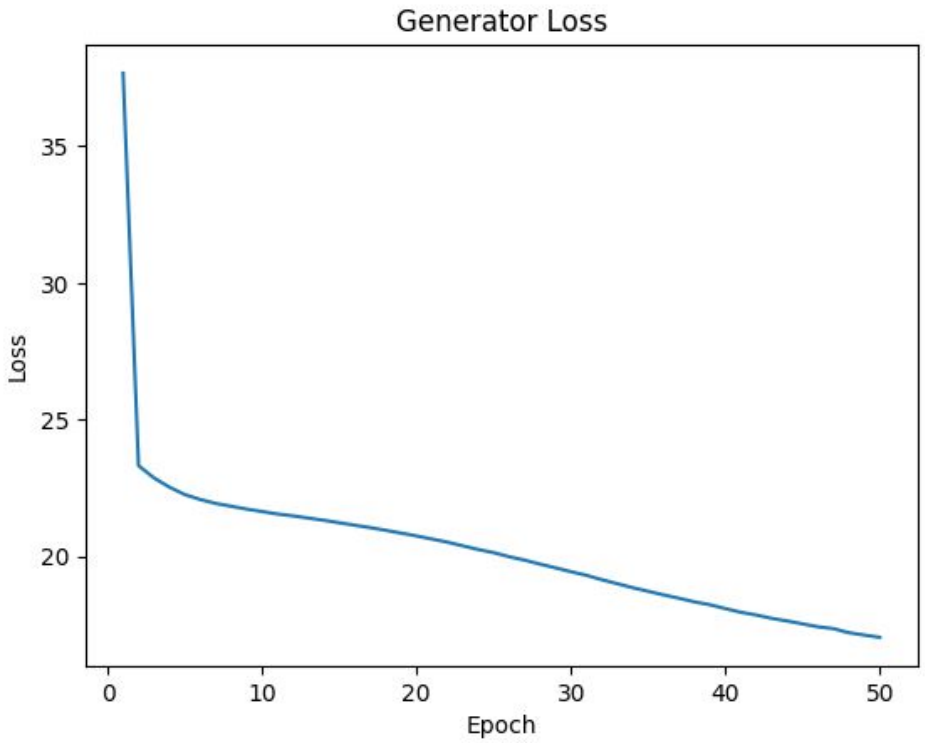

Figure 9: Generator loss after 50 epochs

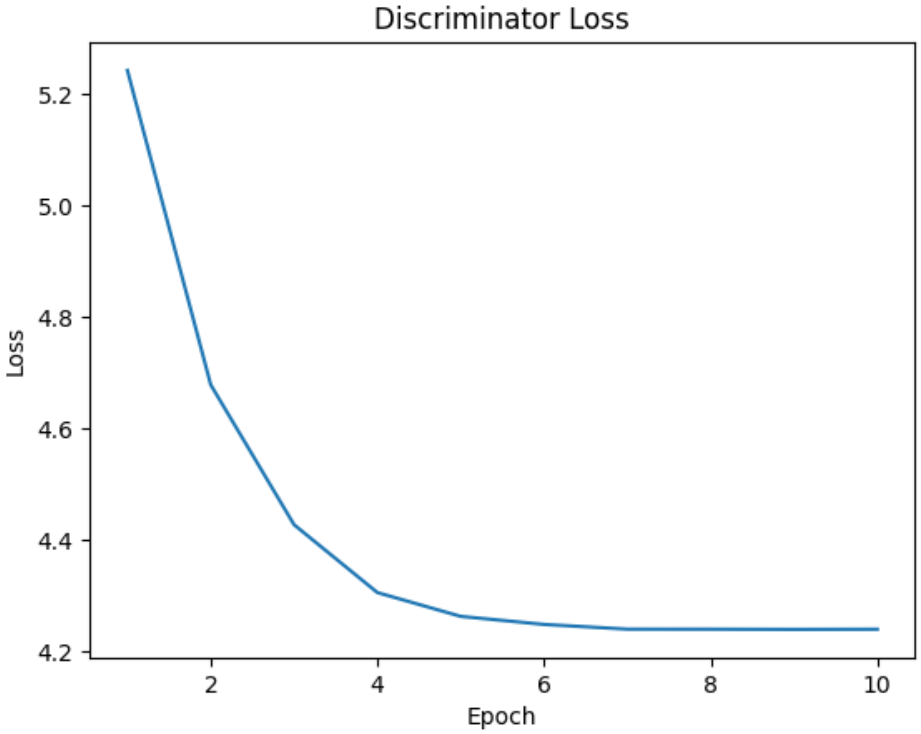

Figure 10: Discriminator loss after 10 epochs

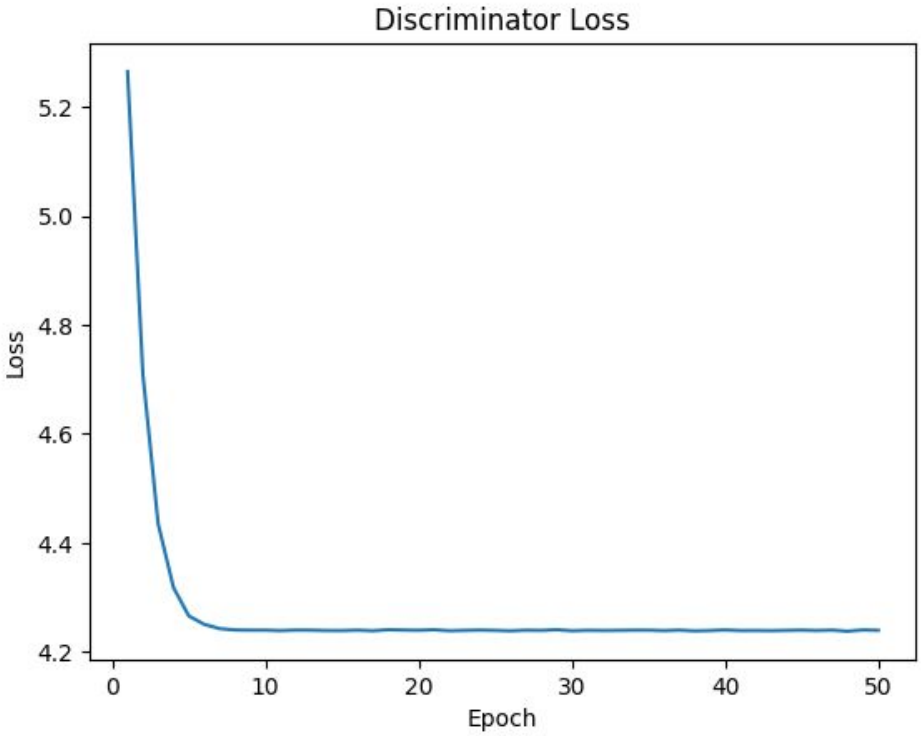

Figure 11: Discriminator loss after 50 epochs

We can observe a constant decrease in both the generator and discriminator loss over the training iterations. This indicates that our GAN model is effectively learning and improving its performance.

