Before we start...

The submission includes three notebooks: WGAN, GAN-L1, and GAN-MSE. Initially, we focused on the WGAN model, dedicating significant time to its development. However, after extensive training, we determined that it did not meet our expectations, prompting us to explore other GAN models.

Ultimately, we chose the **GAN-L1** model as the selected model due to its superior performance. We also discuss the GAN-MSE model in the report for comparative purposes. Additionally, we provide an overview of our initial experiments with the WGAN model to give context to our project's workflow.

In summary, although there are three notebooks in the submission, the GAN-L1 model is the selected model for the tests.

Colorizing Grayscale Images Using Generative Adversarial Networks

Roie Shahar 314985300
Meitar Yeruham212176127

Introduction

Our project focuses on colorizing grayscale images. To achieve this, we experimented with training different U-Net models using Generative Adversarial Networks (GANs). We trained each model with our dataset to see how well they could add color to the images.

Dataset: Flower-102, by Maria- Elena Nilsback and Andrew Zisserman .

Data Management for Image Colorization

Data Provider Class

-Manages image datasets for the model, pairing color and corresponding grayscale images.

Image Preprocessing

-Resizes all images to 256x256 pixels.

-Converts color images to grayscale using PyTorch's transforms.

```
transform_Grayscale = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.Grayscale(),
    transforms.ToTensor()
])

transform = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.ToTensor()
])
```

Split dataset

- We divided the dataset into three distinct subsets: training, validation, and testing

```
train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True, num_workers=num_workers)
test_loader = DataLoader(test_set, batch_size=batch_size, shuffle=True, num_workers=num_workers)
validation_loader = DataLoader(validation_set, batch_size=batch_size, shuffle=True, num_workers=num_workers)
```

Architecture

The architecture implemented in this model is a GAN (Generative Adversarial Network) utilizing U-Net as the generator and a CNN for the discriminator. Both networks extensively employ VGG blocks to construct deep convolutional layers, optimized using the Adam optimizer.

Generator - U-Net Architecture:

- **Purpose :**The U-Net generator is designed for effective image transformation tasks where context and localization are crucial, such as in image coloring or segmentation.
- **Structure :**Each layer of the U-Net generator consists of a VGGBlock, which is a series of two convolutional layers, each followed by batch normalization and ReLU activation. We use them because they are known for their simplicity and effectiveness in feature extraction. The network employs downsampling (encoder) followed by upsampling (decoder) with skip connections to preserve context and details.

```python
class VGGBlock(nn.Module):
    def __init__(self, in_channels, out_channels, dropout_rate=0.0):
        super(VGGBlock, self).__init__()
        layers = [
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        ]

        if dropout_rate > 0.0:
            layers.append(nn.Dropout(dropout_rate))

        self.conv = nn.Sequential(*layers)

    def forward(self, x):
        return self.conv(x)
```

UNet Generator Architecture

- **Input Layer :**Accepts 1-channel grayscale images, starting the processing pipeline for image transformation.
- **Downsampling Path Layers :**Utilizes sequential VGG blocks, increasing in complexity from 64 to 1024 channels, designed to capture progressively deeper features.
- **Components :**Each block contains two convolutional layers followed by batch normalization and ReLU activation, extracting and normalizing feature representations.
- **Dimension Reduction :**Max pooling is applied after each block to reduce spatial dimensions, concentrating feature details.
- **Upsampling Path Layers :**Mirrors the downsampling structure but in reverse, gradually restoring the image's dimensions.
- **Feature Merging**:Each layer includes upsampling and convolution, followed by concatenation with the corresponding downsampling layer to retain critical spatial hierarchies.
- **Output Layer :**A final convolutional layer that converts the processed features into a 3-channel color image.

```
# Pooling and upsampling
self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)
self.upsample = nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True)

# Encoder: Downsampling path
self.encoder_block1 = VGGBlock(1, 64)
self.encoder_block2 = VGGBlock(64, 128)
self.encoder_block3 = VGGBlock(128, 256)
self.encoder_block4 = VGGBlock(256, 512)
self.encoder_block5 = VGGBlock(512, 512)
self.encoder_block6 = VGGBlock(512, 1024)

# Decoder: Upsampling path
self.decoder_block5 = VGGBlock(512 + 1024, 512)
self.decoder_block4 = VGGBlock(512 + 512, 256)
self.decoder_block3 = VGGBlock(256 + 256, 256)
self.decoder_block2 = VGGBlock(128 + 256, 128)
self.decoder_block1 = VGGBlock(128 + 64, 64)

# Final convolution
self.conv_last = nn.Conv2d(64, 3, kernel_size=1)
```

```
def forward(self, x):
    # Downsample
    conv1 = self.encoder_block1(x)
    conv2 = self.encoder_block2(self.maxpool(conv1))
    conv3 = self.encoder_block3(self.maxpool(conv2))
    conv4 = self.encoder_block4(self.maxpool(conv3))
    conv5 = self.encoder_block5(self.maxpool(conv4))
    x = self.encoder_block6(self.maxpool(conv5))

    # Upsample and concatenate
    x = torch.cat([self.upsample(x), conv5], dim=1)
    x = torch.cat([self.upsample(self.decoder_block5(x)), conv4], dim=1)
    x = torch.cat([self.upsample(self.decoder_block4(x)), conv3], dim=1)
    x = torch.cat([self.upsample(self.decoder_block3(x)), conv2], dim=1)
    x = torch.cat([self.upsample(self.decoder_block2(x)), conv1], dim=1)

    # Final convolution
    out = self.conv_last(self.decoder_block1(x))

    return out
```

1x256x256

conv 3x3
batchNorm+ReLU
64x256x256

conv 3x3
batchNorm+ReLU
64x256x256

maxpool 2x2
64x128x128

(192x256x256(

conv 3x3
batchNorm+ReLU
128x128x128

conv 3x3
batchNorm+ReLU
128x128x128

maxpool 2x2
128x64x64

(384x128x128(

conv 3x3
batchNorm+ReLU
256x64x64

conv 3x3
batchNorm+ReLU
256x64x64

maxpool 2x2
256x32x32

(512x64x64(

conv 3x3
batchNorm+ReLU
512x32x32

conv 3x3
batchNorm+ReLU
512x32x32

maxpool 2x2
512x16x16

(1024x32x32(

conv 3x3
batchNorm+ReLU
512x16x16

conv 3x3
batchNorm+ReLU
512x16x16

maxpool 2x2
512x8x8

(1536x16x16(

conv 3x3
batchNorm+ReLU
1024x8x8

conv 3x3
batchNorm+ReLU
1024x8x8

conv 3x3
batchNorm+ReLU
512x16x16

conv 3x3
batchNorm+ReLU
512x16x16

upconv 2x2
1024x16x16

conv 3x3
batchNorm+ReLU
256x32x32

conv 3x3
batchNorm+ReLU
256x32x32

upconv 2x2
512x32x32

conv 3x3
batchNorm+ReLU
256x64x64

conv 3x3
batchNorm+ReLU
256x64x64

upconv 2x2
256x64x64

conv 3x3
batchNorm+ReLU
128x128x128

conv 3x3
batchNorm+ReLU
128x128x128

upconv 2x2
256x128x128

conv 3x3
batchNorm+ReLU
64x256x256

conv 3x3
batchNorm+ReLU
64x256x256

upconv 2x2
128x256x256

3x256x256

Discriminator Architecture -CNN

- **Layers :**Sequential VGG blocks with Instance Normalization and LeakyReLU, increasing in depth from 64 to 1024 channels.Function: Each block contains two convolutional layers with ReLU and batch normalization to stabilize learning.
- **Final Layer :**A convolution that outputs a single value to determine real or fake.

```python
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        self.layer1 = VGG_block(3, 64)
        self.layer2 = VGG_block(64, 128)
        self.layer3 = VGG_block(128, 256)
        self.layer4 = VGG_block(256, 512)
        self.layer5 = VGG_block(512, 1024)
        self.layer6 = VGG_block(1024, 1024)

        # Final convolutional layer
        self.final_conv = nn.Sequential(
            nn.Conv2d(1024, 1, kernel_size=3, stride=1, padding=1, bias=False)
        )
```

```python
def forward(self, img):
    x1 = self.layer1(img)
    x2 = self.layer2(x1)
    x3 = self.layer3(x2)
    x4 = self.layer4(x3)
    x5 = self.layer5(x4)
    x6 = self.layer6(x5)
    output = self.final_conv(x6)
    return torch.sigmoid(output)
```

Optimizer Selection

- **Optimizer Used :**All UNet models utilize the **Adam optimizer**. This choice is based on its proven effectiveness in handling the complexities of image data, as it adapts the learning rates for each parameter, facilitating faster and more stable training.

```
# Generator
optimizer_G = optim.Adam(generator.parameters(), lr=0.0002, betas=(0.5, 0.999))

# Discriminator
optimizer_D = optim.Adam(discriminator.parameters(), lr=0.0002, betas=(0.5, 0.999))
```

Training

In this training loop, both the generator and discriminator of a GAN are trained iteratively over a set number of epochs (350). During each epoch (4 batches - 255 images each one), the discriminator is trained first by alternating between real and fake images, where it learns to distinguish genuine images from those generated by the generator. The generator is then trained to fool the discriminator by minimizing a combined loss, which includes an adversarial loss that makes the generated images more realistic and an loss function that reduces the pixel-wise difference between the generated and real images.

