Introduction to Image Processing - Final Project

Mohammed Azeezulla

This project focuses on building a complete image colorization pipeline where grayscale images are converted back into realistic color versions. I explored both classical computer vision approaches and a custom GAN model built entirely from scratch. The goal was to understand how traditional algorithms behave compared to deep learning methods and then design a full U Net based generator and PatchGAN discriminator that learn to map the L channel in LAB space to accurate ab color channels. The notebook walks through dataset preparation, baseline methods, GAN architecture construction, training logic, evaluation, and visual results to show how each method performs and how learned models improve color reconstruction quality.

Section 1

In this section I set up the full pipeline needed before moving into any colorization work. I loaded the COCO dataset prepared the train and validation splits built the custom dataset class and verified that the LAB conversion and reconstruction were working correctly. This gives me a clean base so I know the data is flowing the way I expect before I start testing the classical methods.

```
# ============================================================================
# CELL 1: ALL IMPORTS
# ============================================================================
# This is my setup cell where I load everything I need for the full project.
# All the core tools for files images training and metrics come in here once so I do not repeat later.

# Standard Libraries
import os
from pathlib import Path
import glob
import time
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from tqdm import tqdm

# Image Processing
# These help me open images work with color spaces and measure reconstruction quality
import PIL
from PIL import Image
from skimage.color import rgb2lab, lab2rgb
from skimage import exposure
from skimage.metrics import peak_signal_noise_ratio, structural_similarity as ssim

# PyTorch
# These are the deep learning tools I use to build and train the GAN in this project
import torch
from torch import nn, optim
from torchvision import transforms
from torchvision.models.resnet import resnet18  # Only for discriminator if needed, NOT for generator
from torch.utils.data import Dataset, DataLoader

# FastAI (for data loading only)
# I only use this to quickly download the COCO sample dataset
from fastai.data.external import untar_data, URLs

# For FID calculation
# This will be useful later if I want to add FID as an extra metric
from scipy import linalg

# Warnings
# I silence warnings so the training logs stay clean and easy to read
import warnings
warnings.filterwarnings('ignore')

# Device Configuration
# Here I check if CUDA is available and decide whether I am running on GPU or CPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")
```

```
# ============================================================================
# CELL 2: CONFIGURATION CLASS
# ============================================================================
# This class holds every setting I use throughout the project.
# I keep all values here so I can control the entire pipeline from one place.
# Makes the experiment clean and easy to tune when I want to adjust something.

class Config:
    # Data Parameters
    # Deciding how many images I want to work with and the image size I want everything resized to.
    external_data_size = 10000
    train_size = 8000
    image_size_1 = 256
    image_size_2 = 256

    # Training Parameters
    # Basic training setup that controls how many samples per batch and how many epochs I train for.
    batch_size = 32
    epochs = 15

    # Architecture Parameters
    # These define how the UNet and PatchGAN layers behave while downsampling and upsampling.
    kernel_size = 4
    stride = 2
    padding = 1
    dropout = 0.5
    LeakyReLU_slope = 0.2

    # Optimizer Parameters
    # Settings for the Adam optimizers for both generator and discriminator.
    gen_lr = 2e-4
    disc_lr = 2e-4
    beta1 = 0.5
    beta2 = 0.999

    # Loss Parameters
    # Lambda L1 controls how strongly I push the generator to match ground truth colors.
    gan_mode = 'vanilla'   # I also keep option for lsgan if I want to test it later.
    lambda_l1 = 100

    # Generator Parameters (Custom U Net)
    # Generator takes only L channel and predicts ab so 1 input channel and 2 output channels.
    gen_input_channels = 1
    gen_output_channels = 2
    gen_n_down = 8
    gen_num_filters = 64

    # Discriminator Parameters
    # Discriminator sees L and ab together so 3 input channels for LAB.
    disc_input_channels = 3
    disc_num_filters = 64
    disc_n_down = 3

    # Pretraining (NOT USED but I keep it for reference)
    pretrain_lr = 1e-4
    layers_to_cut = -2

# Printing configuration so I know everything is set correctly
print("Configuration loaded")
print(f"  - Training images: {Config.train_size}")
print(f"  - Validation images: {Config.external_data_size - Config.train_size}")
print(f"  - Image size: {Config.image_size_1}x{Config.image_size_2}")
print(f"  - Batch size: {Config.batch_size}")
print(f"  - Epochs: {Config.epochs}")
```

```
Configuration loaded
  - Training images: 8000
  - Validation images: 2000
  - Image size: 256x256
  - Batch size: 32
  - Epochs: 15
```

The full configuration is now set up with the dataset split image size batch settings and total epochs all ready for the training pipeline to run smoothly.

```
# ============================================================================
# CELL 3: DOWNLOAD AND LOAD COCO DATASET
# ============================================================================
# In this cell I pull in the COCO sample dataset through FastAI.
# I choose a fixed subset so the whole experiment stays consistent every time I run it.

# Download COCO sample dataset
print("Downloading COCO dataset...")
path = untar_data(URLs.COCO_SAMPLE)
path = str(path) + "/train_sample"

# Get all image paths inside the dataset folder
paths = glob.glob(path + "/*.jpg")
print(f"Found {len(paths)} images in dataset")

# Train and validation split
# I randomly pick the subset I want to work with and then divide it into train and validation sets.
np.random.seed(42)
paths_subset = np.random.choice(paths, Config.external_data_size, replace=False)
print(f"Selected {Config.external_data_size} images randomly")

random_idxs = np.random.permutation(Config.external_data_size)
train_idxs = random_idxs[:Config.train_size]
val_idxs = random_idxs[Config.train_size:]

train_paths = paths_subset[train_idxs]
val_paths = paths_subset[val_idxs]

print(f"Train set: {len(train_paths)} images")
print(f"Validation set: {len(val_paths)} images")
```

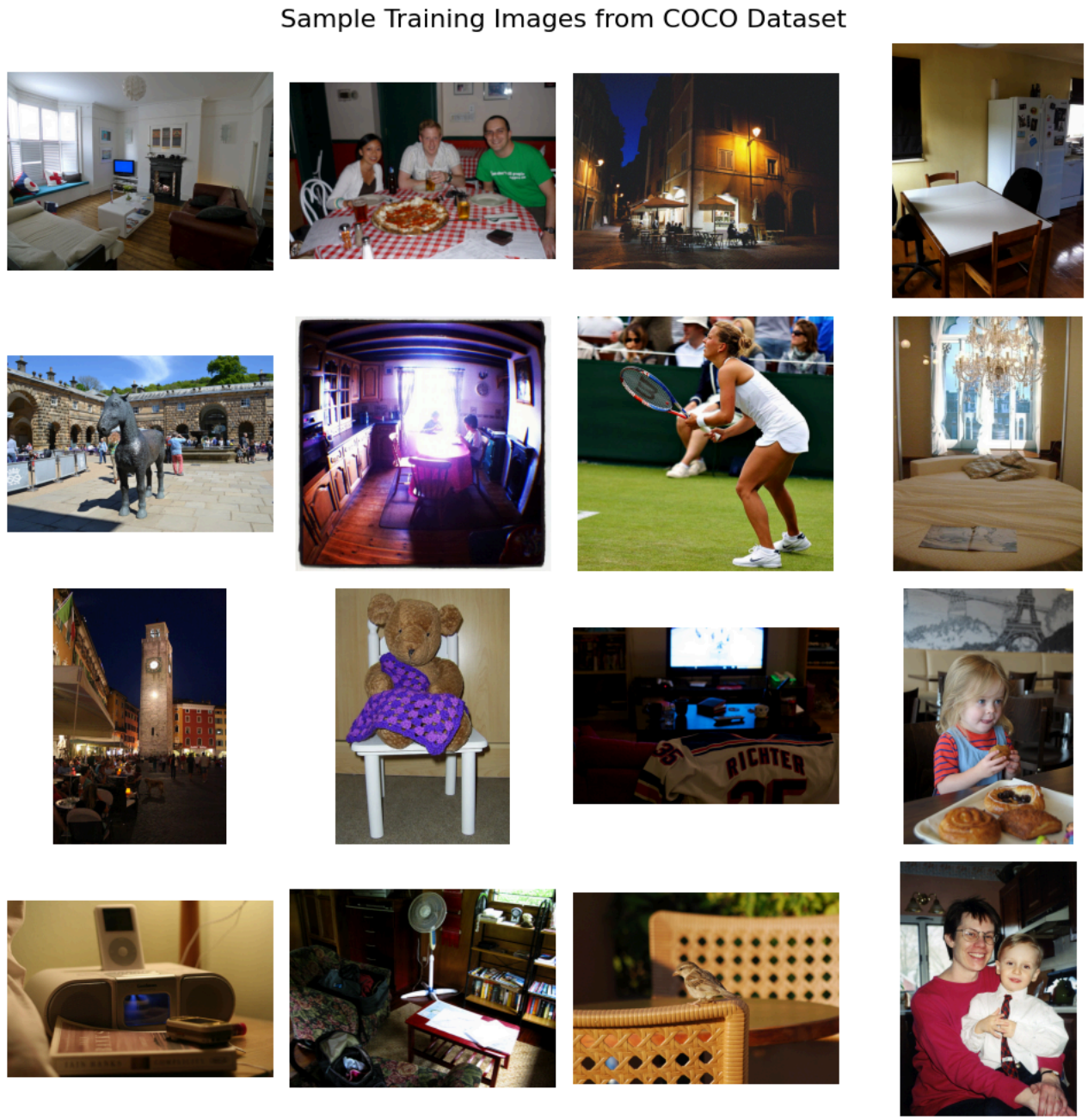
```
Downloading COCO dataset...
Found 21837 images in dataset
Selected 10000 images randomly
Train set: 8000 images
Validation set: 2000 images
```

```
# ============================================================================
# VERIFY DATASET LOCATION
# ============================================================================
# I run this small check to make sure the COCO dataset was downloaded properly
# and to confirm the directory structure before I start using it.

import os
from pathlib import Path

# Check if dataset exists and print its contents
path = untar_data(URLs.COCO_SAMPLE)
print(f"Dataset path: {path}")
print(f"Exists: {os.path.exists(path)}")
print("\nContents:")
for item in os.listdir(path):
    print(f"  - {item}")
```

```
Dataset path: C:\Users\Owner\.fastai\data\coco_sample
Exists: True

Contents:
  - annotations
  - train_sample
```

```
# ============================================================================
# CELL 4: VISUALIZE SAMPLE IMAGES
# ============================================================================
# I always like to look at a few samples before training just to confirm
# that images loaded correctly and the dataset looks as expected.

# Display a 4x4 grid of training images
fig, axes = plt.subplots(4, 4, figsize=(10, 10))

for ax, img_path in zip(axes.flatten(), train_paths[:16]):
    img = Image.open(img_path)
    ax.imshow(img)
    ax.axis("off")

plt.suptitle("Sample Training Images from COCO Dataset", fontsize=16)
plt.tight_layout()
```

```
plt.show()

print("Displayed 16 sample training images")
```




```
Displayed 16 sample training images
```

```
# ================================================================================
# CELL 5: IMAGE DATASET CLASS
# ================================================================================
# This class handles how I load every image in the dataset.
# I resize the image apply simple augmentation for training and convert it into LAB space.
# I also standardize the L and ab channels so the model trains smoothly.

class ImageDataset(Dataset):
    '''
    Dataset class for loading and preprocessing images.
    I resize the images apply random flip during training convert to LAB
    and then scale everything into a clean range before feeding it to the network.
    '''

    def __init__(self, paths, train=True):
        # If this is training data I apply a flip for simple augmentation
        if train == True:
            self.transforms = transforms.Compose([
```

```
                transforms.Resize((Config.image_size_1, Config.image_size_2)),
                transforms.RandomHorizontalFlip()
            ])
        # Validation images only get resized to keep them consistent
        elif train == False:
            self.transforms = transforms.Compose([
                transforms.Resize((Config.image_size_1, Config.image_size_2))
            ])

        self.train = train
        self.paths = paths

    def __len__(self):
        # Total number of images
        return len(self.paths)

    def __getitem__(self, idx):
        # Load image and always convert to RGB
        img = Image.open(self.paths[idx]).convert("RGB")
        img = self.transforms(img)
        img = np.array(img)

        # Convert RGB image to LAB so I can isolate the L channel and predict ab
        lab = rgb2lab(img).astype("float32")
        lab = transforms.ToTensor()(lab)

        # Standardize the channels into the ranges the GAN expects
        L = lab[[0], ...] / 50.0 - 1.0      # L goes to [-1, 1]
        ab = lab[[1, 2], ...] / 128.0       # ab goes to [-1, 1]

        return {'L': L, 'ab': ab}

print("ImageDataset class defined")
```

```
ImageDataset class defined
```