The downward trend in the loss values demonstrates the model's ability to generate high-quality colorized images that are difficult to distinguish from real images.

3.3 Results

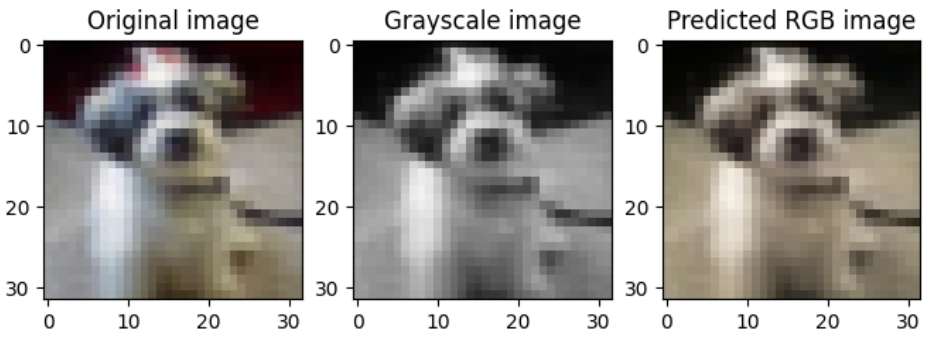

Figure 12: An image with its generated output

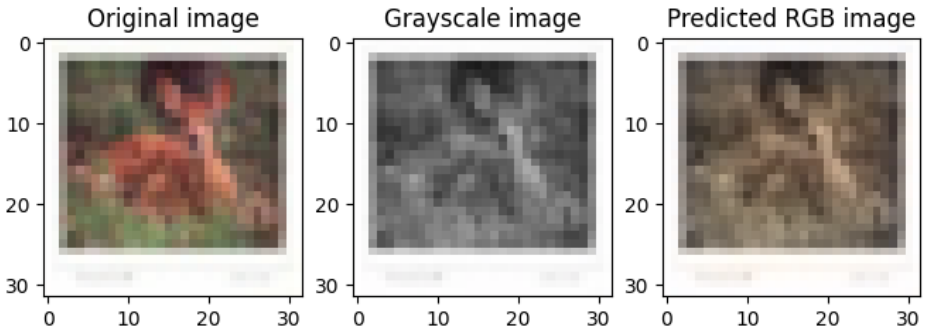

Figure 13: An image with its generated output

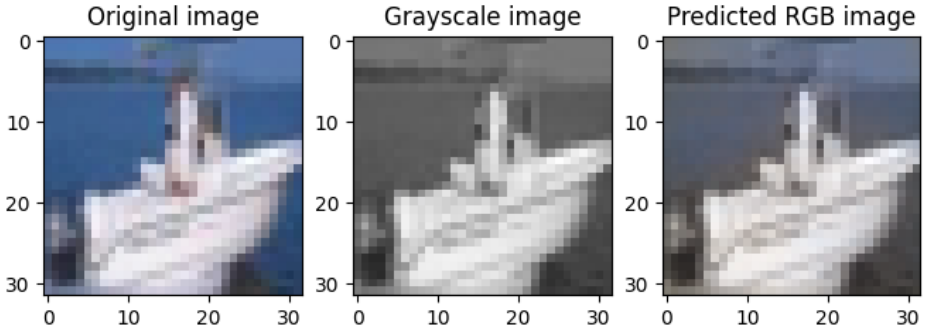

Figure 14: An image with its generated output

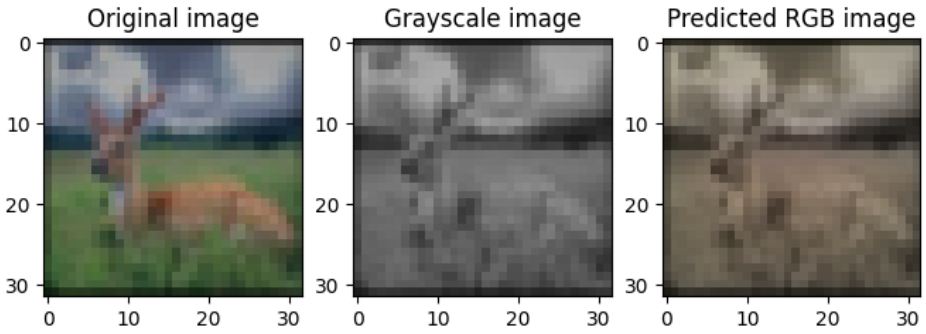

Figure 15: An image with its generated output

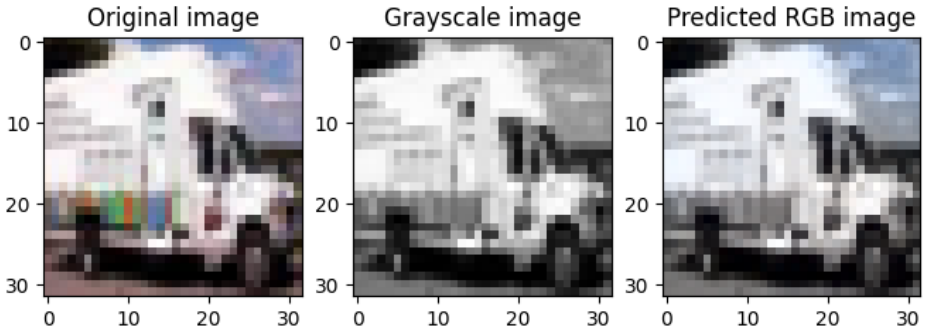

Figure 16: An image with its generated output

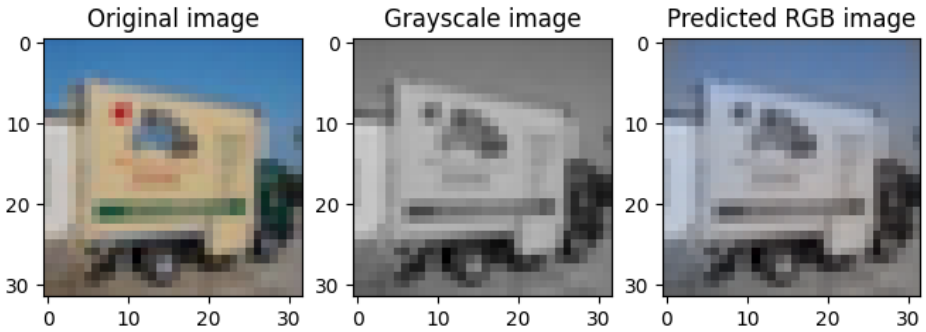

Figure 17: An image with its generated output

4 Architecture 2

A DCGAN consisting of a Generator and a Discriminator Model.
The Generator has the following 26 layers.