GAN + MSE

- The Generator Loss was a linear combination of an MSE Loss and a GAN loss which was calculated using the discriminator.

```
adversarial_loss = nn.BCEWithLogitsLoss()
mse_loss = nn.MSELoss()
```

```
lambda_adv = 0.01
lambda_mse = 0.99
```

```
 # Combine adversarial loss and MSE loss
generator_loss =  lambda_adv * generator_loss_adv + lambda_mse * generator_loss_mse
```

Validation & Testing

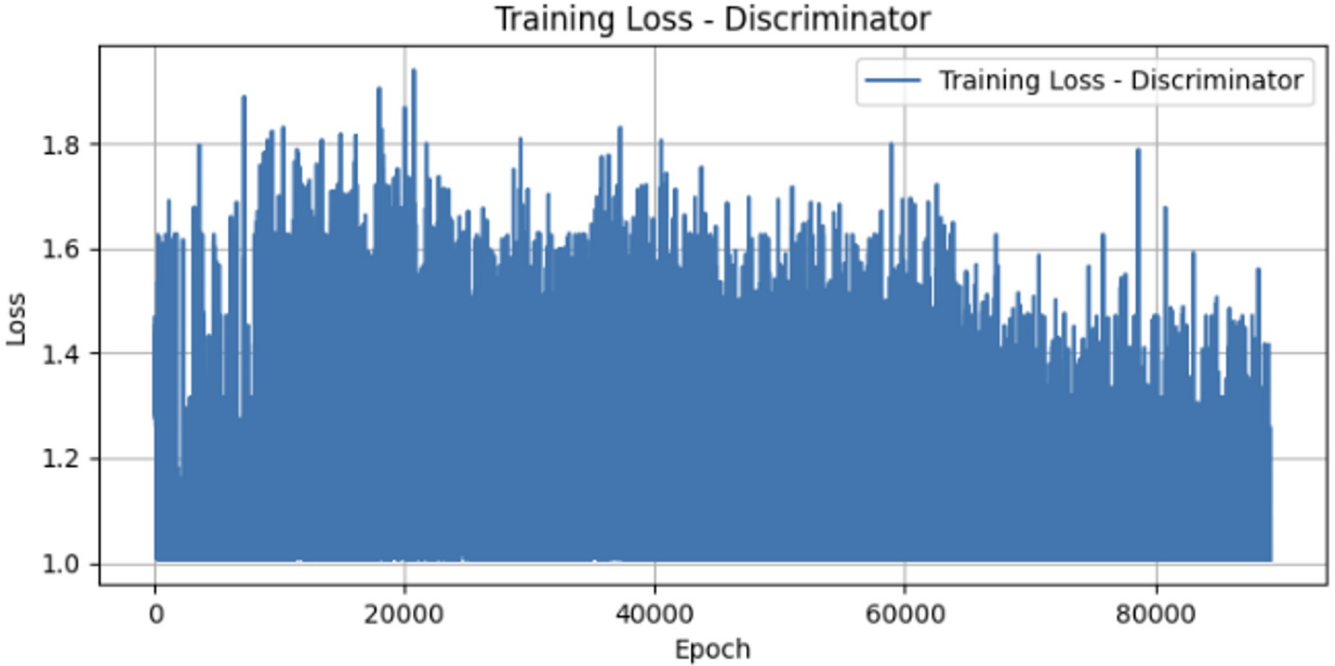
for Validation we create a function:

validate function sets the generator to evaluation mode and measures its performance on a validation dataset by computing metrics such as MSE loss, PSNR, SSIM, and MAE for each batch. It returns the average values of these metrics across the entire validation dataset, providing insights into the model's generalization ability and image quality reproduction.

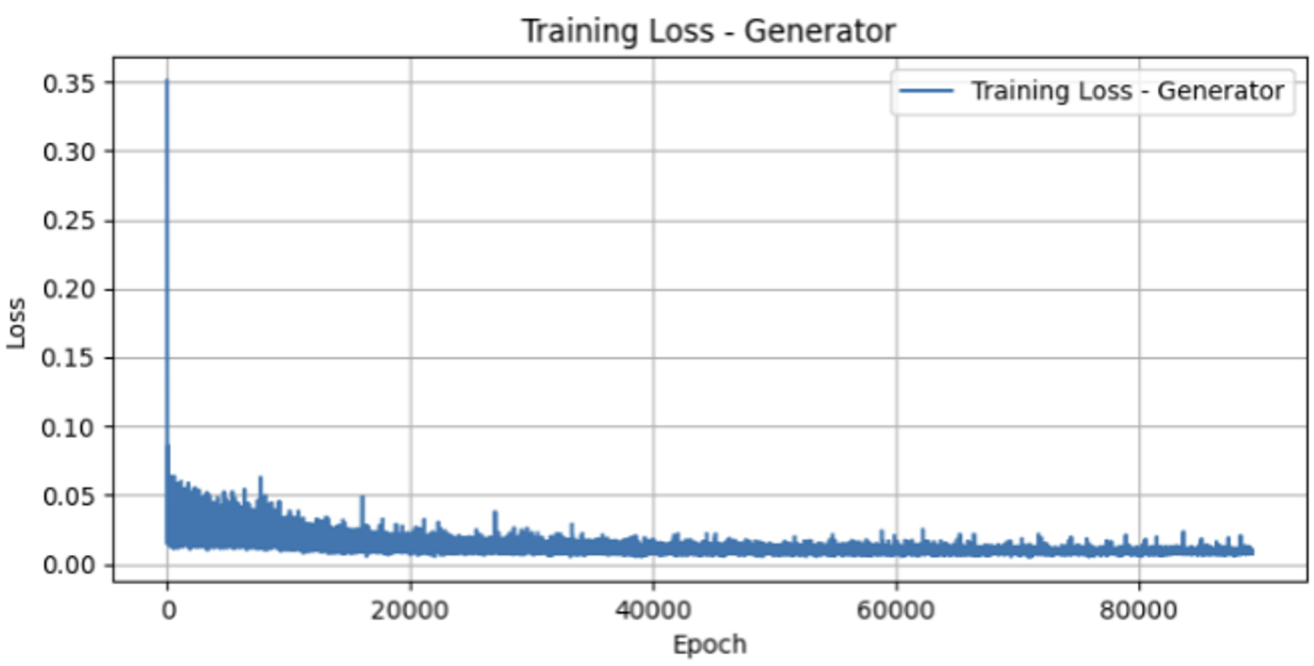
```python
def validate(generator, criterion, validation_loader):
    generator.eval()  # Set the generator to evaluation mode
    total_loss = 0
    total_psnr = 0
    total_mae = 0
    total_ssim = 0

    with torch.no_grad():
        for real_image, grey_image in validation_loader:
            grey_image = grey_image.to(device)
            real_image = real_image.to(device)
            # Generate fake images
            fake_image = generator(grey_image)

            # Compute MSE loss
            loss = criterion(fake_image, real_image)
            # Accumulate total loss
            total_loss += loss.item()

            # Compute PSNR
            psnr_value = psnr(real_image, fake_image)
            total_psnr += psnr_value.item()

            # ssim
            ssim_value = ssim(real_image, fake_image)
            total_ssim += ssim_value.mean().item()

            # Compute MAE
            mae_value = F.l1_loss(fake_image, real_image)
            total_mae += mae_value.item()

    # Calculate average validation loss and other metrics
    avg_loss = total_loss / len(validation_loader)
    avg_psnr = total_psnr / len(validation_loader)
    avg_ssim = total_ssim / len(validation_loader)
    avg_mae = total_mae / len(validation_loader)

    return avg_loss, avg_psnr, avg_ssim, avg_mae
```