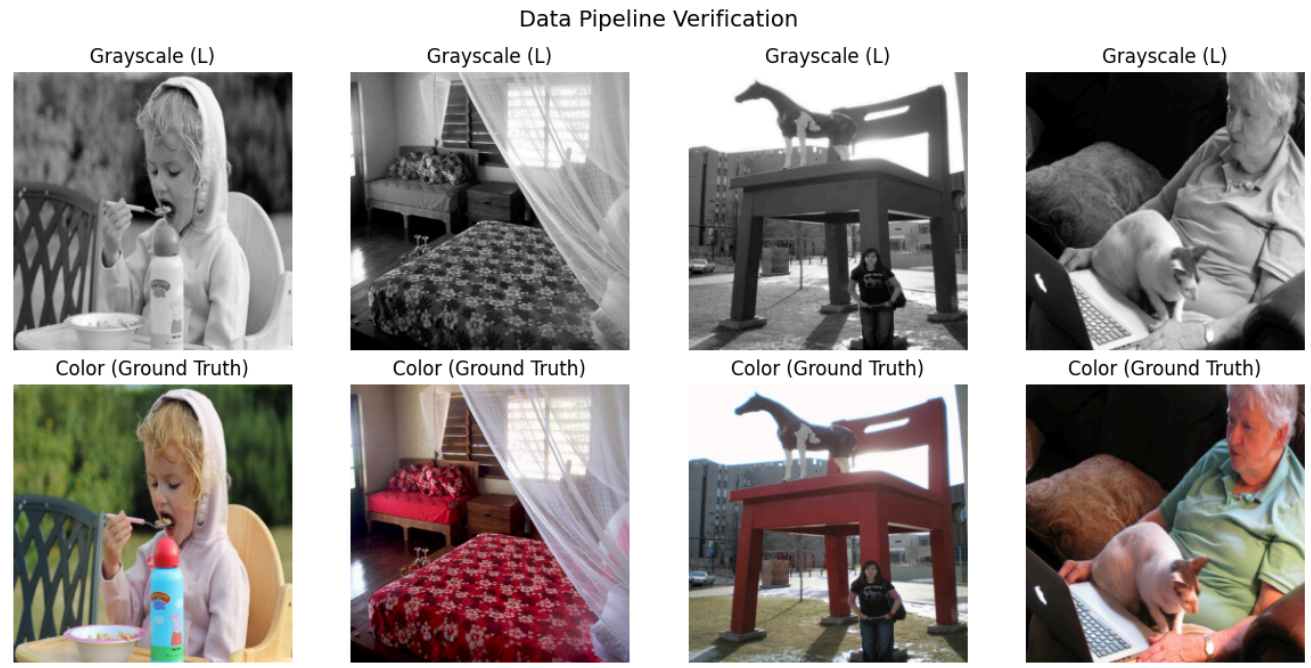
```
# ============================================================================
# CELL 6: CREATE DATALOADERS
# ============================================================================
# Now that the dataset class is ready I wrap it inside dataloaders.
# This is where batching shuffling and fast loading come in so training runs smoothly.

# Create datasets for training and validation
train_dataset = ImageDataset(paths=train_paths, train=True)
val_dataset = ImageDataset(paths=val_paths, train=False)

# Create dataloaders
# Training loader shuffles so the model sees data in a different order each epoch
train_loader = DataLoader(
    train_dataset,
    batch_size=Config.batch_size,
    shuffle=True,
    pin_memory=True
)

# Validation loader stays in order since I only evaluate on it
val_loader = DataLoader(
    val_dataset,
    batch_size=Config.batch_size,
    shuffle=False,
    pin_memory=True
)

print("DataLoaders created")
print(f"  Train batches: {len(train_loader)}")
print(f"  Val batches: {len(val_loader)}")
print(f"  Batch size: {Config.batch_size}")

# Quick check to confirm shapes coming out of the loader
sample_batch = next(iter(train_loader))
print("\nSample batch shapes:")
print(f"  L channel: {sample_batch['L'].shape}")
print(f"  ab channels: {sample_batch['ab'].shape}")
```

```
DataLoaders created
  Train batches: 250
  Val batches: 63
```

```
  Batch size: 32

Sample batch shapes:
  L channel: torch.Size([32, 1, 256, 256])
  ab channels: torch.Size([32, 2, 256, 256])
```

The dataloaders are set up correctly with the expected batch counts and the sample batch confirms that both L and ab channels are coming in with the right shapes for training.

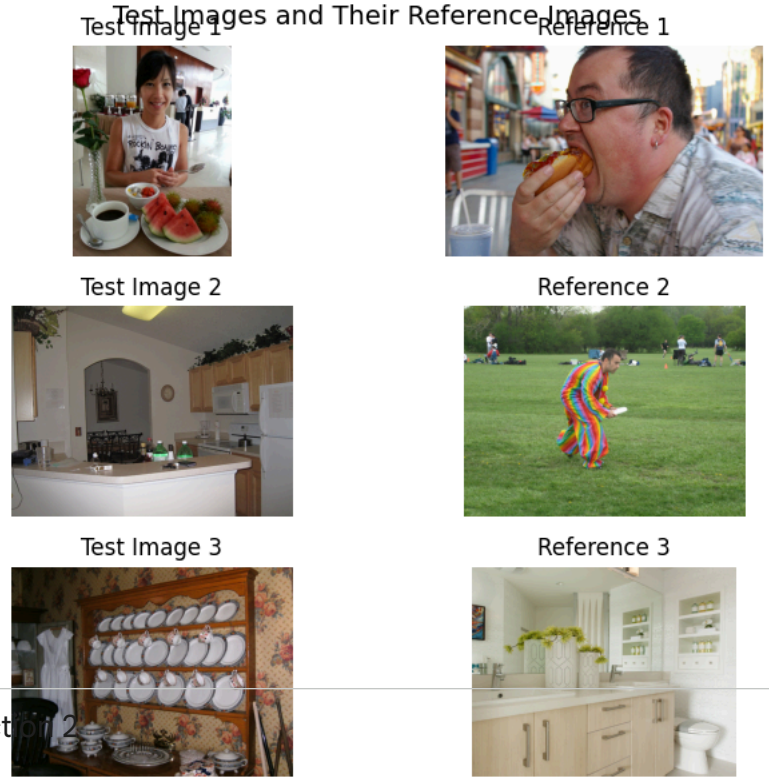
```
# ============================================================================
# CELL 7: VERIFY DATA PIPELINE
# ============================================================================
# This step is just to make sure my preprocessing is correct.
# I take a batch convert L and ab back to RGB and check if the reconstruction looks right.

# Helper function to convert LAB batch back to RGB for visualization
def lab_to_rgb_batch(L, ab):
    """
    Convert LAB batch to RGB.
    L is in [-1, 1] and ab is in [-1, 1] so I undo the scaling before converting back.
    """
    # Unstandardize L and ab channels
    L = (L + 1.0) * 50.0
    ab = ab * 128.0

    # Merge L and ab into a single LAB image and move to CPU for conversion
    Lab = torch.cat([L, ab], dim=1).permute(0, 2, 3, 1).cpu().numpy()

    rgb_imgs = []
    for img in Lab:
        img_rgb = lab2rgb(img)
        rgb_imgs.append(img_rgb)

    return np.stack(rgb_imgs, axis=0)

# Pull a sample batch from the training loader
sample_batch = next(iter(train_loader))
L_batch = sample_batch['L']
ab_batch = sample_batch['ab']

# Convert LAB batch back to RGB
rgb_batch = lab_to_rgb_batch(L_batch, ab_batch)

# Display a few images to confirm everything is working
fig, axes = plt.subplots(2, 4, figsize=(12, 6))

for i in range(4):
    # Only the L channel (grayscale)
    axes[0, i].imshow(L_batch[i][0].cpu(), cmap='gray')
    axes[0, i].axis('off')
    axes[0, i].set_title('Grayscale (L)')

    # Full RGB reconstructed from L and ab channels
    axes[1, i].imshow(rgb_batch[i])
    axes[1, i].axis('off')
    axes[1, i].set_title('Color (Ground Truth)')

plt.suptitle('Data Pipeline Verification', fontsize=14)
plt.tight_layout()
plt.show()

print("Data pipeline verified successfully")
```




```
Data pipeline verified successfully
```

```
# ============================================================================
# CELL 9: SELECT REFERENCE IMAGES FOR CLASSICAL METHOD
# ============================================================================
# Here I pick a small set of validation images that I will use to test the
# classical colorization methods. For each test image I also pick a random
# reference image from the training set so I can transfer color from it.

# Select 10 test images for classical colorization
np.random.seed(123)
test_sample_paths = np.random.choice(val_paths, 10, replace=False)

print(f"Selected {len(test_sample_paths)} test images for classical method")

# For each test image I pick a random reference image from the training set
np.random.seed(456)
reference_paths = np.random.choice(train_paths, len(test_sample_paths), replace=False)

print(f"Selected {len(reference_paths)} reference images")

# Visualize test and reference image pairs
fig, axes = plt.subplots(len(test_sample_paths), 2, figsize=(8, 20))

for i, (test_path, ref_path) in enumerate(zip(test_sample_paths, reference_paths)):
    # Load the actual images
    test_img = Image.open(test_path).convert("RGB")
    ref_img = Image.open(ref_path).convert("RGB")

    # Show test image
    axes[i, 0].imshow(test_img)
    axes[i, 0].set_title(f'Test Image {i+1}')
    axes[i, 0].axis('off')

    # Show reference image
    axes[i, 1].imshow(ref_img)
    axes[i, 1].set_title(f'Reference {i+1}')
    axes[i, 1].axis('off')

plt.suptitle('Test Images and Their Reference Images', fontsize=14)
plt.tight_layout()
plt.show()

print("Test Reference pairs visualized")
```

```
Selected 10 test images for classical method
Selected 10 reference images
```