1. Input layer: 1 layer (Input shape: (img_size, img_size, 1))
2. Convolutional layers:
   - conv1: 6 layers (Conv2D + LeakyReLU)
   - conv2: 6 layers (Conv2D + LeakyReLU)
   - conv3: 6 layers (Conv2D + LeakyReLU)

3. Bottleneck layer:
    - bottleneck: 1 layer (Conv2D)
4. Upsampling layers:
    - conv_up_3: 3 layers (Conv2DTranspose + Activation)
    - conv_up_2: 3 layers (Conv2DTranspose + Activation)
    - conv_up_1: 3 layers (Conv2DTranspose + Activation)
5. Output layer: 1 layer (Conv2D)

```
def generator_model():
    inputs = tf.keras.layers.Input( shape=( img_size , img_size , 1 ) )

    conv1 = tf.keras.layers.Conv2D( 16 , kernel_size=( 5 , 5 ) , strides=1 )( inputs )
    conv1 = tf.keras.layers.LeakyReLU()( conv1 )
    conv1 = tf.keras.layers.Conv2D( 32 , kernel_size=( 3 , 3 ) , strides=1)( conv1 )
    conv1 = tf.keras.layers.LeakyReLU()( conv1 )
    conv1 = tf.keras.layers.Conv2D( 32 , kernel_size=( 3 , 3 ) , strides=1)( conv1 )
    conv1 = tf.keras.layers.LeakyReLU()( conv1 )

    conv2 = tf.keras.layers.Conv2D( 32 , kernel_size=( 5 , 5 ) , strides=1)( conv1 )
    conv2 = tf.keras.layers.LeakyReLU()( conv2 )
    conv2 = tf.keras.layers.Conv2D( 64 , kernel_size=( 3 , 3 ) , strides=1 )( conv2 )
    conv2 = tf.keras.layers.LeakyReLU()( conv2 )
    conv2 = tf.keras.layers.Conv2D( 64 , kernel_size=( 3 , 3 ) , strides=1 )( conv2 )
    conv2 = tf.keras.layers.LeakyReLU()( conv2 )

    conv3 = tf.keras.layers.Conv2D( 64 , kernel_size=( 5 , 5 ) , strides=1 )( conv2 )
    conv3 = tf.keras.layers.LeakyReLU()( conv3 )
    conv3 = tf.keras.layers.Conv2D( 128 , kernel_size=( 3 , 3 ) , strides=1 )( conv3 )
    conv3 = tf.keras.layers.LeakyReLU()( conv3 )
    conv3 = tf.keras.layers.Conv2D( 128 , kernel_size=( 3 , 3 ) , strides=1 )( conv3 )
    conv3 = tf.keras.layers.LeakyReLU()( conv3 )

    bottleneck = tf.keras.layers.Conv2D( 128 , kernel_size=( 3 , 3 ) , strides=1 , activation='tanh' , padding='same' )( conv3 )

    concat_1 = tf.keras.layers.Concatenate()( [ bottleneck , conv3 ] )
    conv_up_3 = tf.keras.layers.Conv2DTranspose( 128 , kernel_size=( 3 , 3 ) , strides=1 , activation='relu' )( concat_1 )
    conv_up_3 = tf.keras.layers.Conv2DTranspose( 128 , kernel_size=( 3 , 3 ) , strides=1 , activation='relu' )( conv_up_3 )
    conv_up_3 = tf.keras.layers.Conv2DTranspose( 64 , kernel_size=( 5 , 5 ) , strides=1 , activation='relu' )( conv_up_3 )

    concat_2 = tf.keras.layers.Concatenate()( [ conv_up_3 , conv2 ] )
    conv_up_2 = tf.keras.layers.Conv2DTranspose( 64 , kernel_size=( 3 , 3 ) , strides=1 , activation='relu' )( concat_2 )
    conv_up_2 = tf.keras.layers.Conv2DTranspose( 64 , kernel_size=( 3 , 3 ) , strides=1 , activation='relu' )( conv_up_2 )
    conv_up_2 = tf.keras.layers.Conv2DTranspose( 32 , kernel_size=( 5 , 5 ) , strides=1 , activation='relu' )( conv_up_2 )

    concat_3 = tf.keras.layers.Concatenate()( [ conv_up_2 , conv1 ] )
    conv_up_1 = tf.keras.layers.Conv2DTranspose( 32 , kernel_size=( 3 , 3 ) , strides=1 , activation='relu')( concat_3 )
    conv_up_1 = tf.keras.layers.Conv2DTranspose( 32 , kernel_size=( 3 , 3 ) , strides=1 , activation='relu')( conv_up_1 )
    conv_up_1 = tf.keras.layers.Conv2DTranspose( 3 , kernel_size=( 5 , 5 ) , strides=1 , activation='relu')( conv_up_1 )

    model = tf.keras.models.Model( inputs , conv_up_1 )
    return model
```

Figure 18: Generator Model Architecture

The discriminator model has 17 layers with the below layer wise breakdown: Input layer: 1 layer (Input shape: (32, 32, 3))

1. Convolutional layers:
    - Conv2D: 2 layers (16 filters, kernel size (5, 5), 'same' padding, ReLU activation)
    - Conv2D: 2 layers (32 filters, kernel size (5, 5), 'same' padding, ReLU activation)
    - MaxPooling2D: 2 layers (2x2 pool size, stride 2)
    - Conv2D: 2 layers (64 filters, kernel size (5, 5), 'same' padding, ReLU activation)
    - Conv2D: 2 layers (64 filters, kernel size (5, 5), 'same' padding, ReLU activation)
    - MaxPooling2D: 2 layers (2x2 pool size, stride 2)
    - Conv2D: 2 layers (128 filters, kernel size (3, 3), 'same' padding, ReLU activation)
    - Conv2D: 2 layers (128 filters, kernel size (3, 3), 'same' padding, ReLU activation)
    - MaxPooling2D: 2 layers (2x2 pool size, stride 2)