GRAPHS RESULT GAN+MSE

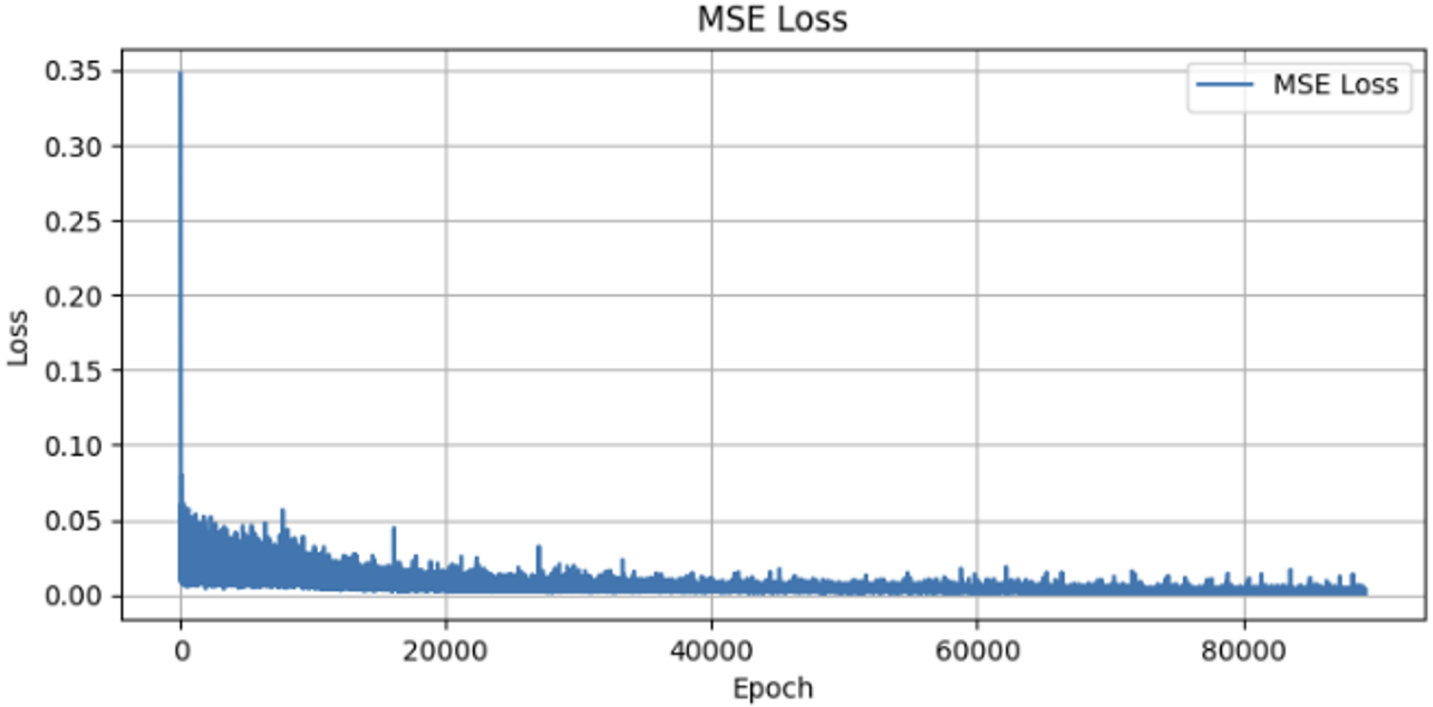
The graph displays the training loss for the discriminator in a GAN over epochs, showing fluctuations but a general decrease in loss, indicating learning and optimization progress over time.



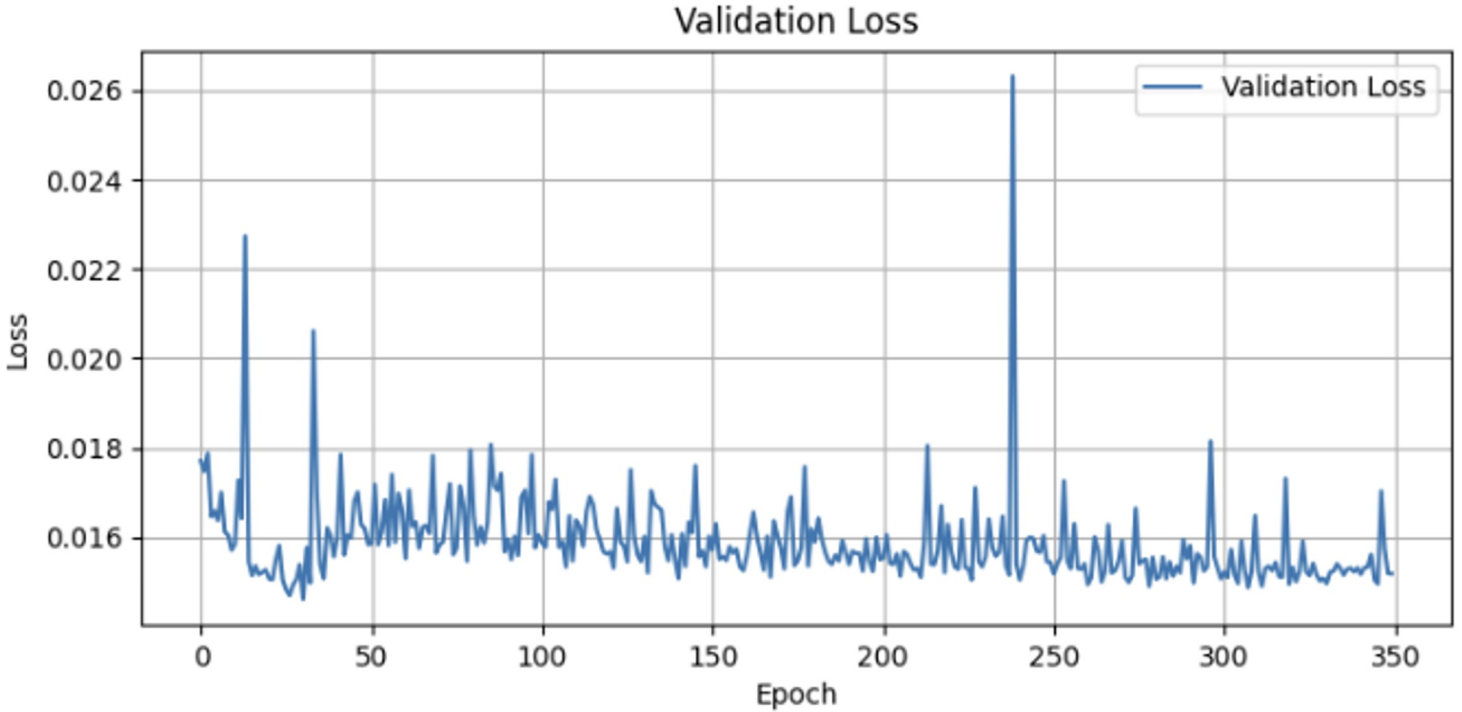
The training loss for the generator- it decreases sharply at first, indicating rapid learning, and then stabilizes, demonstrating consistent performance as training progresses.



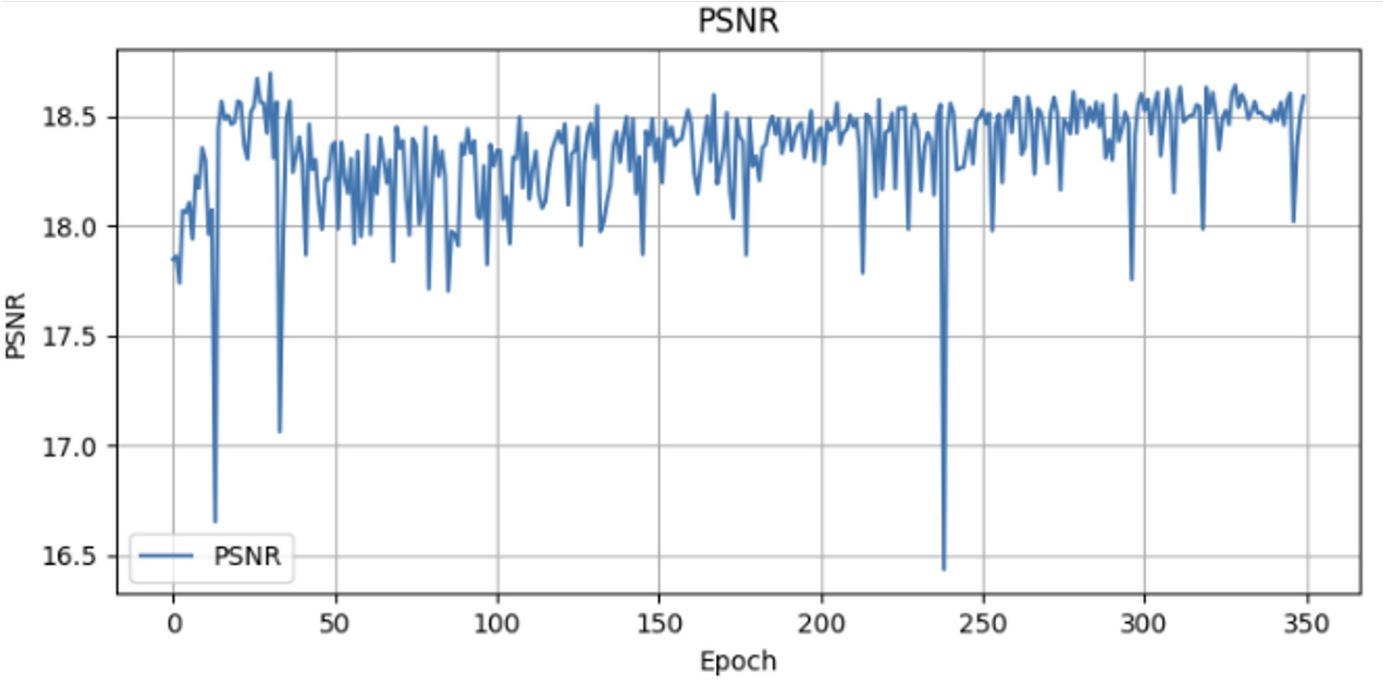
The MSE loss graph displays a rapid initial decline and reaches a plateau quickly, indicating the model's loss has stabilized and is converging as training continues.