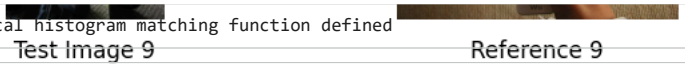
Section 2

In this section I worked through three classical colorization methods and tested them on a fixed set of validation images. I started by pairing each grayscale test image with a random reference image then applied histogram matching K means color transfer and a Gaussian based local method. After running all three I calculated PSNR and SSIM for each sample built a comparison table and saved all outputs for inspection. This gives me a strong baseline before switching to the GAN based model.

```
# ============================================================================
# SECTION 2: CLASSICAL BASELINE
# ============================================================================
# CELL 8: HISTOGRAM MATCHING FUNCTION
# ============================================================================
# This is the first classical method I use. The idea is simple.
# I take the grayscale image in LAB space and shift its a and b channels
# so their distributions match the reference color image. This gives a quick
# deterministic color transfer without any learning.

def classical_histogram_matching(gray_img, reference_img):
    """
    Classical colorization using histogram matching.
    I convert both images to LAB then align the a and b channel distributions.
    """

    # Convert both images to LAB so I can work directly on the color channels
    gray_lab = rgb2lab(gray_img)
    ref_lab = rgb2lab(reference_img)

    # Copy the grayscale LAB image before modifying its color channels
    matched_lab = gray_lab.copy()

    # Match histogram of the a channel
    matched_lab[:, :, 1] = exposure.match_histograms(
        gray_lab[:, :, 1],
        ref_lab[:, :, 1],
        channel_axis=None
    )

    # Match histogram of the b channel
    matched_lab[:, :, 2] = exposure.match_histograms(
        gray_lab[:, :, 2],
        ref_lab[:, :, 2],
        channel_axis=None
    )
```

```
    # Convert back to RGB so I can visualize the final color output
    colorized_rgb = lab2rgb(matched_lab)

    return colorized_rgb

print("Classical histogram matching function defined")
```

```
Classical histogram matching function defined
```




```
# ============================================================================
# CELL 10: RUN CLASSICAL COLORIZATION
# ============================================================================
# Here I run the histogram matching method on all selected test images.
# For every test image I build its grayscale version choose a reference image
# and apply the classical color transfer.

# Store results for later evaluation and visualization
classical_results = []

print("Running classical colorization...")

for i, (test_path, ref_path) in enumerate(zip(test_sample_paths, reference_paths)):
    # Load test and reference images and resize to 256x256
    test_img = np.array(Image.open(test_path).convert("RGB").resize((256, 256)))
    ref_img = np.array(Image.open(ref_path).convert("RGB").resize((256, 256)))

    # Convert test image to grayscale but keep 3 channels for processing
    gray_img = np.array(Image.open(test_path).convert("L").resize((256, 256)))
    gray_img = np.stack([gray_img, gray_img, gray_img], axis=-1)

    # Apply the classical histogram matching colorization
    colorized = classical_histogram_matching(gray_img, ref_img)

    # Save everything so I can later compute PSNR SSIM and visualize
    classical_results.append({
        'original': test_img,
        'grayscale': gray_img,
        'reference': ref_img,
        'colorized': colorized
    })

    print(f"  Processed image {i+1}/{len(test_sample_paths)}")

print(f"\nClassical colorization complete: {len(classical_results)} images")
```

```
Running classical colorization...
  Processed image 1/10
  Processed image 2/10
  Processed image 3/10
  Processed image 4/10
  Processed image 5/10
  Processed image 6/10
  Processed image 7/10
  Processed image 8/10
  Processed image 9/10
  Processed image 10/10

Classical colorization complete: 10 images
```

```
# ============================================================================
# CELL 11: CALCULATE CLASSICAL METRICS (PSNR, SSIM)
# ============================================================================
# Now I check how well the classical method performed by computing PSNR and SSIM.
# These two metrics tell me how close the colorized output is to the ground truth.

def calculate_psnr(img1, img2):
    """
    Calculate PSNR between two images.
    Both images must be in [0, 1] before comparison.
    """
    img1 = np.clip(img1, 0, 1)
    img2 = np.clip(img2, 0, 1)
    return peak_signal_noise_ratio(img1, img2, data_range=1.0)


def calculate_ssim(img1, img2):
    """
```

```
    Calculate SSIM between two images.
    SSIM looks at structure similarity rather than just raw pixel error.
    """
    img1 = np.clip(img1, 0, 1)
    img2 = np.clip(img2, 0, 1)
    return ssim(img1, img2, multichannel=True, channel_axis=2, data_range=1.0)


# Calculate metrics for the full classical baseline
print("Calculating Classical Baseline Metrics...")
print("=" * 60)

classical_psnr_scores = []
classical_ssim_scores = []

for i, result in enumerate(classical_results):
    original = result['original'] / 255.0   # Convert to [0, 1]
    colorized = result['colorized']         # Already in [0, 1]

    psnr_val = calculate_psnr(original, colorized)
    ssim_val = calculate_ssim(original, colorized)

    classical_psnr_scores.append(psnr_val)
    classical_ssim_scores.append(ssim_val)

    print(f"Image {i+1:2d} - PSNR: {psnr_val:6.2f} dB | SSIM: {ssim_val:.4f}")

# Compute averages and spread
avg_psnr = np.mean(classical_psnr_scores)
avg_ssim = np.mean(classical_ssim_scores)
std_psnr = np.std(classical_psnr_scores)
std_ssim = np.std(classical_ssim_scores)

print("=" * 60)
print(" CLASSICAL BASELINE RESULTS (Histogram Matching):")
print(f"   Average PSNR: {avg_psnr:.2f} ± {std_psnr:.2f} dB")
print(f"   Average SSIM: {avg_ssim:.4f} ± {std_ssim:.4f}")
print("=" * 60)

# Store everything for later comparison with other classical methods
classical_metrics = {
    'method': 'Histogram Matching',
    'psnr_scores': classical_psnr_scores,
    'ssim_scores': classical_ssim_scores,
    'avg_psnr': avg_psnr,
    'avg_ssim': avg_ssim,
    'std_psnr': std_psnr,
    'std_ssim': std_ssim
}

print("Metrics calculated and stored")
```

```
Calculating Classical Baseline Metrics...
============================================================
Image  1 - PSNR:  21.30 dB | SSIM: 0.8902
Image  2 - PSNR:  18.56 dB | SSIM: 0.8696
Image  3 - PSNR:  22.91 dB | SSIM: 0.8945
Image  4 - PSNR:  20.55 dB | SSIM: 0.8482
Image  5 - PSNR:  20.83 dB | SSIM: 0.8684
Image  6 - PSNR:  18.08 dB | SSIM: 0.8403
Image  7 - PSNR:  19.51 dB | SSIM: 0.8523
Image  8 - PSNR:  19.48 dB | SSIM: 0.8248
Image  9 - PSNR:  18.06 dB | SSIM: 0.7926
Image 10 - PSNR:  18.17 dB | SSIM: 0.7717
============================================================
 CLASSICAL BASELINE RESULTS (Histogram Matching):
   Average PSNR: 19.75 ± 1.55 dB
   Average SSIM: 0.8453 ± 0.0378
============================================================
Metrics calculated and stored
```

```
# ============================================================================
# CELL 12: ADDITIONAL CLASSICAL BASELINE - K MEANS COLOR TRANSFER (FUNCTION)
# ============================================================================
# This is the second classical method I use.
# Here I cluster the reference image based on luminosity and then transfer the
# average ab values of each cluster onto the grayscale image. This gives a more
# structured color transfer compared to histogram matching.
```

```
from sklearn.cluster import KMeans

def classical_kmeans_colorization(gray_img, reference_img, n_clusters=8):
    """
    Colorization using K means based color transfer.
    I group pixels by their L channel values and assign colors from the matching
    clusters in the reference image.
    """

    # Convert both images to LAB since clustering will be done on the L channel
    gray_lab = rgb2lab(gray_img)
    ref_lab = rgb2lab(reference_img)

    # Flatten luminosity values so I can feed them to K means
    gray_L = gray_lab[:, :, 0].flatten().reshape(-1, 1)
    ref_L = ref_lab[:, :, 0].flatten()
    ref_ab = ref_lab[:, :, 1:].reshape(-1, 2)

    # Run K means on the reference image to learn luminosity clusters
    kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init=10)
    ref_clusters = kmeans.fit_predict(ref_L.reshape(-1, 1))

    # Compute mean ab color for each cluster
    cluster_colors = np.zeros((n_clusters, 2))
    for i in range(n_clusters):
        cluster_mask = (ref_clusters == i)
        if cluster_mask.sum() > 0:
            cluster_colors[i] = ref_ab[cluster_mask].mean(axis=0)

    # Predict the cluster for each pixel in the grayscale image
    gray_clusters = kmeans.predict(gray_L)

    # Build the final LAB image by assigning cluster colors
    colorized_lab = gray_lab.copy()
    colorized_ab = cluster_colors[gray_clusters].reshape(gray_lab.shape[0], gray_lab.shape[1], 2)
    colorized_lab[:, :, 1:] = colorized_ab

    # Convert back to RGB for the final output
    colorized_rgb = lab2rgb(colorized_lab)

    return colorized_rgb

print("K means colorization function defined")
```

```
K means colorization function defined
```

```
# ============================================================================
# CELL 13: RUN K MEANS COLORIZATION ON TEST IMAGES
# ============================================================================
# Here I apply the K means color transfer method to the same test images.
# For each test image I load the grayscale version pick a reference image
# run the K means colorization and store everything for later comparison.

# Store results
kmeans_results = []
print("Running K means colorization...")

for i, (test_path, ref_path) in enumerate(zip(test_sample_paths, reference_paths)):
    # Load test and reference images
    test_img = np.array(Image.open(test_path).convert("RGB").resize((256, 256)))
    ref_img = np.array(Image.open(ref_path).convert("RGB").resize((256, 256)))

    # Convert test image to grayscale and keep 3 channels so processing stays consistent
    gray_img = np.array(Image.open(test_path).convert("L").resize((256, 256)))
    gray_img = np.stack([gray_img, gray_img, gray_img], axis=-1)

    # Run the K means colorization
    colorized = classical_kmeans_colorization(gray_img, ref_img, n_clusters=8)

    # Save all outputs for evaluation
    kmeans_results.append({
        'original': test_img,
        'grayscale': gray_img,
        'reference': ref_img,
        'colorized': colorized
    })
```

```
    print(f"  Processed image {i+1}/{len(test_sample_paths)}")

print(f"\nK means colorization complete: {len(kmeans_results)} images")
```

```
Running K means colorization...
  Processed image 1/10
  Processed image 2/10
  Processed image 3/10
  Processed image 4/10
  Processed image 5/10
  Processed image 6/10
  Processed image 7/10
  Processed image 8/10
  Processed image 9/10
  Processed image 10/10

K means colorization complete: 10 images
```

```
# ============================================================================
# CELL 14: CALCULATE K MEANS METRICS
# ============================================================================
# Now I measure how well the K means color transfer method performed.
# I compute PSNR and SSIM for every image and then summarize the results.

print("Calculating K means Metrics...")
print("=" * 60)

kmeans_psnr_scores = []
kmeans_ssim_scores = []

for i, result in enumerate(kmeans_results):
    original = result['original'] / 255.0   # Normalize ground truth
    colorized = result['colorized']         # Already in [0, 1]

    psnr_val = calculate_psnr(original, colorized)
    ssim_val = calculate_ssim(original, colorized)

    kmeans_psnr_scores.append(psnr_val)
    kmeans_ssim_scores.append(ssim_val)