- Conv2D: 2 layers (256 filters, kernel size (3, 3), 'same' padding, ReLU activation)
- Conv2D: 2 layers (256 filters, kernel size (3, 3), 'same' padding, ReLU activation)
- MaxPooling2D: 2 layers (2x2 pool size, stride 2)

2. Flatten layer: 1 layer
3. Dense layers:
    - Dense: 1 layer (512 units, ReLU activation)
    - Dense: 1 layer (128 units, ReLU activation)
    - Dense: 1 layer (16 units, ReLU activation)
    - Dense: 1 layer (1 unit, sigmoid activation)

```
def discriminator_model():
    layers = [
        tf.keras.layers.Conv2D( 16 , kernel_size=( 5 , 5 ) , strides=1, padding='same', activation='relu' , input_shape=( 32 , 32 , 3 ) ),
        tf.keras.layers.Conv2D( 32 , kernel_size=( 5, 5 ) , strides=1, padding='same',activation='relu'  ),
        tf.keras.layers.MaxPooling2D(),
        tf.keras.layers.Conv2D( 64 , kernel_size=( 5 , 5 ) , strides=1, padding='same', activation='relu'  ),
        tf.keras.layers.Conv2D( 64 , kernel_size=( 5 , 5 ) , strides=1, padding='same', activation='relu'  ),
        tf.keras.layers.MaxPooling2D(),
        tf.keras.layers.Conv2D( 128 , kernel_size=( 3 , 3 ) , strides=1, padding='same', activation='relu'  ),
        tf.keras.layers.Conv2D( 128 , kernel_size=( 3 , 3 ) , strides=1, padding='same',activation='relu'  ),
        tf.keras.layers.MaxPooling2D(),
        tf.keras.layers.Conv2D( 256 , kernel_size=( 3 , 3 ) , strides=1, padding='same',activation='relu'  ),
        tf.keras.layers.Conv2D( 256 , kernel_size=( 3 , 3 ) , strides=1, padding='same',activation='relu'  ),
        tf.keras.layers.MaxPooling2D(),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense( 512, activation='relu'  )  ,
        tf.keras.layers.Dense( 128 , activation='relu' ) ,
        tf.keras.layers.Dense( 16 , activation='relu' ) ,
        tf.keras.layers.Dense( 1 , activation='sigmoid' )
    ]
    model = tf.keras.models.Sequential( layers )
    return model
```

Figure 19: Discriminator Model Architecture

4.1 Training

Both Generator and Discriminator models were trained simultaneously for 50 epochs. The time duration for the training along with validation evaluation took around 3minutes/epoch.

```
Epoch: 1 / 50
Training - Generator loss: 0.00944301 Discriminator loss: 0.40936387
Validation - Generator loss: 0.007789274267455999 Discriminator loss: 0.4193618840359627
Epoch: 2 / 50
Training - Generator loss: 0.008783165 Discriminator loss: 0.4128043
Validation - Generator loss: 0.006975069592528521 Discriminator loss: 0.41197196384693713
Epoch: 3 / 50
Training - Generator loss: 0.008417128 Discriminator loss: 0.4105218
Validation - Generator loss: 0.006693355264538463 Discriminator loss: 0.5109121828637224
Epoch: 4 / 50
Training - Generator loss: 0.00823066 Discriminator loss: 0.39912507
Validation - Generator loss: 0.006449931169404312 Discriminator loss: 0.41767077268438135
Epoch: 5 / 50
Training - Generator loss: 0.007976937 Discriminator loss: 0.4219334
Validation - Generator loss: 0.006302613769955141 Discriminator loss: 0.4193700300886276
Epoch: 6 / 50
Training - Generator loss: 0.00769397 Discriminator loss: 0.4061789
Validation - Generator loss: 0.006173519486997356 Discriminator loss: 0.4215333736957388
Epoch: 7 / 50
Training - Generator loss: 0.0073764734 Discriminator loss: 0.37811324
Validation - Generator loss: 0.006001569441658385 Discriminator loss: 0.42949992608516774
Epoch: 8 / 50
Training - Generator loss: 0.007382046 Discriminator loss: 0.41135824
Validation - Generator loss: 0.005905297415410267 Discriminator loss: 0.43139002640196616
Epoch: 9 / 50
...
Validation - Generator loss: 0.006121100406063364 Discriminator loss: 0.4311880951232098
Epoch: 50 / 50
Training - Generator loss: 0.004310263 Discriminator loss: 0.42816448
Validation - Generator loss: 0.006135719540984707 Discriminator loss: 0.4092469054333707
```

Figure 20: Training for 50 epochs

Both Generator and Discriminator losses for train and test set decreased steadily over time, as shown below.