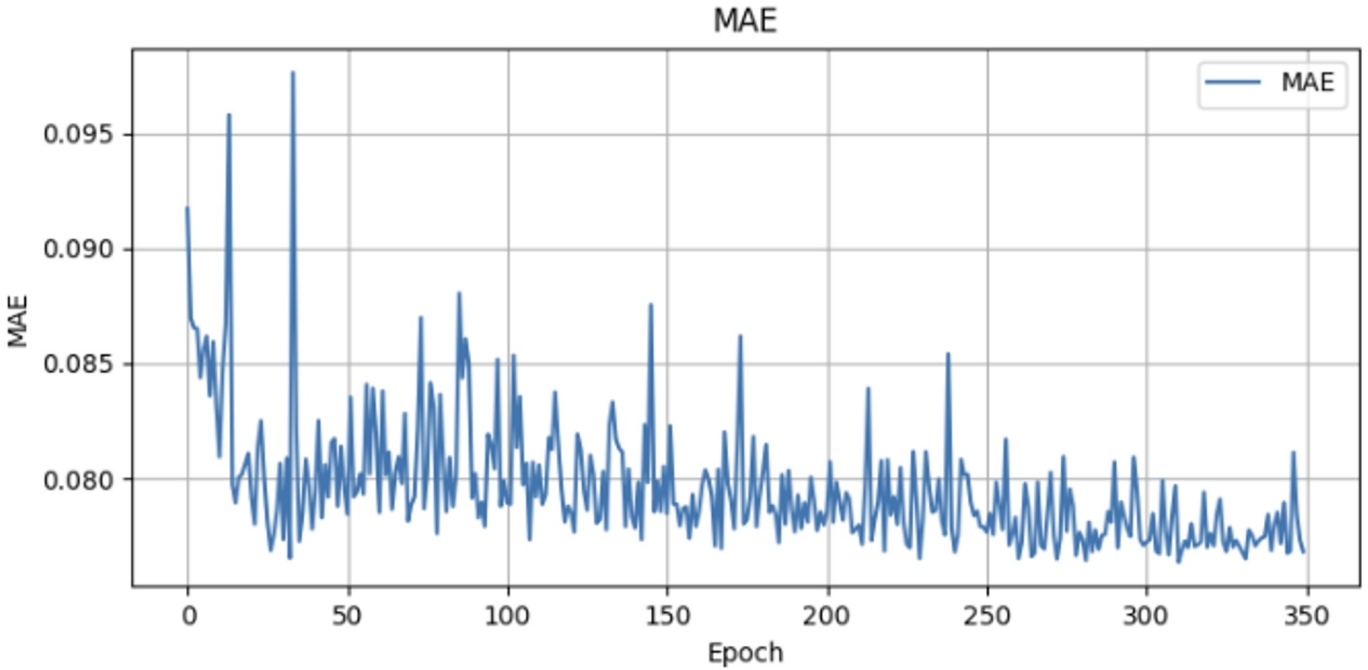
The validation loss chart indicates a modest but consistent decrease, punctuated by occasional spikes, reflecting a stable yet imperfect learning trajectory.



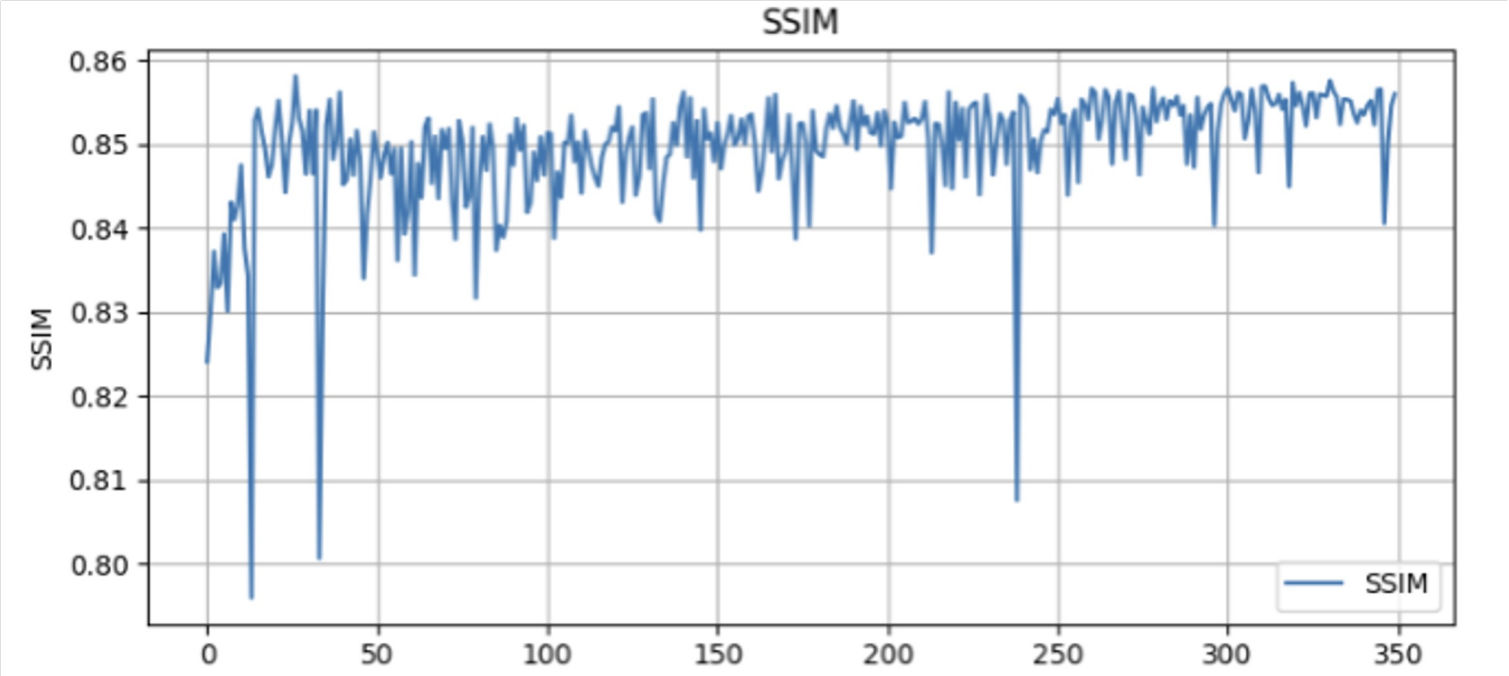
The PSNR graph shows slight variations within a narrow range, indicating that image quality, as measured by peak signal-to-noise ratio, is relatively stable throughout the epochs.



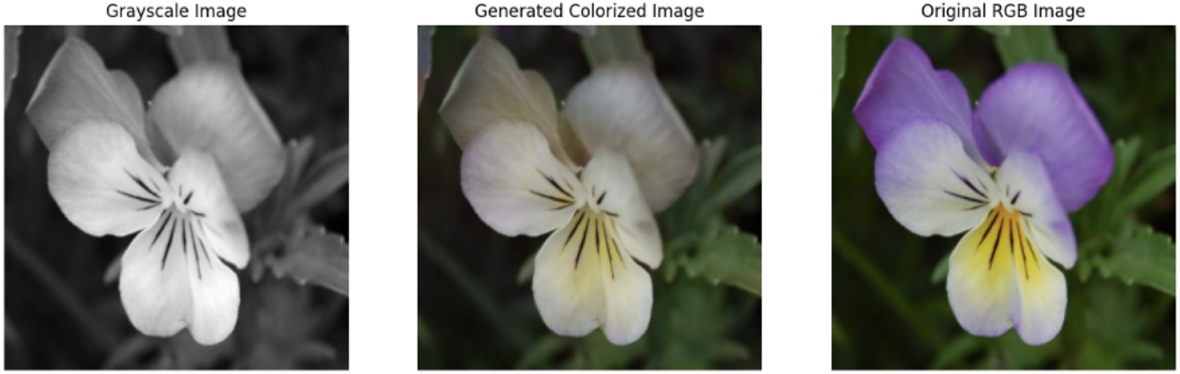
The MAE graph shows a decrease and then levels out, indicating the model's predictions are gradually aligning closer to the actual values as training progresses.



The SSIM graph shows minor fluctuations, indicating a stable model performance in preserving image structures over the training epochs with slight adjustments.



Test Results GAN+MSE



Grayscale Image

Generated Colorized Image

Original RGB Image

Grayscale Image

Generated Colorized Image

Original RGB Image

Grayscale Image

Generated Colorized Image

Original RGB Image

Grayscale Image

Generated Colorized Image

Original RGB Image

Grayscale Image

Generated Colorized Image

Original RGB Image

Grayscale Image

Generated Colorized Image

Original RGB Image

Transformation to L1

We were not satisfied with the initial results, so we searched the internet for ways to improve them. We found that using the L1 loss function instead of MSE can lead to better outcomes because it makes the images sharper and clearer (because of Edge Preservation).