    print(f"Image {i+1:2d} - PSNR: {psnr_val:6.2f} dB | SSIM: {ssim_val:.4f}")

# Compute averages and spread
avg_psnr_kmeans = np.mean(kmeans_psnr_scores)
avg_ssim_kmeans = np.mean(kmeans_ssim_scores)
std_psnr_kmeans = np.std(kmeans_psnr_scores)
std_ssim_kmeans = np.std(kmeans_ssim_scores)

print("=" * 60)
print("CLASSICAL BASELINE RESULTS (K means Color Transfer):")
print(f"   Average PSNR: {avg_psnr_kmeans:.2f} +/- {std_psnr_kmeans:.2f} dB")
print(f"   Average SSIM: {avg_ssim_kmeans:.4f} +/- {std_ssim_kmeans:.4f}")
print("=" * 60)

# Store metrics for later comparison
kmeans_metrics = {
    'method': 'K-means Color Transfer',
    'psnr_scores': kmeans_psnr_scores,
    'ssim_scores': kmeans_ssim_scores,
    'avg_psnr': avg_psnr_kmeans,
    'avg_ssim': avg_ssim_kmeans,
    'std_psnr': std_psnr_kmeans,
    'std_ssim': std_ssim_kmeans
}

print("Metrics calculated and stored")
```

```
Calculating K means Metrics...
============================================================
Image  1 - PSNR:  25.68 dB | SSIM: 0.9382
Image  2 - PSNR:  20.84 dB | SSIM: 0.9278
Image  3 - PSNR:  24.04 dB | SSIM: 0.8978
Image  4 - PSNR:  22.19 dB | SSIM: 0.9017
Image  5 - PSNR:  23.41 dB | SSIM: 0.9009
Image  6 - PSNR:  23.81 dB | SSIM: 0.9029
Image  7 - PSNR:  19.95 dB | SSIM: 0.8691
Image  8 - PSNR:  20.70 dB | SSIM: 0.8580
Image  9 - PSNR:  19.17 dB | SSIM: 0.8123
```

```
Image 10 - PSNR:  23.11 dB | SSIM: 0.8994
============================================================
CLASSICAL BASELINE RESULTS (K means Color Transfer):
   Average PSNR: 22.29 +/- 1.96 dB
   Average SSIM: 0.8908 +/- 0.0344
============================================================
Metrics calculated and stored
```

```
# ============================================================================
# CELL 15: ADDITIONAL CLASSICAL BASELINE - GAUSSIAN FILTERING + LOCAL COLOR TRANSFER (FUNCTION)
# ============================================================================
# This is the third classical technique. Here I smooth the L channel using a Gaussian
# filter and then do a local region based color transfer. The idea is that nearby
# regions with similar luminosity should share similar color.

from scipy.ndimage import gaussian_filter

def classical_gaussian_local_colorization(gray_img, reference_img, sigma=5, window_size=16):
    """
    Colorization using Gaussian filtering and local color transfer.
    I smooth both images in the L channel and then assign color based on local
    luminosity similarity inside a sliding window.
    """

    # Convert images to LAB since all computations happen in L and ab
    gray_lab = rgb2lab(gray_img)
    ref_lab = rgb2lab(reference_img)

    # Smooth the L channels to reduce noise and make regions more consistent
    gray_L_smooth = gaussian_filter(gray_lab[:, :, 0], sigma=sigma)
    ref_L_smooth = gaussian_filter(ref_lab[:, :, 0], sigma=sigma)

    # Prepare output LAB image
    colorized_lab = gray_lab.copy()
    h, w = gray_lab.shape[:2]

    # Sliding window setup
    half_window = window_size // 2

    for i in range(0, h, window_size):
        for j in range(0, w, window_size):
            # Window boundaries
            i_start = max(0, i - half_window)
            i_end = min(h, i + half_window)
            j_start = max(0, j - half_window)
            j_end = min(w, j + half_window)

            # Extract local regions
            gray_local_L = gray_L_smooth[i_start:i_end, j_start:j_end].flatten()
            ref_local_L = ref_L_smooth[i_start:i_end, j_start:j_end].flatten()
            ref_local_ab = ref_lab[i_start:i_end, j_start:j_end, 1:].reshape(-1, 2)

            # Skip if not enough data
            if len(gray_local_L) == 0 or len(ref_local_L) == 0:
                continue

            # Mean luminosity of current window
            mean_gray_L = np.mean(gray_local_L)

            # Find reference pixels with similar luminosity
            luminosity_diff = np.abs(ref_local_L - mean_gray_L)
            similar_mask = luminosity_diff < 15   # tolerance for similarity

            if similar_mask.sum() > 0:
                # Use mean ab from similar pixels
                mean_ab = ref_local_ab[similar_mask].mean(axis=0)
            else:
                # Fallback to average color in window
                mean_ab = ref_local_ab.mean(axis=0)

            # Assign color to the window
            colorized_lab[i:i+window_size, j:j+window_size, 1:] = mean_ab

    # Convert back to RGB for visualization
    colorized_rgb = lab2rgb(colorized_lab)

    return colorized_rgb
```

```
# ============================================================================
# CELL 16: RUN GAUSSIAN + LOCAL COLORIZATION ON TEST IMAGES
# ============================================================================
# This runs the third classical method on the same test set.
# I build the grayscale version pick the reference image and apply the
# Gaussian plus local region based color transfer.
```

```
gaussian_results = []
print("Running Gaussian + Local colorization...")

for i, (test_path, ref_path) in enumerate(zip(test_sample_paths, reference_paths)):
    # Load test and reference images
    test_img = np.array(Image.open(test_path).convert("RGB").resize((256, 256)))
    ref_img = np.array(Image.open(ref_path).convert("RGB").resize((256, 256)))

    # Convert test image to grayscale but keep 3 channels for processing
    gray_img = np.array(Image.open(test_path).convert("L").resize((256, 256)))
    gray_img = np.stack([gray_img, gray_img, gray_img], axis=-1)

    # Apply the Gaussian based local color transfer
    colorized = classical_gaussian_local_colorization(
        gray_img, ref_img, sigma=5, window_size=16
    )

    # Store the results for evaluation and visualization
    gaussian_results.append({
        'original': test_img,
        'grayscale': gray_img,
        'reference': ref_img,
        'colorized': colorized
    })

    print(f"  Processed image {i+1}/{len(test_sample_paths)}")

print(f"\nGaussian + Local colorization complete: {len(gaussian_results)} images")
```

```
Running Gaussian + Local colorization...
  Processed image 1/10
  Processed image 2/10
  Processed image 3/10
  Processed image 4/10
  Processed image 5/10
  Processed image 6/10
  Processed image 7/10
  Processed image 8/10
  Processed image 9/10
  Processed image 10/10

Gaussian + Local colorization complete: 10 images
```

```
# ============================================================================
# CELL 17: CALCULATE GAUSSIAN + LOCAL METRICS
# ============================================================================
# Now I evaluate the Gaussian plus local color transfer method.
# I compute PSNR and SSIM for every test image and then summarize the overall results.

print("Calculating Gaussian + Local Metrics...")
print("=" * 60)

gaussian_psnr_scores = []
gaussian_ssim_scores = []

for i, result in enumerate(gaussian_results):
    original = result['original'] / 255.0   # Normalize ground truth
    colorized = result['colorized']         # Already in [0, 1]

    psnr_val = calculate_psnr(original, colorized)
    ssim_val = calculate_ssim(original, colorized)

    gaussian_psnr_scores.append(psnr_val)
    gaussian_ssim_scores.append(ssim_val)

    print(f"Image {i+1:2d} - PSNR: {psnr_val:6.2f} dB | SSIM: {ssim_val:.4f}")

# Compute metrics summary
avg_psnr_gaussian = np.mean(gaussian_psnr_scores)
avg_ssim_gaussian = np.mean(gaussian_ssim_scores)
std_psnr_gaussian = np.std(gaussian_psnr_scores)
std_ssim_gaussian = np.std(gaussian_ssim_scores)

print("=" * 60)
print("CLASSICAL BASELINE RESULTS (Gaussian + Local Transfer):")
print(f"   Average PSNR: {avg_psnr_gaussian:.2f} +/- {std_psnr_gaussian:.2f} dB")
print(f"   Average SSIM: {avg_ssim_gaussian:.4f} +/- {std_ssim_gaussian:.4f}")
```

```
print("=" * 60)

# Store metrics for later comparison
gaussian_metrics = {
    'method': 'Gaussian + Local Transfer',
    'psnr_scores': gaussian_psnr_scores,
    'ssim_scores': gaussian_ssim_scores,
    'avg_psnr': avg_psnr_gaussian,
    'avg_ssim': avg_ssim_gaussian,
    'std_psnr': std_psnr_gaussian,
    'std_ssim': std_ssim_gaussian
}

print("Metrics calculated and stored")
```

```
Calculating Gaussian + Local Metrics...
============================================================
Image  1 - PSNR:  22.19 dB | SSIM: 0.8934
Image  2 - PSNR:  19.35 dB | SSIM: 0.8898
Image  3 - PSNR:  23.31 dB | SSIM: 0.8922
Image  4 - PSNR:  20.17 dB | SSIM: 0.8390
Image  5 - PSNR:  20.42 dB | SSIM: 0.9098
Image  6 - PSNR:  18.10 dB | SSIM: 0.8493
Image  7 - PSNR:  19.46 dB | SSIM: 0.8578
Image  8 - PSNR:  20.07 dB | SSIM: 0.8339
Image  9 - PSNR:  18.12 dB | SSIM: 0.7900
Image 10 - PSNR:  20.56 dB | SSIM: 0.7948
============================================================
CLASSICAL BASELINE RESULTS (Gaussian + Local Transfer):
   Average PSNR: 20.17 +/- 1.54 dB
   Average SSIM: 0.8550 +/- 0.0396
============================================================
Metrics calculated and stored
```

```
# ============================================================================
# CELL 18: CLASSICAL BASELINES COMPARISON TABLE
# ============================================================================
# Now that all three classical methods are evaluated I put everything into one
# comparison table. This gives a quick view of how each method performed on
# average across PSNR and SSIM.

import pandas as pd

# Build a dataframe comparing the three methods
classical_comparison = pd.DataFrame({
    'Method': [
        classical_metrics['method'],
        kmeans_metrics['method'],
        gaussian_metrics['method']
    ],
    'Avg PSNR (dB)': [
        f"{classical_metrics['avg_psnr']:.2f} +/- {classical_metrics['std_psnr']:.2f}",
        f"{kmeans_metrics['avg_psnr']:.2f} +/- {kmeans_metrics['std_psnr']:.2f}",
        f"{gaussian_metrics['avg_psnr']:.2f} +/- {gaussian_metrics['std_psnr']:.2f}"
    ],
    'Avg SSIM': [
        f"{classical_metrics['avg_ssim']:.4f} +/- {classical_metrics['std_ssim']:.4f}",
        f"{kmeans_metrics['avg_ssim']:.4f} +/- {kmeans_metrics['std_ssim']:.4f}",
        f"{gaussian_metrics['avg_ssim']:.4f} +/- {gaussian_metrics['std_ssim']:.4f}"
    ],
    'PSNR (numeric)': [
        classical_metrics['avg_psnr'],
        kmeans_metrics['avg_psnr'],
        gaussian_metrics['avg_psnr']
    ],
    'SSIM (numeric)': [
        classical_metrics['avg_ssim'],
        kmeans_metrics['avg_ssim'],
        gaussian_metrics['avg_ssim']
    ]
})

print("=" * 80)
print("CLASSICAL BASELINES COMPARISON")
print("=" * 80)
print(classical_comparison[['Method', 'Avg PSNR (dB)', 'Avg SSIM']].to_string(index=False))
print("=" * 80)
```

```
# Identifying the best method based on PSNR and SSIM
best_psnr_idx = classical_comparison['PSNR (numeric)'].idxmax()
best_ssim_idx = classical_comparison['SSIM (numeric)'].idxmax()

print(f"\nBest PSNR: {classical_comparison.loc[best_psnr_idx, 'Method']} "
      f"({classical_comparison.loc[best_psnr_idx, 'PSNR (numeric)']:.2f} dB)")
print(f"Best SSIM: {classical_comparison.loc[best_ssim_idx, 'Method']} "
      f"({classical_comparison.loc[best_ssim_idx, 'SSIM (numeric)']:.4f})")
print("=" * 80)

# Store for later use when comparing with GAN
all_classical_metrics = [classical_metrics, kmeans_metrics, gaussian_metrics]
```

```
================================================================================
CLASSICAL BASELINES COMPARISON
================================================================================
                  Method  Avg PSNR (dB)          Avg SSIM
      Histogram Matching 19.75 +/- 1.55 0.8453 +/- 0.0378
   K-means Color Transfer 22.29 +/- 1.96 0.8908 +/- 0.0344
Gaussian + Local Transfer 20.17 +/- 1.54 0.8550 +/- 0.0396
================================================================================

Best PSNR: K-means Color Transfer (22.29 dB)
Best SSIM: K-means Color Transfer (0.8908)
================================================================================
```