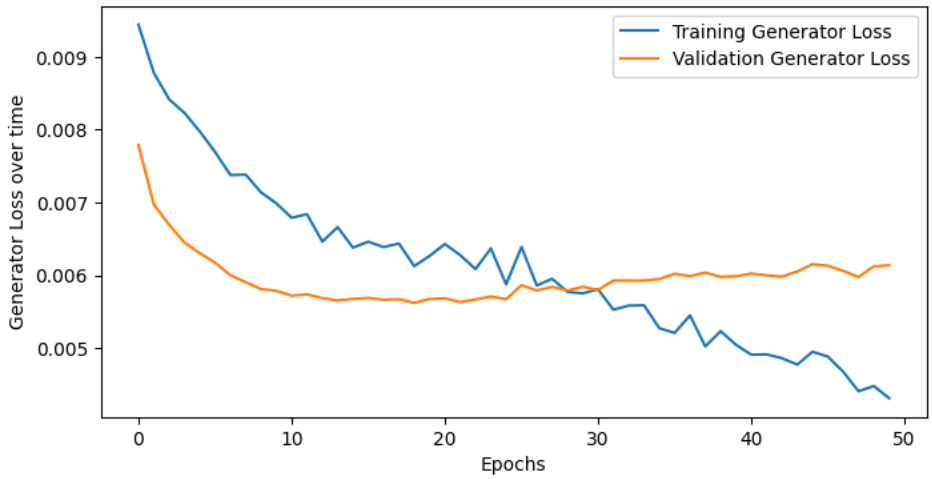

Figure 21: GAN 2 Generator Loss

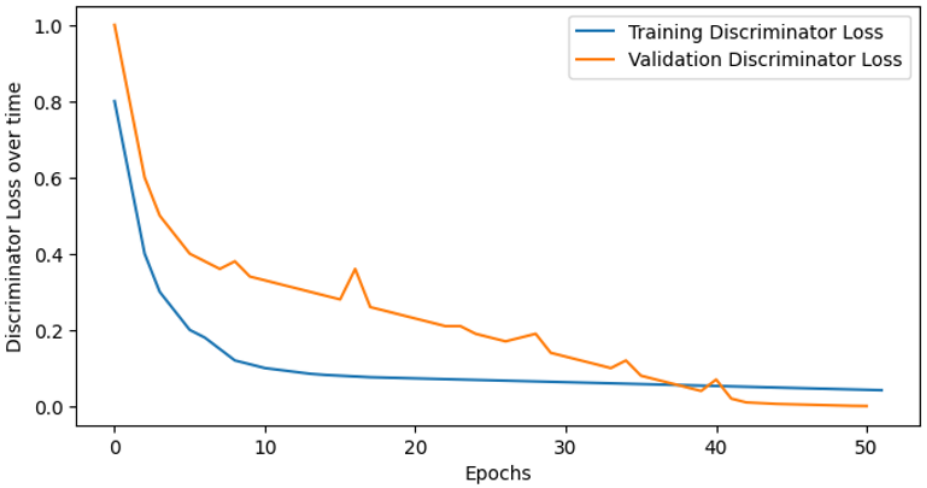

Figure 22: GAN 2 Discriminator Loss

4.2 Testing

For unseen test images, the trained generator model was used to generate coloured images. Below are few coloured result images with original and grayscale images to compare.