GAN + L1

- The Generator Loss was a linear combination of an L1 Loss and a GAN loss which was calculated using the discriminator.
- In each epoch we trained the Generator and the discriminator.
- The L1 loss (also known as Mean Absolute Error, MAE) in images measures the average of the absolute differences between predicted and actual pixel values: $\text{L1 Loss} = \frac{1}{n}\sum_{i=1}^{n}|Y_i - \hat{Y}_i|$

  Here, Y_i is the actual pixel value, $\hat{Y}_i$ is the predicted pixel value, and n is the total number of pixels. Lower L1 loss indicates a more accurate prediction.

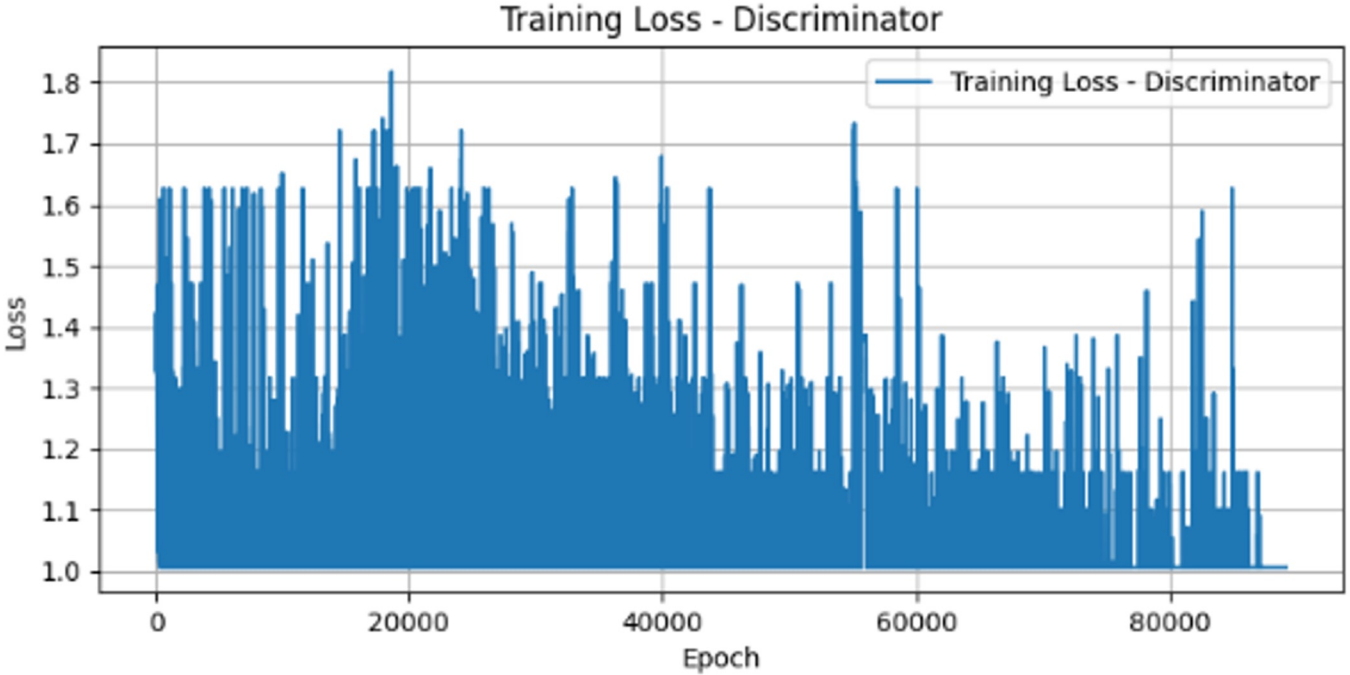
```
lambda_adv = 0.01
lambda_mse = 0.99
```

```
adversarial_loss = nn.BCEWithLogitsLoss()
mse_loss = nn.MSELoss()
l1_loss = nn.L1Loss()
```

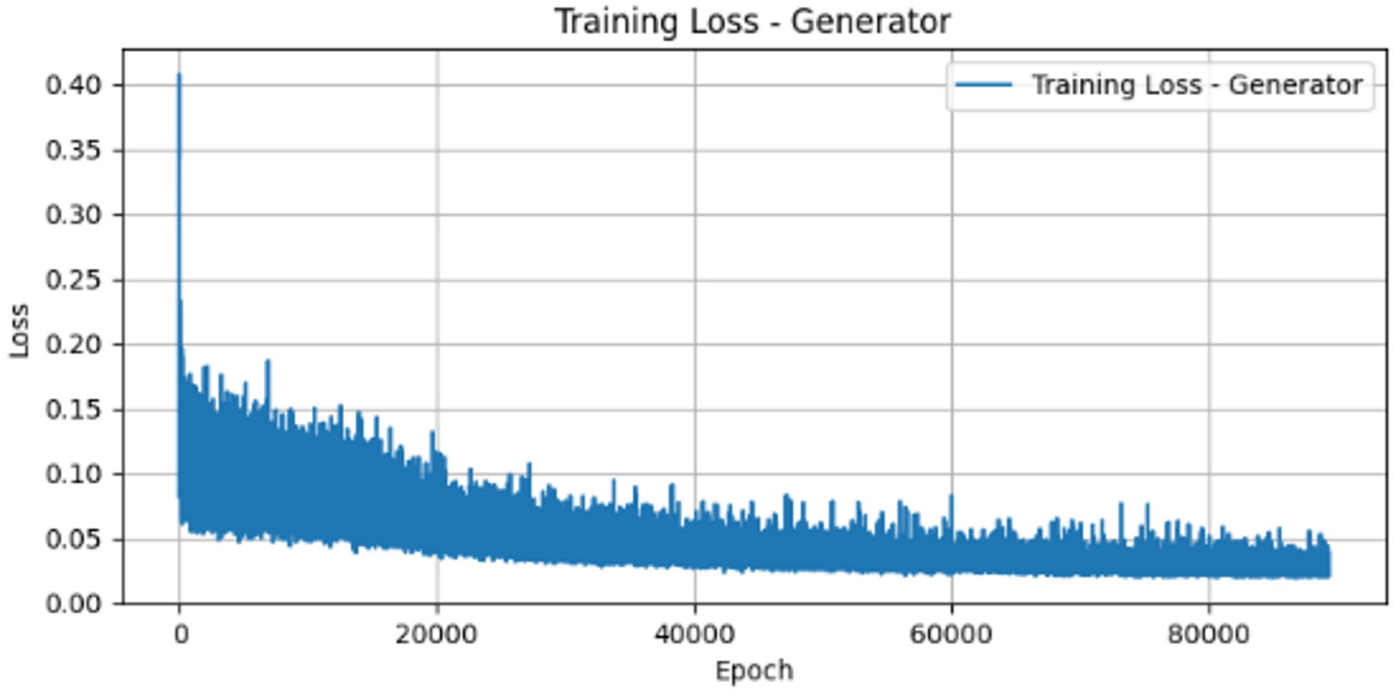
```
 # Combine adversarial loss and MSE loss
generator_loss =  lambda_adv * generator_loss_adv + lambda_mse * generator_loss_l1
```

GAN + L1 GRAPHS

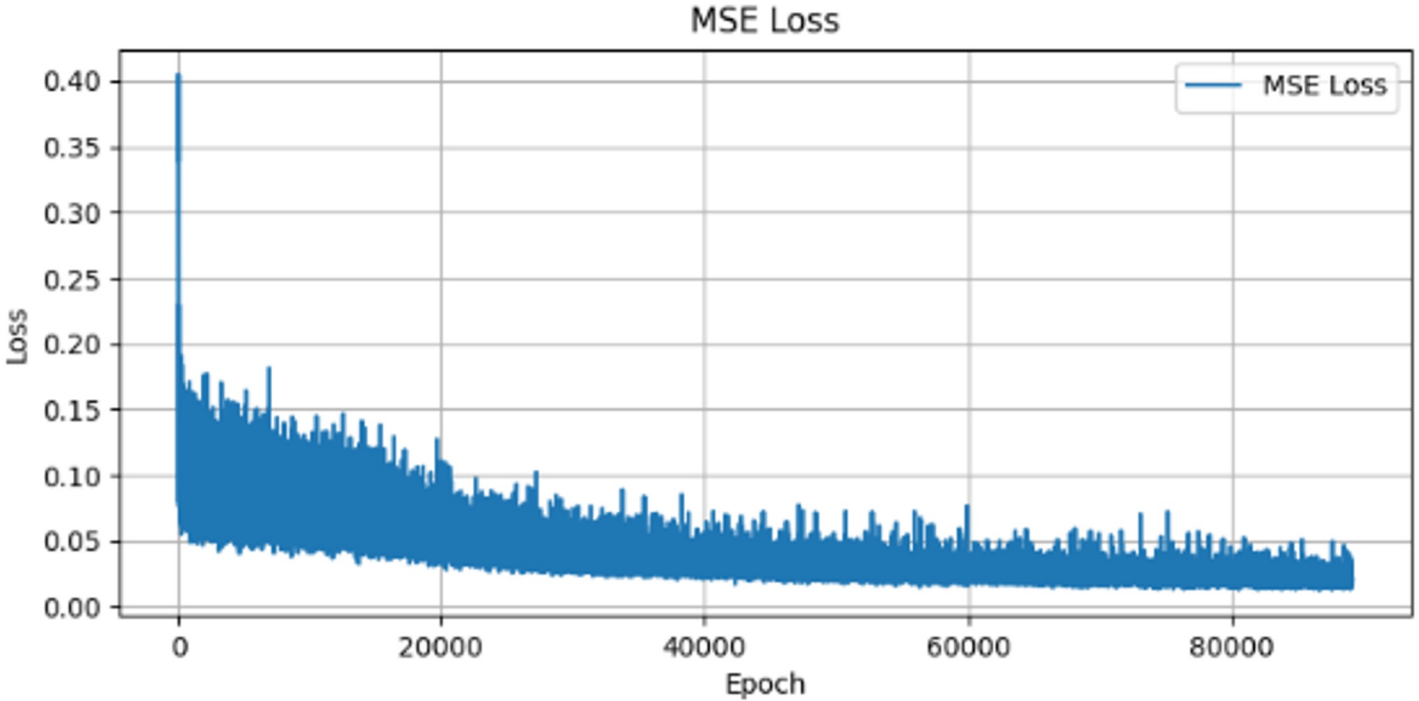
The discriminator’s training loss graph features a downward trend with noticeable variability across epochs.