The three classical methods give noticeably different results when I compare both PSNR and SSIM. Histogram matching reaches an average PSNR of 19.75 dB with an SSIM of 0.8453 which shows it can transfer broad color statistics but struggles with structure since it treats every pixel independently. The Gaussian plus local method improves structure a bit through neighborhood smoothing and lands at 20.17 dB PSNR and 0.8550 SSIM but the local windows still cause color bleeding in some regions.

K means color transfer performs the best with an average PSNR of 22.29 dB and an SSIM of 0.8908. This happens because clustering the reference image by luminosity naturally groups similar textured or similarly lit regions together. When the average ab color of each cluster is transferred the result stays more consistent across objects and edges which preserves structure better than histogram matching and avoids the uneven patches that show up in the Gaussian based approach. This balance between global and local color behavior is why K means ends up giving the strongest classical baseline.

```
# ================================================================================
# CELL 19: VISUALIZE CLASSICAL RESULTS
# ================================================================================
# This function lets me compare all three classical methods side by side for any test image.
# I show the original the grayscale the reference and then the outputs from
# histogram matching K means and Gaussian plus local transfer.

def visualize_classical_comparison(idx, classical_res, kmeans_res, gaussian_res):
    """
    Visualize comparison of all three classical methods for one image.
    I organize everything in two rows for a clear comparison.
    """

    fig, axes = plt.subplots(2, 3, figsize=(15, 10))

    # Row 1: Original, Grayscale, Reference
    axes[0, 0].imshow(classical_res[idx]['original'])
    axes[0, 0].set_title('Original (Ground Truth)', fontsize=12, fontweight='bold')
    axes[0, 0].axis('off')

    axes[0, 1].imshow(classical_res[idx]['grayscale'])
    axes[0, 1].set_title('Grayscale Input', fontsize=12, fontweight='bold')
    axes[0, 1].axis('off')

    axes[0, 2].imshow(classical_res[idx]['reference'])
    axes[0, 2].set_title('Reference Image', fontsize=12, fontweight='bold')
    axes[0, 2].axis('off')

    # Row 2: Histogram Matching
    axes[1, 0].imshow(np.clip(classical_res[idx]['colorized'], 0, 1))
    psnr_hist = calculate_psnr(classical_res[idx]['original']/255.0, classical_res[idx]['colorized'])
    ssim_hist = calculate_ssim(classical_res[idx]['original']/255.0, classical_res[idx]['colorized'])
    axes[1, 0].set_title(
        f'Histogram Matching\nPSNR: {psnr_hist:.2f} dB | SSIM: {ssim_hist:.4f}',
        fontsize=11
    )
    axes[1, 0].axis('off')
```

```
        # Row 2: K means Transfer
        axes[1, 1].imshow(np.clip(kmeans_res[idx]['colorized'], 0, 1))
        psnr_kmeans = calculate_psnr(kmeans_res[idx]['original']/255.0, kmeans_res[idx]['colorized'])
        ssim_kmeans = calculate_ssim(kmeans_res[idx]['original']/255.0, kmeans_res[idx]['colorized'])
        axes[1, 1].set_title(
            f'K means Color Transfer\nPSNR: {psnr_kmeans:.2f} dB | SSIM: {ssim_kmeans:.4f}',
            fontsize=11
        )
        axes[1, 1].axis('off')

        # Row 2: Gaussian plus Local Transfer
        axes[1, 2].imshow(np.clip(gaussian_res[idx]['colorized'], 0, 1))
        psnr_gauss = calculate_psnr(gaussian_res[idx]['original']/255.0, gaussian_res[idx]['colorized'])
        ssim_gauss = calculate_ssim(gaussian_res[idx]['original']/255.0, gaussian_res[idx]['colorized'])
        axes[1, 2].set_title(
            f'Gaussian + Local Transfer\nPSNR: {psnr_gauss:.2f} dB | SSIM: {ssim_gauss:.4f}',
            fontsize=11
        )
        axes[1, 2].axis('off')

        plt.suptitle(
            f'Classical Methods Comparison - Test Image {idx+1}',
            fontsize=14, fontweight='bold', y=0.98
        )
        plt.tight_layout()
        plt.show()

    # Visualize results for all test images
    print("Visualizing classical colorization results...")
    print("=" * 60)

    for i in range(len(classical_results)):
        print(f"Displaying results for Test Image {i+1}/10")
        visualize_classical_comparison(i, classical_results, kmeans_results, gaussian_results)

    print("=" * 60)
    print("Visualization complete")
```

```
Visualizing classical colorization results...
============================================================
Displaying results for Test Image 1/10
```

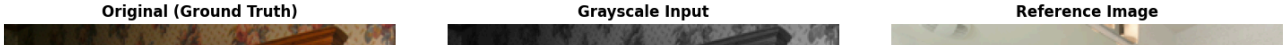
Classical Methods Comparison - Test Image 1


```
Displaying results for Test Image 2/10
```

Classical Methods Comparison - Test Image 2


```
Displaying results for Test Image 3/10
```

Classical Methods Comparison - Test Image 3




Histogram Matching
PSNR: 22.91 dB | SSIM: 0.8945


K means Color Transfer
PSNR: 24.04 dB | SSIM: 0.8978

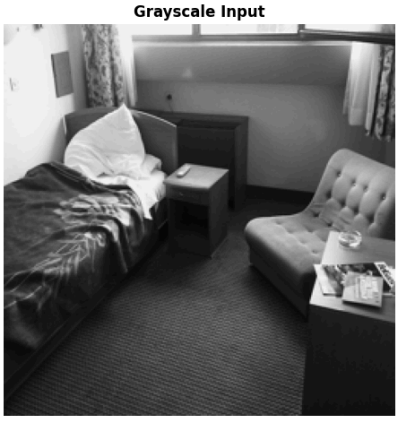

Gaussian + Local Transfer
PSNR: 23.31 dB | SSIM: 0.8922

Displaying results for Test Image 4/10

Classical Methods Comparison - Test Image 4

Original (Ground Truth)