Figure 23: An image with its generated output

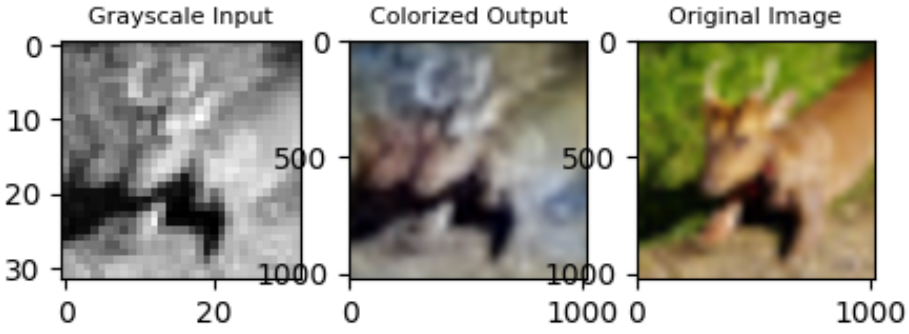

Figure 24: An image with its generated output

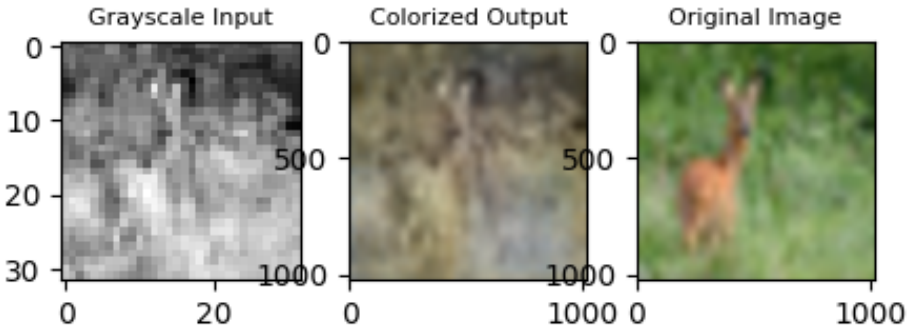

Figure 25: An image with its generated output

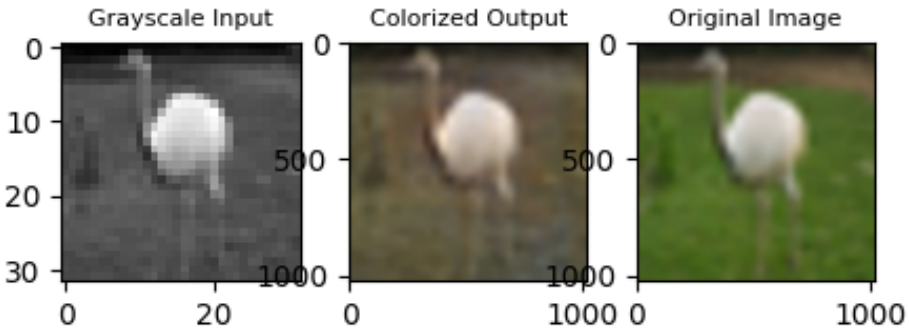

Figure 26: An image with its generated output

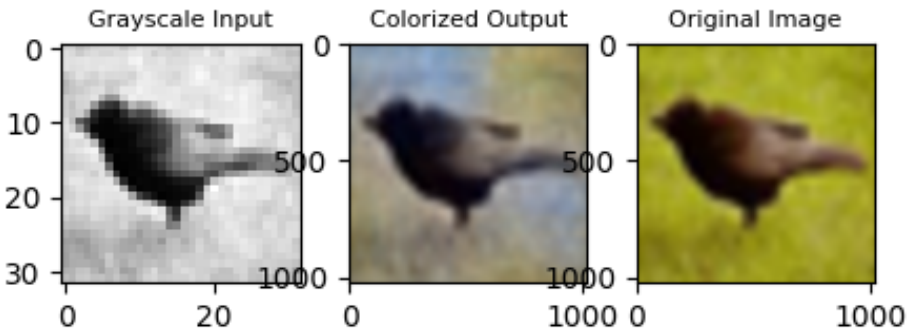

Figure 27: An image with its generated output

5 Architecture 3

An autoencoder with modified U-Net Architecture was used for image colorization.

```
def UNet():
    inputs = KL.Input(shape=[None, None, 1])
    conv1 = KL.Conv2D(64, (3, 3), padding='same')(inputs)
    conv1 = KL.BatchNormalization()(conv1)
    conv1 = KL.LeakyReLU(alpha=0.2)(conv1)
    conv1 = KL.Conv2D(64, (3, 3), strides=1, padding='same')(conv1)
    conv1 = KL.BatchNormalization()(conv1)
    conv1 = KL.LeakyReLU(alpha=0.2)(conv1)

    pool1 = KL.MaxPooling2D((2, 2), strides=2)(conv1)
    conv2 = KL.Conv2D(128, (3, 3), padding='same')(pool1)
    conv2 = KL.BatchNormalization()(conv2)
    conv2 = KL.LeakyReLU(alpha=0.2)(conv2)
    conv2 = KL.Conv2D(128, (3, 3), padding='same')(conv2)
    conv2 = KL.BatchNormalization()(conv2)
    conv2 = KL.LeakyReLU(alpha=0.2)(conv2)

    pool2 = KL.MaxPooling2D((2, 2), strides=2)(conv2)
    conv3 = KL.Conv2D(256, (3, 3), padding='same')(pool2)
    conv3 = KL.BatchNormalization()(conv3)
    conv3 = KL.LeakyReLU(alpha=0.2)(conv3)
    conv3 = KL.Conv2D(256, (3, 3), padding='same')(conv3)
    conv3 = KL.BatchNormalization()(conv3)
    conv3 = KL.LeakyReLU(alpha=0.2)(conv3)

    pool3 = KL.MaxPooling2D((2, 2), strides=2)(conv3)
    conv4 = KL.Conv2D(512, (3, 3), padding='same')(pool3)
    conv4 = KL.BatchNormalization()(conv4)
    conv4 = KL.LeakyReLU(alpha=0.2)(conv4)
    conv4 = KL.Conv2D(512, (3, 3), padding='same')(conv4)
    conv4 = KL.BatchNormalization()(conv4)
    conv4 = KL.LeakyReLU(alpha=0.2)(conv4)

    pool4 = KL.MaxPooling2D((2, 2), strides=2)(conv4)
    conv5 = KL.Conv2D(1024, (3, 3), padding='same')(pool4)
    conv5 = KL.BatchNormalization()(conv5)
    conv5 = KL.LeakyReLU(alpha=0.2)(conv5)
    conv5 = KL.Conv2D(1024, (3, 3), padding='same')(conv5)
    conv5 = KL.BatchNormalization()(conv5)
    conv5 = KL.LeakyReLU(alpha=0.2)(conv5)
```

Figure 28: U-Net Architecture

```
    up6 = KL.Conv2DTranspose(512, (2, 2), strides=2)(conv5)
    up6 = KL.Concatenate()([conv4, up6])
    conv6 = KL.Conv2D(512, (3, 3), padding='same')(up6)
    conv6 = KL.BatchNormalization()(conv6)
    conv6 = KL.Activation('relu')(conv6)
    conv6 = KL.Conv2D(512, (3, 3), padding='same')(conv6)
    conv6 = KL.BatchNormalization()(conv6)
    conv6 = KL.Activation('relu')(conv6)

    up7 = KL.Conv2DTranspose(256, (2, 2), strides=2)(conv6)
    up7 = KL.Concatenate()([conv3, up7])
    conv7 = KL.Conv2D(256, (3, 3), padding='same')(up7)
    conv7 = KL.BatchNormalization()(conv7)
    conv7 = KL.Activation('relu')(conv7)
    conv7 = KL.Conv2D(256, (3, 3), padding='same')(conv7)
    conv7 = KL.BatchNormalization()(conv7)
    conv7 = KL.Activation('relu')(conv7)

    up8 = KL.Conv2DTranspose(128, (2, 2), strides=2)(conv7)
    up8 = KL.Concatenate()([conv2, up8])
    conv8 = KL.Conv2D(128, (3, 3), padding='same')(up8)
    conv8 = KL.BatchNormalization()(conv8)
    conv8 = KL.Activation('relu')(conv8)
    conv8 = KL.Conv2D(128, (3, 3), padding='same')(conv8)
    conv8 = KL.BatchNormalization()(conv8)
    conv8 = KL.Activation('relu')(conv8)

    up9 = KL.Conv2DTranspose(64, (2, 2), strides=2)(conv8)
    up9 = KL.Concatenate()([conv1, up9])
    conv9 = KL.Conv2D(64, (3, 3), padding='same')(up9)
    conv9 = KL.BatchNormalization()(conv9)
    conv9 = KL.Activation('relu')(conv9)
    conv9 = KL.Conv2D(64, (3, 3), padding='same')(conv9)
    conv9 = KL.BatchNormalization()(conv9)
    conv9 = KL.Activation('relu')(conv9)

    outputs = KL.Conv2D(3, (1, 1), strides=1)(conv9)
    model = Model(inputs=inputs, outputs=outputs)
    return model
```

Figure 29: U-Net Architecture

5.1 Training

We ran the model for 100 epochs, carefully monitoring the losses and accuracy at each iteration. Additionally, we performed periodic comparisons between the generated colorized images and the original ground truth images after every 10 epochs. This allowed us to evaluate the model's progress and assess the quality of the colorization results over time.

During the training, we observed a continuous decrease in both the loss values and the Mean Absolute Error (MAE) values. This reduction in losses indicates that our model consistently improved its ability to generate colorized images that closely resembled the original ground truth images. The decreasing MAE values further validated the model's accuracy in reproducing the color details with minimal discrepancies.

```
Epoch: 1 / 100
125/125 [==============================] - 6s 52ms/step - loss: 0.0434 - mae: 0.1083 - acc: 0.3725 - val_loss: 0.0697 - val_mae: 0.2116 - val_acc: 0.4171
Epoch: 2 / 100
125/125 [==============================] - 5s 43ms/step - loss: 0.0071 - mae: 0.0604 - acc: 0.4052 - val_loss: 0.0452 - val_mae: 0.1734 - val_acc: 0.3815
Epoch: 3 / 100
125/125 [==============================] - 5s 43ms/step - loss: 0.0065 - mae: 0.0567 - acc: 0.4172 - val_loss: 0.0294 - val_mae: 0.1408 - val_acc: 0.4241
Epoch: 4 / 100
125/125 [==============================] - 5s 43ms/step - loss: 0.0063 - mae: 0.0551 - acc: 0.4256 - val_loss: 0.0148 - val_mae: 0.0991 - val_acc: 0.4243
Epoch: 5 / 100
125/125 [==============================] - 5s 42ms/step - loss: 0.0062 - mae: 0.0549 - acc: 0.4302 - val_loss: 0.0072 - val_mae: 0.0639 - val_acc: 0.4529
Epoch: 6 / 100
125/125 [==============================] - 5s 43ms/step - loss: 0.0060 - mae: 0.0538 - acc: 0.4399 - val_loss: 0.0057 - val_mae: 0.0531 - val_acc: 0.4316
Epoch: 7 / 100
125/125 [==============================] - 5s 43ms/step - loss: 0.0058 - mae: 0.0526 - acc: 0.4615 - val_loss: 0.0054 - val_mae: 0.0502 - val_acc: 0.4714
Epoch: 8 / 100
125/125 [==============================] - 5s 42ms/step - loss: 0.0056 - mae: 0.0518 - acc: 0.4741 - val_loss: 0.0054 - val_mae: 0.0508 - val_acc: 0.4698
Epoch: 9 / 100
125/125 [==============================] - 5s 43ms/step - loss: 0.0054 - mae: 0.0507 - acc: 0.4847 - val_loss: 0.0053 - val_mae: 0.0498 - val_acc: 0.5137
Epoch: 10 / 100
125/125 [==============================] - 5s 42ms/step - loss: 0.0052 - mae: 0.0494 - acc: 0.4957 - val_loss: 0.0052 - val_mae: 0.0492 - val_acc: 0.4872
1/1 [==============================] - 1s 783ms/step
```