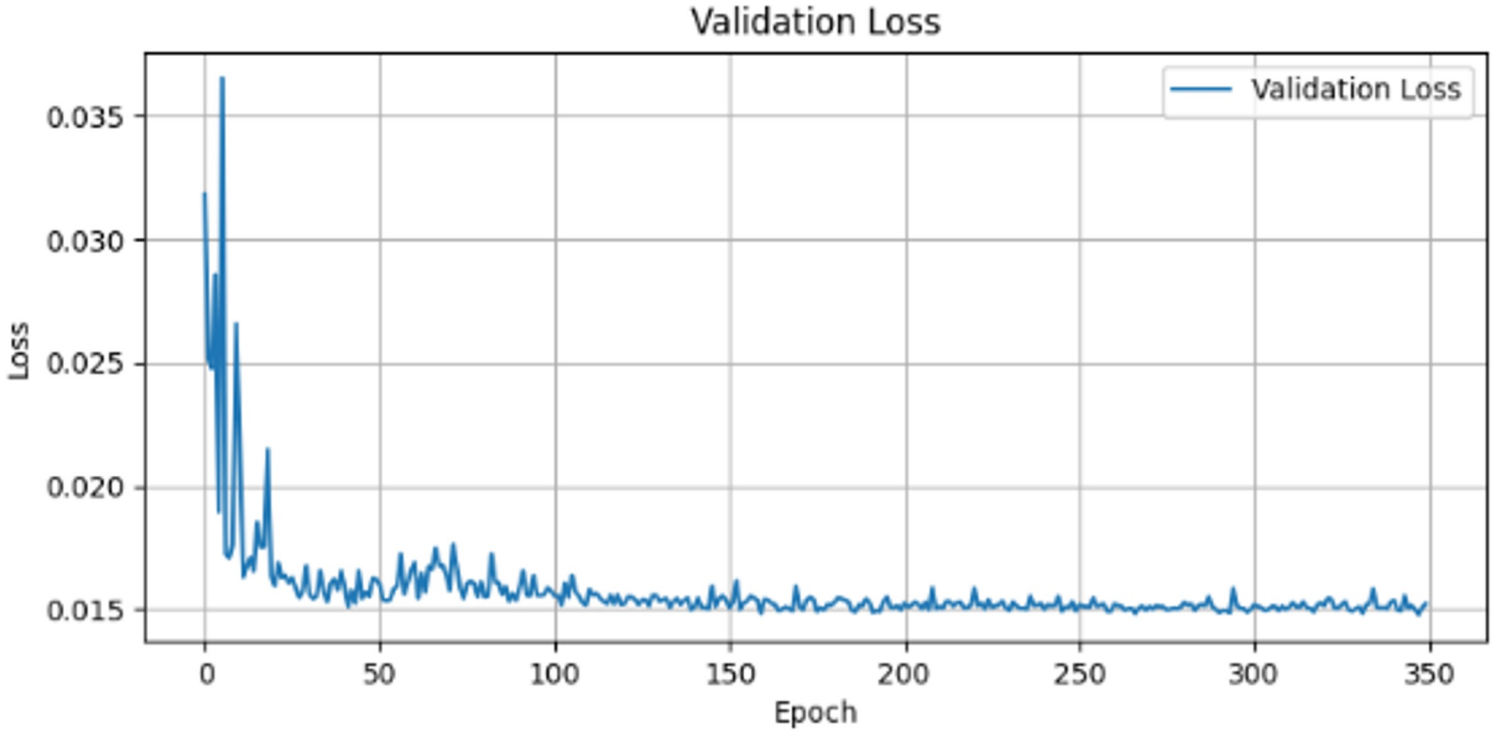
The generator's training loss graph indicates a steep initial decline, followed by a plateau, which suggests an early rapid learning phase that stabilizes as training progresses.



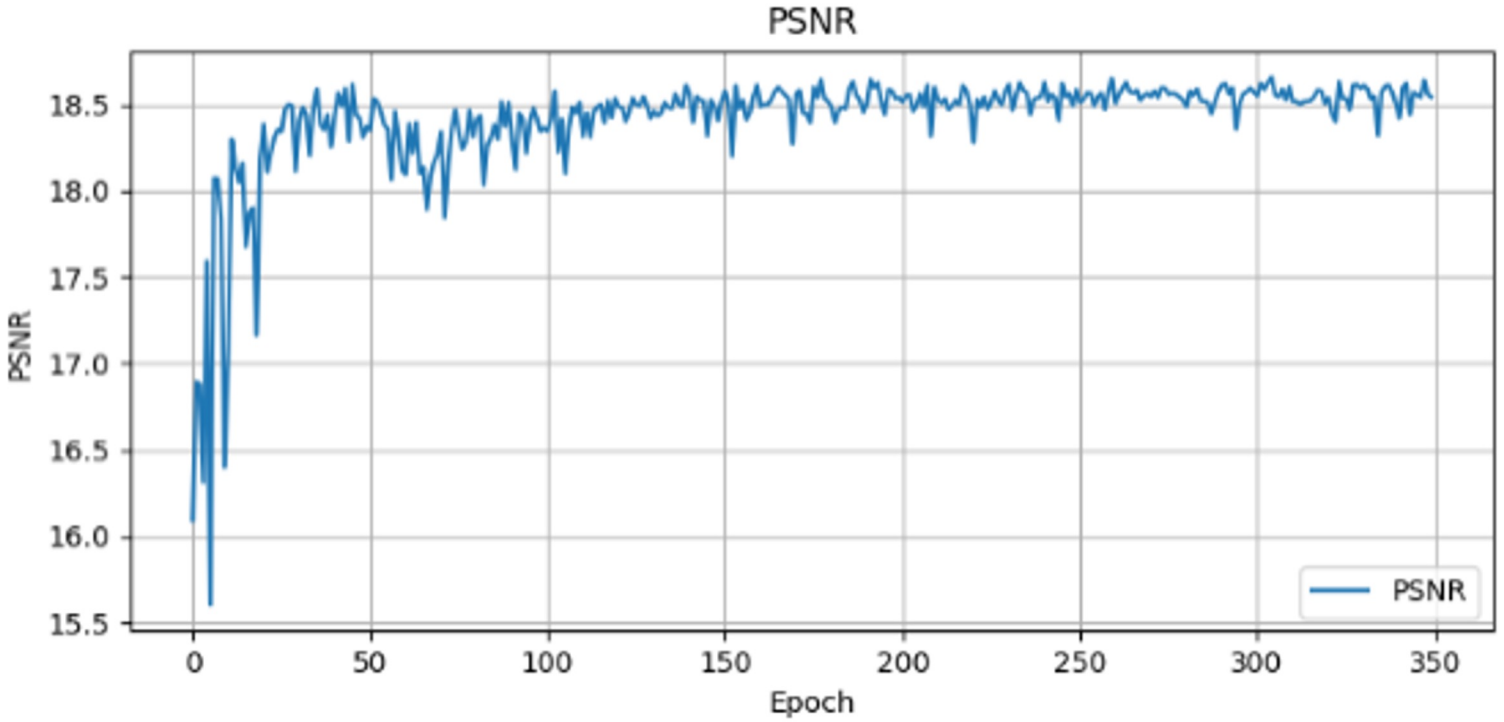
The MSE loss chart demonstrates a quick initial drop and then steadies, hinting at the model's stabilizing performance.