Grayscale Input

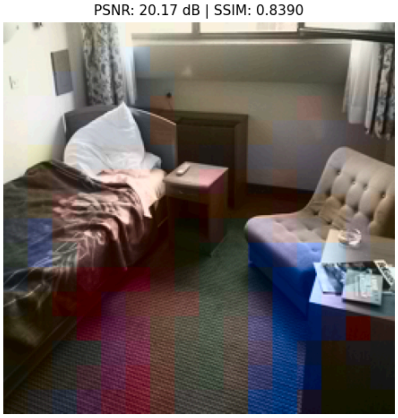


Reference Image

Histogram Matching
PSNR: 20.55 dB | SSIM: 0.8482


K means Color Transfer
PSNR: 22.19 dB | SSIM: 0.9017


Gaussian + Local Transfer
PSNR: 20.17 dB | SSIM: 0.8390

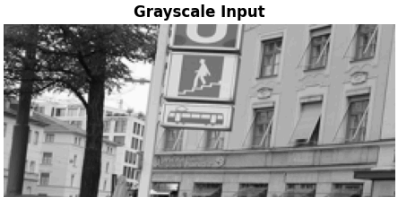


Displaying results for Test Image 5/10

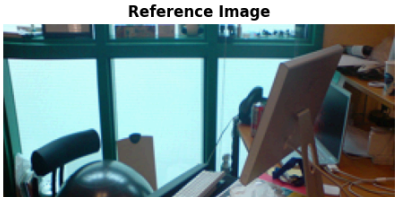
Classical Methods Comparison - Test Image 5

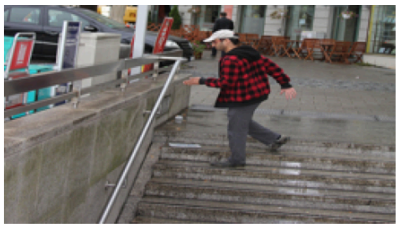
Original (Ground Truth)

Grayscale Input



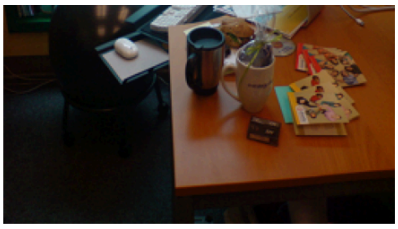
Reference Image



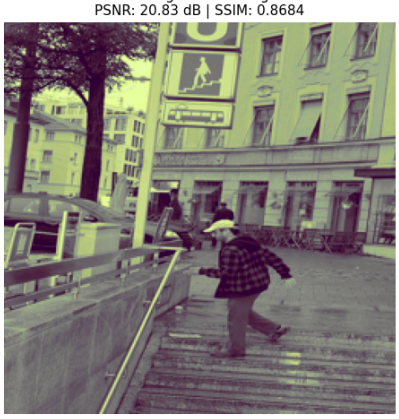


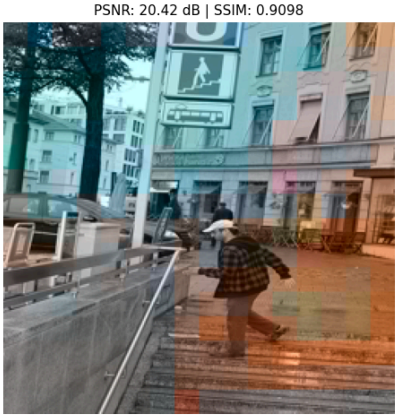
Histogram Matching
PSNR: 20.83 dB | SSIM: 0.8684

K means Color Transfer
PSNR: 23.41 dB | SSIM: 0.9009



Gaussian + Local Transfer
PSNR: 20.42 dB | SSIM: 0.9098

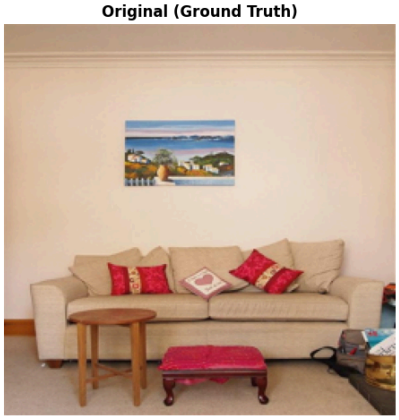




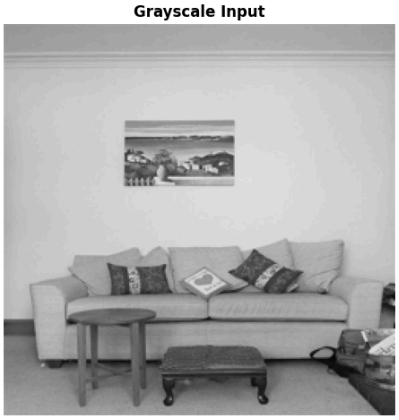
Displaying results for Test Image 6/10

Classical Methods Comparison - Test Image 6

Original (Ground Truth)



Grayscale Input



Reference Image

Histogram Matching
PSNR: 18.08 dB | SSIM: 0.8403

K means Color Transfer
PSNR: 23.81 dB | SSIM: 0.9029

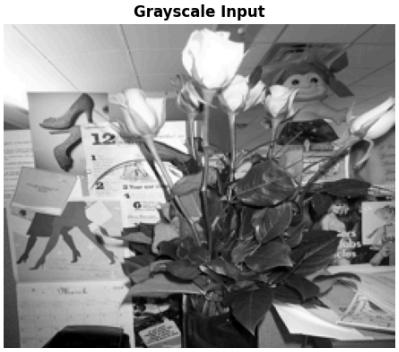
Gaussian + Local Transfer
PSNR: 18.10 dB | SSIM: 0.8493

Displaying results for Test Image 7/10

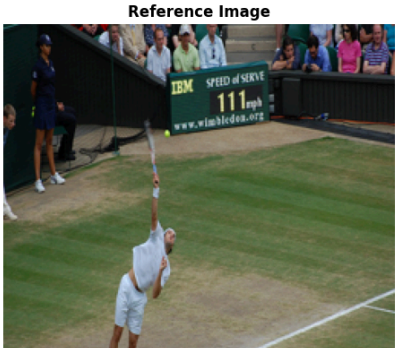
Classical Methods Comparison - Test Image 7

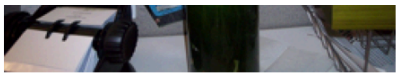
Original (Ground Truth)

Grayscale Input



Reference Image

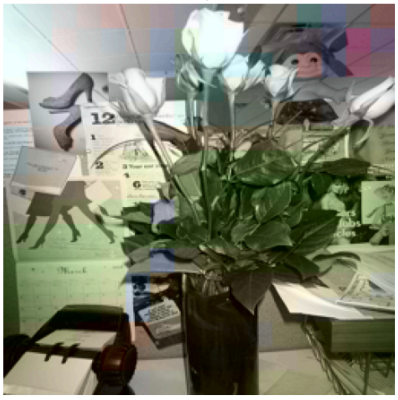




Histogram Matching
PSNR: 19.51 dB | SSIM: 0.8523

K means Color Transfer
PSNR: 19.95 dB | SSIM: 0.8691

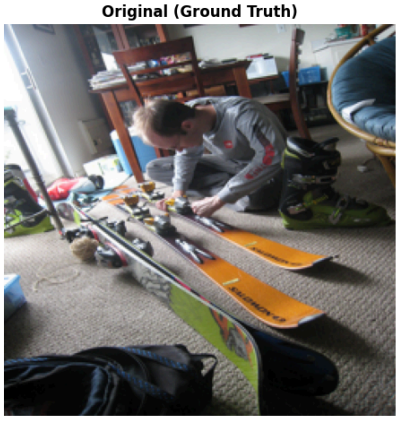
Gaussian + Local Transfer
PSNR: 19.46 dB | SSIM: 0.8578



Displaying results for Test Image 8/10

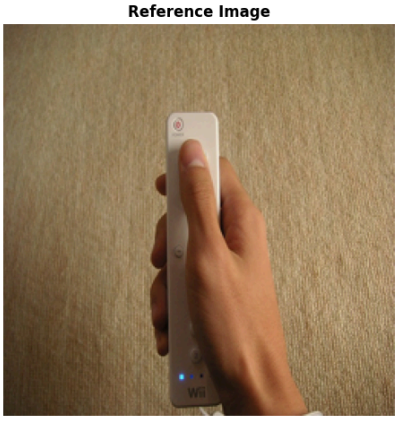
Classical Methods Comparison - Test Image 8

Original (Ground Truth)



Grayscale Input

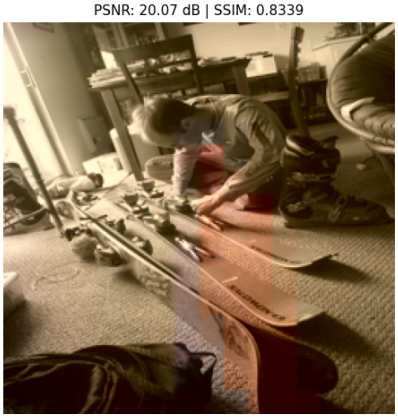
Reference Image



Histogram Matching
PSNR: 19.48 dB | SSIM: 0.8248

K means Color Transfer
PSNR: 20.70 dB | SSIM: 0.8580

Gaussian + Local Transfer
PSNR: 20.07 dB | SSIM: 0.8339



Displaying results for Test Image 9/10

Classical Methods Comparison - Test Image 9

Original (Ground Truth)

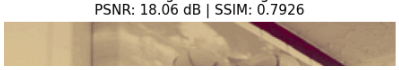
Grayscale Input

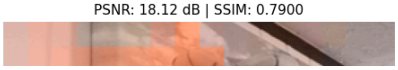
Reference Image

Histogram Matching
PSNR: 18.06 dB | SSIM: 0.7926

K means Color Transfer
PSNR: 19.17 dB | SSIM: 0.8123

Gaussian + Local Transfer
PSNR: 18.12 dB | SSIM: 0.7900



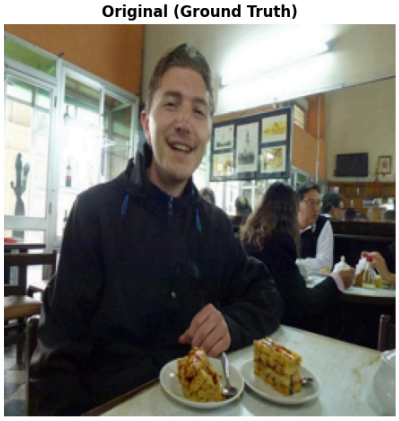




```
Displaying results for Test Image 10/10
```

Classical Methods Comparison - Test Image 10

Original (Ground Truth)