Figure 30: Training U-Net for 100 epochs

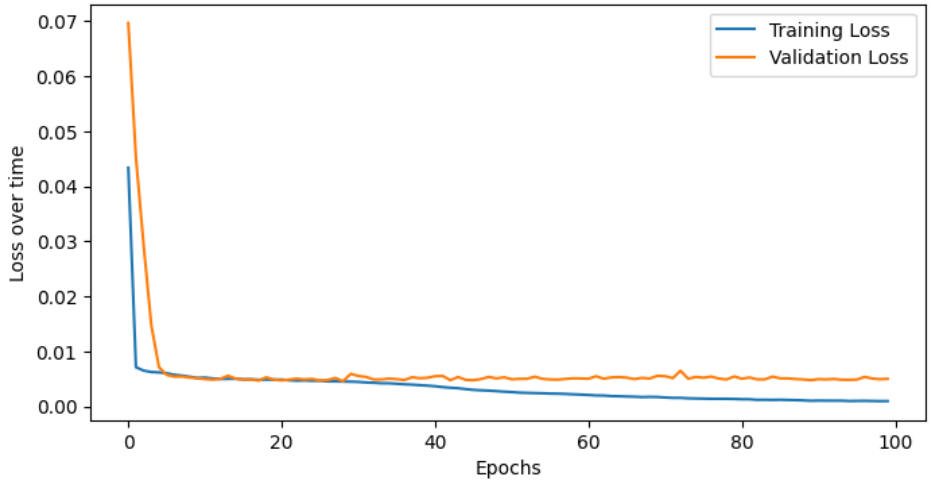

Figure 31: U-Net Loss over time

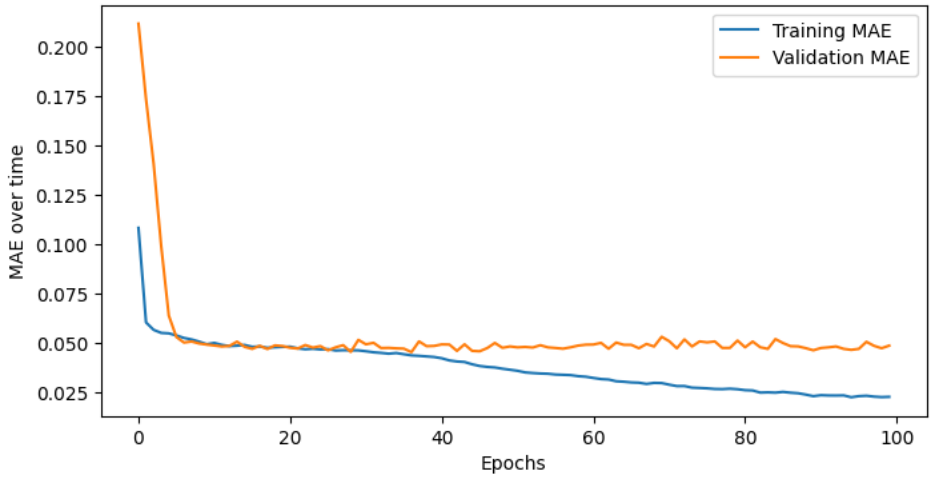

Figure 32: U-Net MAE over time

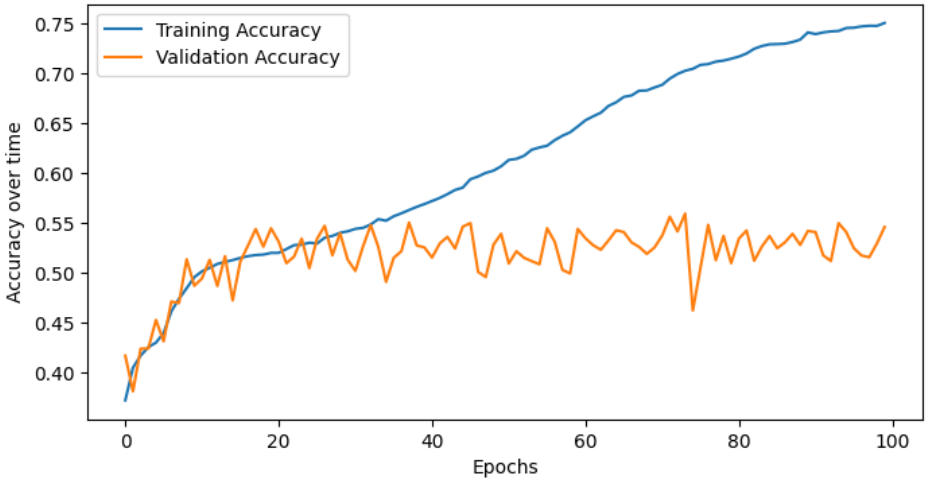

Figure 33: U-Net Accuracy over time

5.2 Results

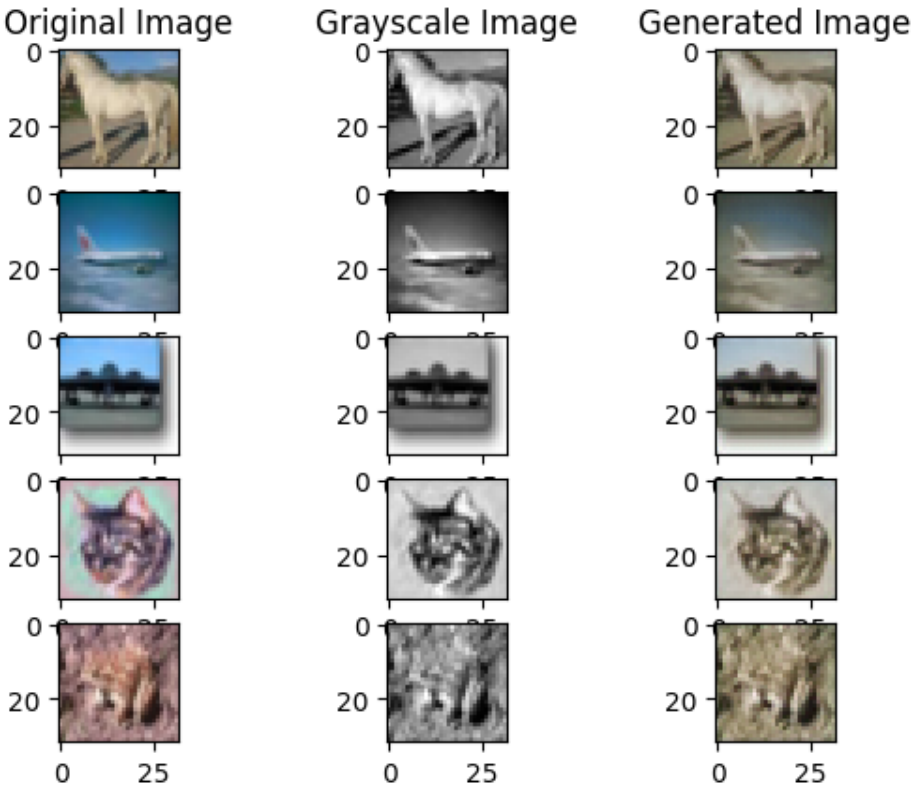

Figure 34: Results after 10 epochs

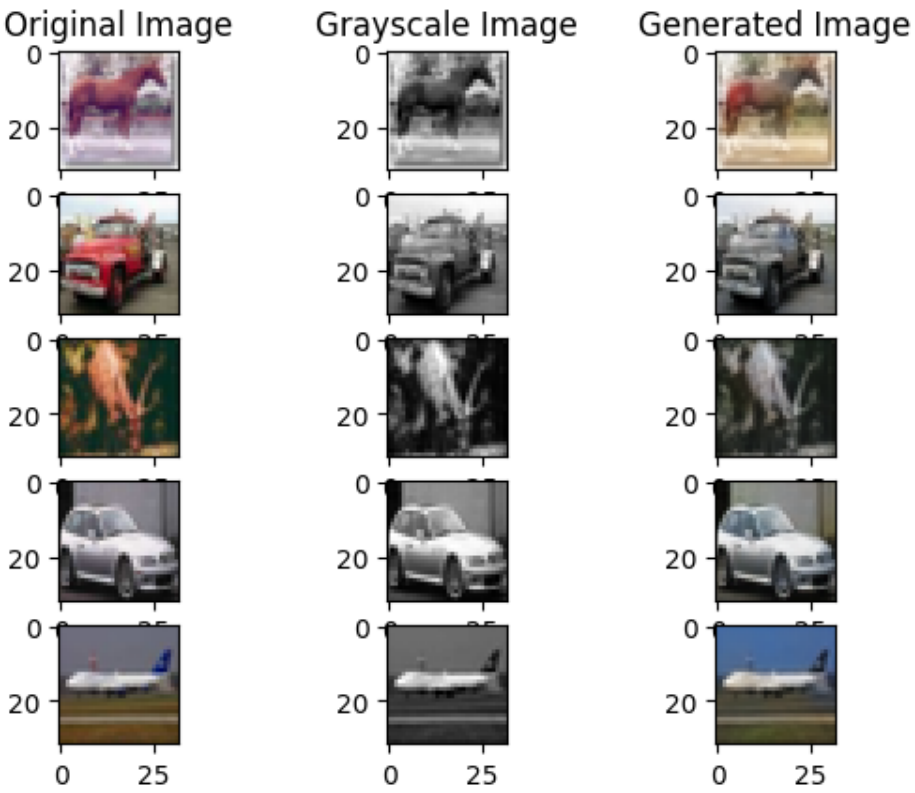

Figure 35: Results after 100 epochs

After training the U-Net model for 100 epochs, performance was evaluated on validation samples. The results revealed an accuracy of approximately 75%. This accuracy metric indicates the model's ability to correctly predict the colorization of the validation images, demonstrating its proficiency in capturing the essential color details and reproducing