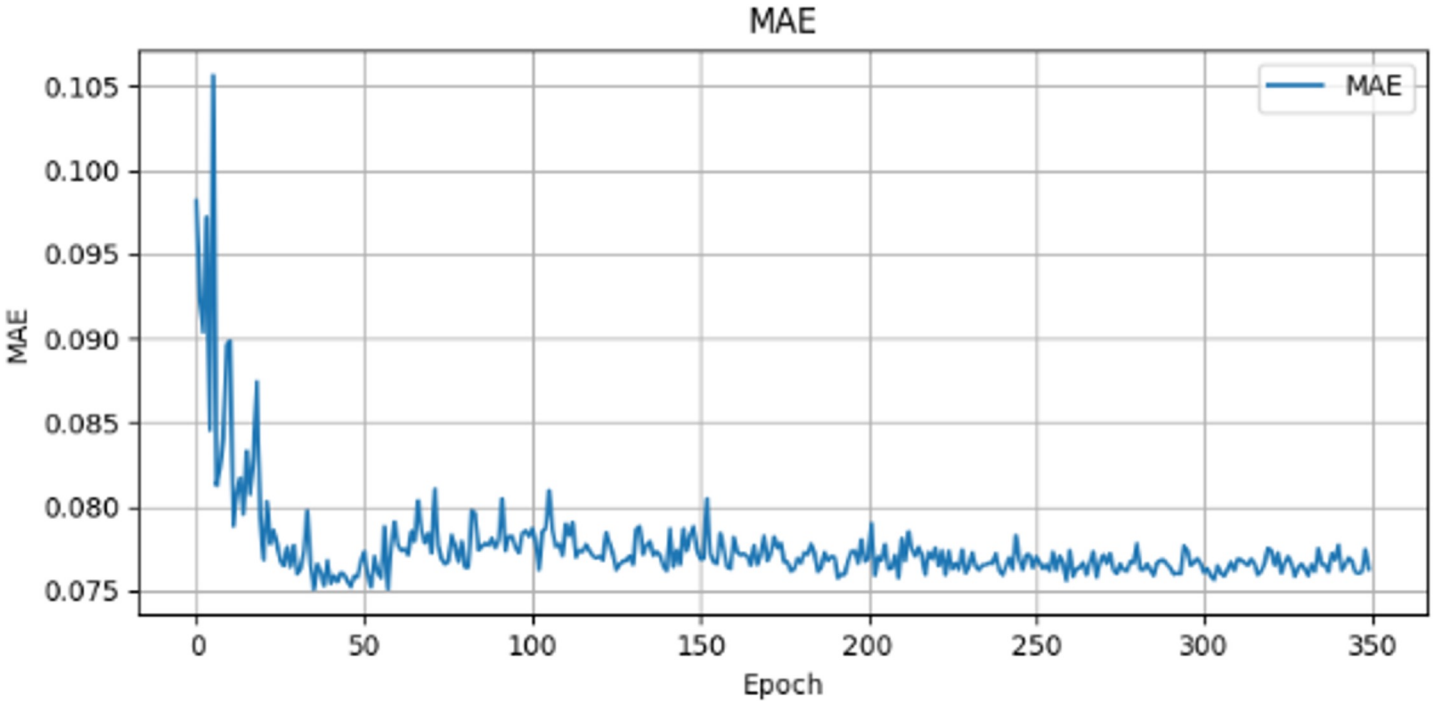
The graph for validation loss decreases sharply at the start and then maintains a relatively flat line, indicating the model's consistent performance after the initial training period.



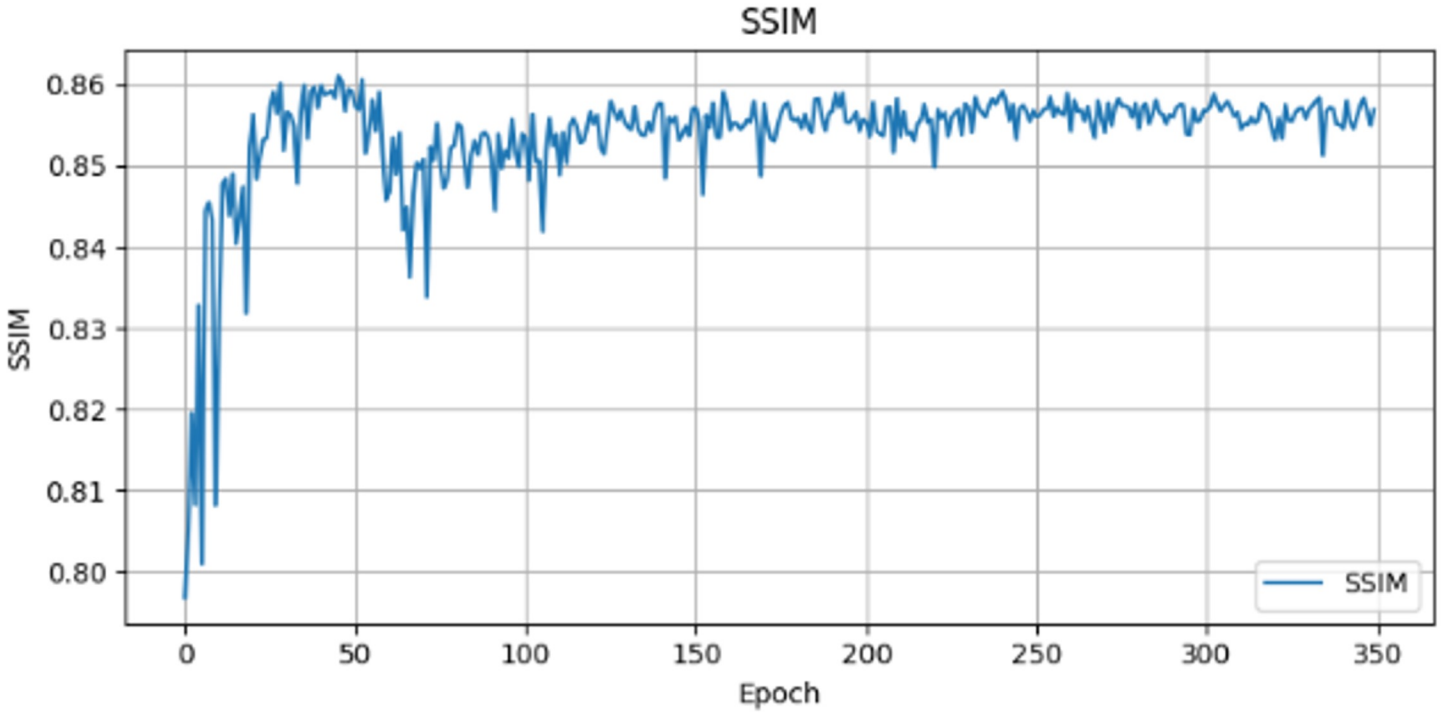
The PSNR graph initially rises quickly and then plateaus, indicating early improvements in image quality that stabilize as training proceeds.



The MAE graph shows a steep decline early in training, leading to a stable, low error as more epochs are completed.



The SSIM graph climbs sharply at the start and then remains consistently high, suggesting the model consistently preserves image structural integrity throughout training.



IMAGES TEST RESULTS GAN+L1

Grayscale Image

Generated Colorized Image

Original RGB Image

Grayscale Image

Generated Colorized Image

Original RGB Image

Grayscale Image

Generated Colorized Image

Original RGB Image

Grayscale Image

Generated Colorized Image

Original RGB Image

Grayscale Image

Generated Colorized Image

Original RGB Image

Grayscale Image

Generated Colorized Image

Original RGB Image

Grayscale Image

Generated Colorized Image

Original RGB Image

Grayscale Image

Generated Colorized Image

Original RGB Image

Final Selection

In our opinion ,**L1 loss is better than MSE** because it results in more stable and converging graphs. Visually, there is also a slight improvement in the output images; they appear more realistic .

This improved stability and realism highlight the practical benefits of L1 loss in enhancing both model performance and visual fidelity in our image processing tasks

WGAN

At the beginning of our project, we initially developed a model using the WGAN (Wasserstein Generative Adversarial Network) approach. However, this model required extensive computational resources and a very long training period—approximately two and a half weeks. Despite our efforts to train and refine the model, the substantial time required led us to reassess and ultimately change our approach due to these time constraints.

Encoder & Decoder

EncoderBlock:

-Convolutional Layer : Reduces spatial dimensions by half using a stride of 2.

-Batch Normalization : Normalizes features to stabilize learning.

-ReLU Activation : Adds non-linearity, improving the ability to capture complex patterns.

DecoderBlock:

-Transposed Convolutional Layer : Increases spatial dimensions to reconstruct features.