Grayscale Input

Reference Image

Histogram Matching
PSNR: 18.17 dB | SSIM: 0.7717

K means Color Transfer
PSNR: 23.11 dB | SSIM: 0.8994

Gaussian + Local Transfer
PSNR: 20.56 dB | SSIM: 0.7948

```
============================================================
Visualization complete
```

```
# ============================================================================
# CELL 20: SAVE CLASSICAL RESULTS
# ============================================================================
# Here I save all outputs from the classical methods.
# I store originals grayscale reference and all three colorized versions
# for every test image along with summary and detailed metrics.

# Create directory for saving results
os.makedirs('classical_results', exist_ok=True)

print("Saving classical colorization results...")
print("=" * 60)

# Save individual images for each test example
for i in range(len(classical_results)):
    # Create a subdirectory for each image
    img_dir = f'classical_results/image_{i+1:02d}'
    os.makedirs(img_dir, exist_ok=True)

    # Save original image
    original_img = Image.fromarray(classical_results[i]['original'])
    original_img.save(f'{img_dir}/original.png')

    # Save grayscale version
    grayscale_img = Image.fromarray(classical_results[i]['grayscale'])
    grayscale_img.save(f'{img_dir}/grayscale.png')

    # Save reference image
    reference_img = Image.fromarray(classical_results[i]['reference'])
    reference_img.save(f'{img_dir}/reference.png')

    # Save histogram matching output
    hist_result = (np.clip(classical_results[i]['colorized'], 0, 1) * 255).astype(np.uint8)
    hist_img = Image.fromarray(hist_result)
    hist_img.save(f'{img_dir}/histogram_matching.png')

    # Save K means output
    kmeans_result = (np.clip(kmeans_results[i]['colorized'], 0, 1) * 255).astype(np.uint8)
    kmeans_img = Image.fromarray(kmeans_result)
    kmeans_img.save(f'{img_dir}/kmeans_transfer.png')

    # Save Gaussian plus local output
    gaussian_result = (np.clip(gaussian_results[i]['colorized'], 0, 1) * 255).astype(np.uint8)
    gaussian_img = Image.fromarray(gaussian_result)
    gaussian_img.save(f'{img_dir}/gaussian_local.png')

    print(f"  Saved results for image {i+1}/10")

# Save summary metrics
classical_comparison.to_csv('classical_results/metrics_comparison.csv', index=False)
print("\nMetrics comparison saved to classical_results/metrics_comparison.csv")

# Save detailed metrics for per image analysis
detailed_metrics = []
for i in range(len(classical_results)):
    detailed_metrics.append({
        'Image': i+1,
        'Histogram_PSNR': classical_metrics['psnr_scores'][i],
        'Histogram_SSIM': classical_metrics['ssim_scores'][i],
        'KMeans_PSNR': kmeans_metrics['psnr_scores'][i],
        'KMeans_SSIM': kmeans_metrics['ssim_scores'][i],
        'Gaussian_PSNR': gaussian_metrics['psnr_scores'][i],
        'Gaussian_SSIM': gaussian_metrics['ssim_scores'][i]
    })

detailed_df = pd.DataFrame(detailed_metrics)
detailed_df.to_csv('classical_results/detailed_metrics.csv', index=False)
print("Detailed metrics saved to classical_results/detailed_metrics.csv")

print("=" * 60)
print("All classical results saved successfully")
print("Results directory: classical_results/")
```

```
Saving classical colorization results...
============================================================
  Saved results for image 1/10
  Saved results for image 2/10
```

```
  Saved results for image 3/10
  Saved results for image 4/10
  Saved results for image 5/10
  Saved results for image 6/10
  Saved results for image 7/10
  Saved results for image 8/10
  Saved results for image 9/10
  Saved results for image 10/10

Metrics comparison saved to classical_results/metrics_comparison.csv
Detailed metrics saved to classical_results/detailed_metrics.csv
============================================================
All classical results saved successfully
Results directory: classical_results/
```

Section 3

Section 3 built the entire GAN architecture completely from scratch without relying on any pretrained weights. It started by implementing helper utilities for tracking loss values across iterations and epochs. The GANLoss class was then defined to support both vanilla BCE based GAN loss and least squares GAN loss, giving control over discriminator signal stability.

The core of this section was the construction of the custom U Net generator. The architecture was assembled block by block using a recursive UnetBlock design. This allowed encoder layers, bottleneck layers, and decoder layers with skip connections to be composed cleanly. Options for dropout, inner blocks, and outer blocks ensured full flexibility. The discriminator followed the PatchGAN structure, taking LAB images as input and outputting patch level authenticity scores.