them accurately. These results validate the effectiveness of our trained U-Net model in generating high-quality colorized outputs during the training process.

6 Project Management Tracker

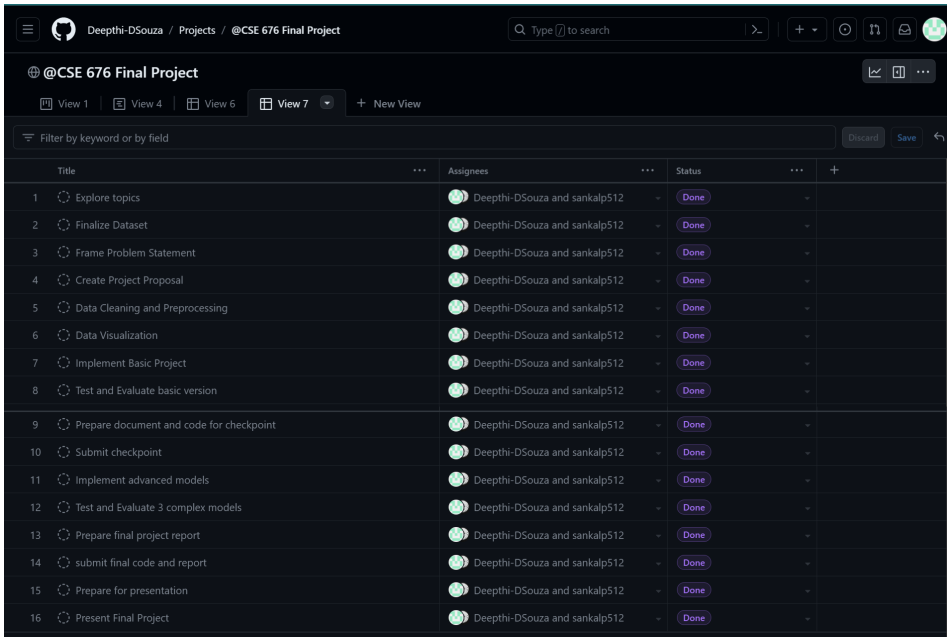

Figure 36: Snapshot of the Project Management Tracker

7 Contribution

Team Member	Project Part	Contribution (%)
Sankalp	Model 1 and 2	50
Deepthi	Model 2 and 3	50

8 References

https://pyimagesearch.com/2022/02/21/u-net-image-segmentation-in-keras/
https://www.tensorflow.org/tutorials/generative/dcgan
https://pub.towardsai.net/how-to-make-a-gan-to-generate-color-images-33d29f8a79c8
http://cs231n.stanford.edu/reports/2017/pdfs/302.pdf
https://towardsdatascience.com/colorizing-black-white-images-with-u-net-and-conditional-gan-a-tutorial-81b2df111cd8