- Batch Normalization and ReLU Activation : Ensures smooth and effective feature enhancement during reconstruction.

```python
class EncoderBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(EncoderBlock, self).__init__()
        self.block = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        return self.block(x)

class DecoderBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DecoderBlock, self).__init__()
        self.block = nn.Sequential(
            nn.ConvTranspose2d(in_channels, out_channels, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )
```

WGAN - Unet Model

WGAN Unet Architecture

- **Encoder :**Starts with a layer of 64 channels, increasing to 128, 256, 512, and up to 1024, using convolutional layers to reduce dimensions progressively.
- **Decoder :**Consists of five decoder blocks, mirroring the encoder in reverse, from 1024 channels back down to 64, with skip connections at each stage for detail recovery.
- **Objective :**Employs Wasserstein loss with gradient penalty, enhancing training stability and image quality.

```
# Define encoder blocks
self.encoder1 = EncoderBlock(in_channels, 64)
self.encoder2 = EncoderBlock(64, 128)
self.encoder3 = EncoderBlock(128, 256)
self.encoder4 = EncoderBlock(256, 512)
self.encoder5 = EncoderBlock(512, 1024)

# Define decoder blocks=
self.decoder1 = DecoderBlock(1024, 512)
self.decoder2 = DecoderBlock(512 + 512, 256)
self.decoder3 = DecoderBlock(256 + 256, 128)
self.decoder4 = DecoderBlock(128 + 128, 64)
self.decoder5 = DecoderBlock(64 + 64, out_channels)
```

The forward method in the UNetGenerator handles the encoding and decoding within the U-Net architecture. It compresses the input through successive encoder layers, then expands it using decoder layers, incorporating skipped connections from encoder outputs to preserve detail. This process supports accurate reconstruction for tasks like image segmentation.

```
def forward(self, x):
    # Encoding
    x1 = self.encoder1(x)
    x2 = self.encoder2(x1)
    x3 = self.encoder3(x2)
    x4 = self.encoder4(x3)
    x5 = self.encoder5(x4)

    # Decoding with skip connections
    y = self.decoder1(x5)
    y = torch.cat([y, x4], dim=1)
    y = self.decoder2(y)
    y = torch.cat([y, x3], dim=1)
    y = self.decoder3(y)
    y = torch.cat([y, x2], dim=1)
    y = self.decoder4(y)
    y = torch.cat([y, x1], dim=1)
    y = self.decoder5(y)

    return y
```

Training

We found this algorithm in the internet and we used it as reference .

```
# Hyperparameters
mini_batch = 64
gradient_penalty_lambda = 10
lambda_mse = 0.95
lambda_wgan = 0.05
n_critic = 2
```

*mini_batch- is the amount of random images for the critic training.

Algorithm 1 WGAN with gradient penalty. We use default values of $\lambda = 10$, $n_{\text{critic}} = 5$, $\alpha = 0.0001$, $\beta_1 = 0$, $\beta_2 = 0.9$.

Require: The gradient penalty coefficient λ, the number of critic iterations per generator iteration n_{critic}, the batch size m, Adam hyperparameters α, β_1, β_2.
Require: initial critic parameters w_0, initial generator parameters θ_0.

while θ has not converged **do**
  for $t = 1, \ldots, n_{\text{critic}}$ **do**
    for $i = 1, \ldots, m$ **do**
      Sample real data $\boldsymbol{x} \sim \mathbb{P}_r$, latent variable $\boldsymbol{z} \sim p(\boldsymbol{z})$, a random number $\epsilon \sim U[0, 1]$.
      $\tilde{\boldsymbol{x}} \leftarrow G_\theta(\boldsymbol{z})$
      $\hat{\boldsymbol{x}} \leftarrow \epsilon \boldsymbol{x} + (1 - \epsilon)\tilde{\boldsymbol{x}}$
      $L^{(i)} \leftarrow D_w(\tilde{\boldsymbol{x}}) - D_w(\boldsymbol{x}) + \lambda(\|\nabla_{\hat{\boldsymbol{x}}} D_w(\hat{\boldsymbol{x}})\|_2 - 1)^2$
    end for
    $w \leftarrow \text{Adam}(\nabla_w \frac{1}{m} \sum_{i=1}^{m} L^{(i)}, w, \alpha, \beta_1, \beta_2)$
  end for
  Sample a batch of latent variables $\{\boldsymbol{z}^{(i)}\}_{i=1}^{m} \sim p(\boldsymbol{z})$.
  $\theta \leftarrow \text{Adam}(\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} -D_w(G_\theta(\boldsymbol{z})), \theta, \alpha, \beta_1, \beta_2)$
end while

Before presenting the results, it's important to note that due to the lengthy training time required, we were only able to complete 21 epochs, with each set of 8 epochs taking approximately 12 hours. Therefore, the results you're about to see reflect the model's performance after these 21 epochs of training.

WGAN - Results

Grayscale Image

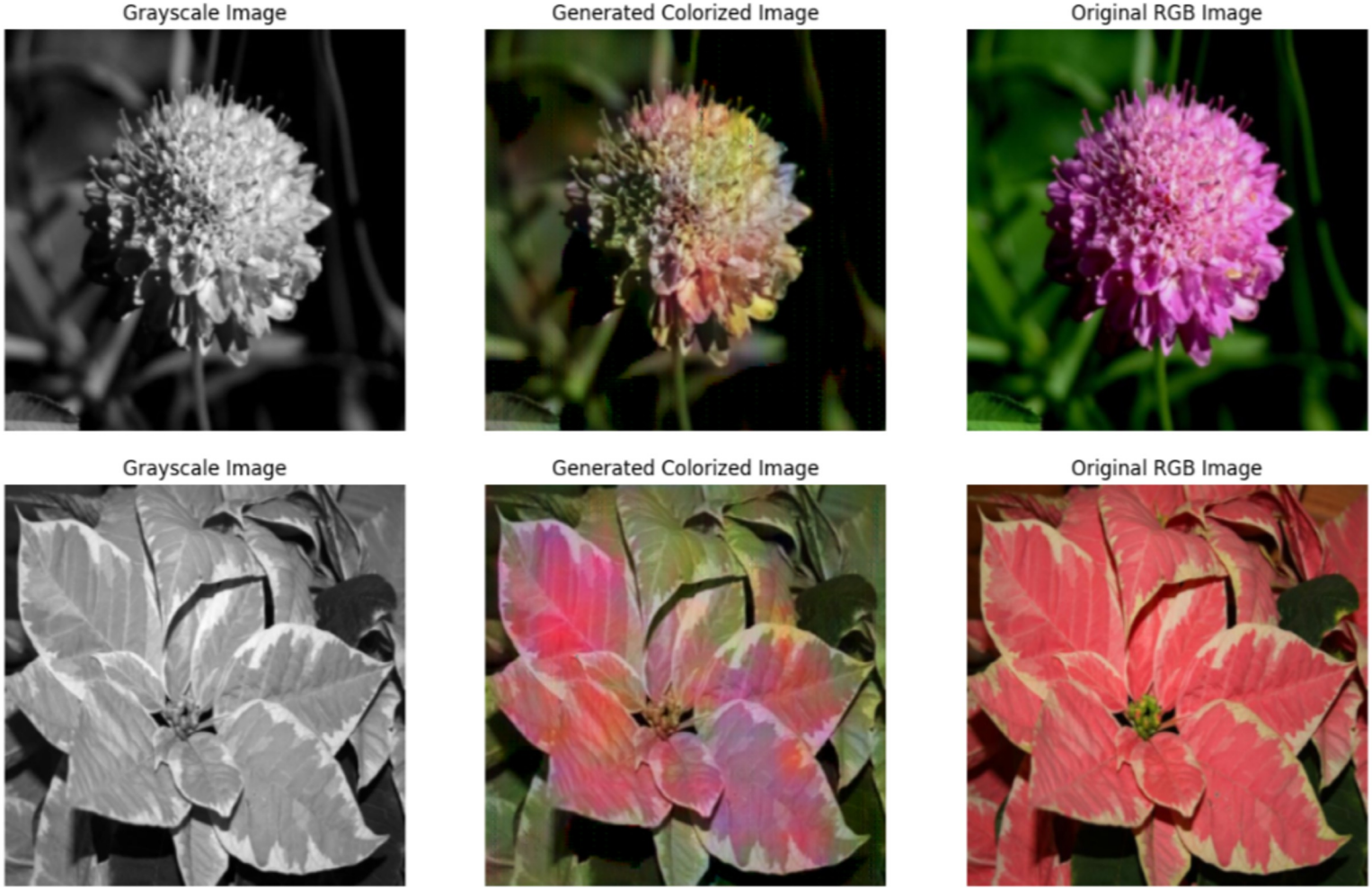
Generated Colorized Image

Original RGB Image

Grayscale Image

Generated Colorized Image

Original RGB Image


Grayscale Image
Generated Colorized Image
Original RGB Image
Grayscale Image
Generated Colorized Image
Original RGB Image

Grayscale Image

Generated Colorized Image

Original RGB Image

Grayscale Image

Generated Colorized Image

Original RGB Image

Model running instructions:

1. upload GAN-L1 notebook to Google Colab, and download the model for the link below:

   https://drive.google.com/file/d/1aHGpMI5VdSlpQUCmUJnsR8ONWL2k_MOr/view?usp=share_link

1. now upload the model to your Google drive.
2. copy the path to model from your Google drive and paste it in this line (last block)

```
model_path = '' # replace path here
```

1. run the first block (where all the imports are) and the second block (under the headline Generator - UNET)
2. run the last block
3. enjoy the results!