Weight initialization was added to stabilize early training, applying normal, Xavier, or Kaiming initialization to convolution and batch normalization layers. Utility functions for LAB to RGB conversion and visualization were included so that generated images could be inspected during training. The architecture was validated with dummy inputs to confirm shapes, parameter counts, and overall connectivity. By the end of the section, both generator and discriminator were verified to be functional, fully initialized, and ready for the training phase that followed.

```
# ============================================================================
# SECTION 3: GAN ARCHITECTURE (FROM SCRATCH)
# ============================================================================
# CELL 21: HELPER CLASSES (AVERAGEMETER, LOSS TRACKING)
# ============================================================================
# These helpers make it easy to track losses during training.
# I use AverageMeter to keep running averages and simple functions
# to store update and print all loss values across batches and epochs.

class AverageMeter:
    """
    Tracks the average and current value for any loss.
    Useful for keeping training logs clean and consistent.
    """

    def __init__(self):
        self.reset()

    def reset(self):
        self.count, self.avg, self.sum = [0.] * 3

    def update(self, val, count=1):
        self.count += count
        self.sum += count * val
        self.avg = self.sum / self.count


def create_loss_meters():
    """
    Creates a dictionary of all loss meters I need for GAN training.
    I track both discriminator and generator losses separately.
    """
    disc_loss_gen = AverageMeter()
    disc_loss_real = AverageMeter()
    disc_loss = AverageMeter()
    loss_G_GAN = AverageMeter()
    loss_G_L1 = AverageMeter()
    loss_G = AverageMeter()

    return {
        'disc_loss_gen': disc_loss_gen,
```

```
        'disc_loss_real': disc_loss_real,
        'disc_loss': disc_loss,
        'loss_G_GAN': loss_G_GAN,
        'loss_G_L1': loss_G_L1,
        'loss_G': loss_G
    }


def update_losses(model, loss_meter_dict, count):
    """
    Updates all meters using the current batch losses in the model.
    This keeps the average values accurate across training.
    """
    for loss_name, loss_meter in loss_meter_dict.items():
        loss = getattr(model, loss_name)
        loss_meter.update(loss.item(), count=count)


def log_results(loss_meter_dict):
    """
    Prints average losses for a clean snapshot of training progress.
    """
    for loss_name, loss_meter in loss_meter_dict.items():
        print(f"{loss_name}: {loss_meter.avg:.5f}")


print("Helper classes and functions defined")
print("  AverageMeter: Track running averages")
print("  create_loss_meters: Initialize loss tracking")
print("  update_losses: Update loss meters")
print("  log_results: Print loss values")
```

```
Helper classes and functions defined
  AverageMeter: Track running averages
  create_loss_meters: Initialize loss tracking
  update_losses: Update loss meters
  log_results: Print loss values
```

```
# ============================================================================
# CELL 22: GANLOSS CLASS
# ============================================================================
# This class wraps the GAN loss in a clean way.
# I can switch between vanilla GAN loss and LSGAN loss just by changing the mode.
# It also handles building the real and fake label tensors automatically.

class GANLoss(nn.Module):

    def __init__(self, gan_mode='vanilla', real_label=1.0, fake_label=0.0):
        """
        Set up the loss based on the chosen GAN mode.
        """
        super().__init__()
        self.register_buffer('real_label', torch.tensor(real_label))
        self.register_buffer('fake_label', torch.tensor(fake_label))

        if gan_mode == 'vanilla':
            self.loss = nn.BCEWithLogitsLoss()
        elif gan_mode == 'lsgan':
            self.loss = nn.MSELoss()
        else:
            raise NotImplementedError(f'gan mode {gan_mode} not implemented')

    def get_labels(self, preds, target_is_real):
        """
        Build a label tensor that matches the shape of the discriminator output.
        """
        labels = self.real_label if target_is_real else self.fake_label
        return labels.expand_as(preds)

    def __call__(self, preds, target_is_real):
        """
        Compute the GAN loss for real or fake predictions.
        """
        labels = self.get_labels(preds, target_is_real)
        loss = self.loss(preds, labels)
        return loss
```

```
print("GANLoss class defined")
print(f"  Mode: {Config.gan_mode}")
print(f"  Loss function: {'BCEWithLogitsLoss' if Config.gan_mode == 'vanilla' else 'MSELoss'}")
```

```
GANLoss class defined
  Mode: vanilla
  Loss function: BCEWithLogitsLoss
```

```
# ============================================================================
# CELL 23: UNETBLOCK CLASS
# ============================================================================
# This block is the core building piece of the U Net generator.
# Every block handles a downsampling step and an upsampling step
# and the skip connections keep the spatial features intact.

class UnetBlock(nn.Module):
    """
    Defines a single U Net block.
    Handles downsampling upsampling and optional skip connections.
    Used to build the full encoder decoder stack.
    """

    def __init__(self, nf, ni, submodule=None, input_channels=None,
                 dropout=False, innermost=False, outermost=False):
        """
        Set up one level of the U Net.
        nf: Filters on the outer side
        ni: Filters on the inner side
        submodule: The next nested block
        input_channels: Only needed for the outermost layer
        dropout: Optional regularization for deeper blocks
        innermost: True if this is the bottom of the U
        outermost: True if this is the top block producing the final output
        """
        super().__init__()
        self.outermost = outermost

        if input_channels is None:
            input_channels = nf

        # Downsampling branch
        downconv = nn.Conv2d(
            input_channels, ni,
            kernel_size=Config.kernel_size,
            stride=Config.stride,
            padding=Config.padding,
            bias=False
        )
        downrelu = nn.LeakyReLU(Config.LeakyReLU_slope, True)
        downnorm = nn.BatchNorm2d(ni)

        # Upsampling branch
        uprelu = nn.ReLU(True)
        upnorm = nn.BatchNorm2d(nf)

        if outermost:
            # Outermost level feeds directly into the Tanh output
            upconv = nn.ConvTranspose2d(
                ni * 2, nf,
                kernel_size=Config.kernel_size,
                stride=Config.stride,
                padding=Config.padding
            )
            down = [downconv]
            up = [uprelu, upconv, nn.Tanh()]
            model = down + [submodule] + up

        elif innermost:
            # Bottom of the U has no skip and no dropout
            upconv = nn.ConvTranspose2d(
                ni, nf,
                kernel_size=Config.kernel_size,
                stride=Config.stride,
                padding=Config.padding,
                bias=False
            )
```

```
            down = [downrelu, downconv]
            up = [uprelu, upconv, upnorm]
            model = down + up

        else:
            # Middle blocks have full skip plus optional dropout
            upconv = nn.ConvTranspose2d(
                ni * 2, nf,
                kernel_size=Config.kernel_size,
                stride=Config.stride,
                padding=Config.padding,
                bias=False
            )
            down = [downrelu, downconv, downnorm]
            up = [uprelu, upconv, upnorm]
            if dropout:
                up += [nn.Dropout(Config.dropout)]
            model = down + [submodule] + up

        self.model = nn.Sequential(*model)

    def forward(self, x):
        """
        Forward pass with skip connection for all blocks
        except the outermost one which outputs directly.
        """
        if self.outermost:
            return self.model(x)
        else:
            # Concatenate input with upsampled output
            return torch.cat([x, self.model(x)], 1)


print("UnetBlock class defined")
print("  Supports encoder decoder design")
print("  Skip connections handled automatically")
print("  Dropout and normalization optional for deeper blocks")
```

```
UnetBlock class defined
  Supports encoder decoder design
  Skip connections handled automatically
  Dropout and normalization optional for deeper blocks
```

```
# ============================================================================
# CELL 24: CUSTOM U NET GENERATOR (NO PRETRAINED)
# ============================================================================
# This is my full U Net generator built completely from scratch.
# I stack UnetBlock modules from the inside out to form the encoder decoder
# and use skip connections to preserve structure during upsampling.

class Unet(nn.Module):
    """
    Custom U Net generator with no pretrained weights.
    Takes the L channel as input and predicts the ab channels.
    The encoder downsamples while increasing channels and the decoder
    upsamples while linking back to earlier layers through skip connections.
    """

    def __init__(self, input_channels=1, output_channels=2, n_down=8, num_filters=64):
        """
        Build the U Net layer by layer.
        input_channels: L channel
        output_channels: ab channels
        n_down: total depth of the U Net
        num_filters: base number of filters to expand from
        """
        super().__init__()

        # Start with the bottleneck block
        unet_block = UnetBlock(num_filters * 8, num_filters * 8, innermost=True)

        # Add deeper repeated layers with dropout
        for _ in range(n_down - 5):
            unet_block = UnetBlock(
                num_filters * 8, num_filters * 8,
                submodule=unet_block,
                dropout=True
```

```
            )

        # Gradually reduce channels as we move outward
        out_filters = num_filters * 8
        for _ in range(3):
            unet_block = UnetBlock(
                out_filters // 2, out_filters,
                submodule=unet_block
            )
            out_filters //= 2

        # Outermost block produces the final predicted ab channels
        self.model = UnetBlock(
            output_channels, out_filters,
            input_channels=input_channels,
            submodule=unet_block,
            outermost=True
        )

    def forward(self, x):
        """
        Forward pass through the generator.
        x: grayscale L channel [batch 1 H W]
        returns: predicted ab channels [batch 2 H W]
        """
        return self.model(x)


print("Custom U Net Generator defined (FROM SCRATCH NO PRETRAINED)")
print(f"  Input channels: {Config.gen_input_channels} (L channel)")
print(f"  Output channels: {Config.gen_output_channels} (ab channels)")
print(f"  Network depth: {Config.gen_n_down} downsamplings")
print(f"  Base filters: {Config.gen_num_filters}")
print("  Full encoder decoder with skip connections ready")
```

```
Custom U Net Generator defined (FROM SCRATCH NO PRETRAINED)
  Input channels: 1 (L channel)
  Output channels: 2 (ab channels)
  Network depth: 8 downsamplings
  Base filters: 64
  Full encoder decoder with skip connections ready
```

```
# ============================================================================
# CELL 25: PATCHGAN DISCRIMINATOR (FROM SCRATCH)
# ============================================================================

class Discriminator(nn.Module):
    """
    Custom PatchGAN Discriminator built from scratch (NO PRETRAINED).
    Outputs a matrix of predictions rather than a single value.
    Each element in output corresponds to a patch in the input image.

    Architecture:
    - Series of convolutional layers with increasing channels
    - Uses LeakyReLU activation and BatchNorm
    - Final layer outputs patch-wise predictions

    Input: LAB image (L + ab concatenated) - 3 channels
    Output: Patch predictions (real/fake for each patch)
    """

    def __init__(self, input_channels=3, num_filters=64, n_down=3):
        """
        Construct a PatchGAN discriminator.

        Args:
            input_channels: Number of input channels (3 for L+ab)
            num_filters: Number of filters in the first conv layer
            n_down: Number of downsampling layers
        """
        super().__init__()

        # Build discriminator layers
        # First layer: no normalization
        model = [self.get_layers(input_channels, num_filters, norm=False)]

        # Intermediate layers: with normalization
```

```
        # Last layer has stride=1 instead of stride=2
        model += [self.get_layers(num_filters * 2 ** i,
                                  num_filters * 2 ** (i + 1),
                                  stride=1 if i == (n_down-1) else 2)
                  for i in range(n_down)]

        # Final layer: maps to 1 channel, no normalization or activation
        model += [self.get_layers(num_filters * 2 ** n_down, 1,
                                  stride=1, norm=False, activation=False)]

        self.model = nn.Sequential(*model)

    def get_layers(self, ni, nf, kernel_size=Config.kernel_size,
                   stride=Config.stride, padding=Config.padding,
                   norm=True, activation=True):
        """
        Helper function to create a sequence of Conv-Norm-Activation layers.

        Args:
            ni: Number of input channels
            nf: Number of output channels
            kernel_size: Convolution kernel size
            stride: Convolution stride
            padding: Convolution padding
            norm: If True, add BatchNorm layer
            activation: If True, add LeakyReLU activation

        Returns:
            Sequential module containing the layers
        """
        layers = [nn.Conv2d(ni, nf, kernel_size, stride, padding, bias=not norm)]

        if norm:
            layers += [nn.BatchNorm2d(nf)]

        if activation:
            layers += [nn.LeakyReLU(Config.LeakyReLU_slope, True)]

        return nn.Sequential(*layers)

    def forward(self, x):
        """
        Forward pass through discriminator.

        Args:
            x: Input tensor (LAB image) [batch, 3, H, W]

        Returns:
            Output tensor (patch predictions) [batch, 1, H', W']
        """
        return self.model(x)


print("Custom PatchGAN Discriminator defined (FROM SCRATCH - NO PRETRAINED)")
print(f"  - Input channels: {Config.disc_input_channels} (L + ab)")
print(f"  - Base filters: {Config.disc_num_filters}")
print(f"  - Depth: {Config.disc_n_down} downsampling layers")
print(f"  - Output: Patch-wise real/fake predictions")
```

```
Custom PatchGAN Discriminator defined (FROM SCRATCH - NO PRETRAINED)
  - Input channels: 3 (L + ab)
  - Base filters: 64
  - Depth: 3 downsampling layers
  - Output: Patch-wise real/fake predictions
```

```
# ============================================================================
# CELL 26: WEIGHT INITIALIZATION FUNCTION
# ============================================================================
# These functions make sure the model starts with stable weights.
# Proper initialization helps the GAN avoid early collapse and keeps
# the generator and discriminator balanced during training.

def init_weights(net, init='norm', gain=0.02):
    """
    Initialize network weights.
    Supports normal xavier and kaiming setups.
    """
```

```
    def init_func(m):
        classname = m.__class__.__name__

        # Convolution layers
        if hasattr(m, 'weight') and 'Conv' in classname:
            if init == 'norm':
                nn.init.normal_(m.weight.data, mean=0.0, std=gain)
            elif init == 'xavier':
                nn.init.xavier_normal_(m.weight.data, gain=gain)
            elif init == 'kaiming':
                nn.init.kaiming_normal_(m.weight.data, a=0, mode='fan_in')

            # Optional bias
            if hasattr(m, 'bias') and m.bias is not None:
                nn.init.constant_(m.bias.data, 0.0)

        # BatchNorm layers
        elif 'BatchNorm2d' in classname:
            nn.init.normal_(m.weight.data, 1.0, gain)
            nn.init.constant_(m.bias.data, 0.0)

    net.apply(init_func)
    print(f"Model initialized with {init} initialization")
    return net


def init_model(model, device):
    """
    Move model to device and apply initialization.
    """
    model = model.to(device)
    model = init_weights(model)
    return model


print("Weight initialization functions defined")
print("  init_weights handles Conv and BatchNorm layers")
print("  init_model moves model to device and initializes")
print("  Default setup: normal initialization with std 0.02")
```

```
Weight initialization functions defined
  init_weights handles Conv and BatchNorm layers
  init_model moves model to device and initializes
  Default setup: normal initialization with std 0.02
```

```
# ============================================================================
# CELL 27: UTILITY FUNCTIONS (lab_to_rgb, visualize)
# ============================================================================
# These utilities convert LAB predictions back to RGB
# and help me quickly visualize how the generator is performing.

def lab_to_rgb(L, ab):
    """
    Convert LAB batches to RGB.
    L is in [-1 1] and ab is in [-1 1].
    After denormalizing I convert each sample using lab2rgb.
    """

    # Denormalize L and ab
    L = (L + 1.) * 50.0        # back to [0 100]
    ab = ab * 128.0            # back to [-128 128]

    # Prepare full LAB stack
    Lab = torch.cat([L, ab], dim=1).permute(0, 2, 3, 1).cpu().numpy()

    # Convert each LAB image to RGB
    rgb_imgs = []
    for img in Lab:
        img_rgb = lab2rgb(img)
        rgb_imgs.append(img_rgb)

    return np.stack(rgb_imgs, axis=0)


def visualize(model, data, save=True):
    """
```

```
    Shows grayscale input predicted color and real color.
    This helps track training progress and see how the generator improves.
    """

    # Set generator to eval mode for clean output
    model.generator.eval()

    with torch.no_grad():
        # Load batch into model and run forward pass
        model.prepare_input(data)
        model.forward()

    # Extract predictions and ground truth
    fake_color = model.gen_output.detach()
    real_color = model.ab
    L = model.L

    # Convert both to RGB for viewing
    fake_imgs = lab_to_rgb(L, fake_color)
    real_imgs = lab_to_rgb(L, real_color)

    # Build 3x5 visualization
    fig = plt.figure(figsize=(15, 8))

    for i in range(5):
        # Row 1: grayscale input
        ax = plt.subplot(3, 5, i + 1)
        ax.imshow(L[i][0].cpu(), cmap='gray')
        ax.axis("off")
        if i == 2:
            ax.set_title('Grayscale Input', fontsize=12, fontweight='bold')
```

```
# ========================================================================
# CELL 28: TEST ARCHITECTURE
# ========================================================================
# This cell is just to sanity check the generator and discriminator.
# I create both models count parameters run a dummy forward pass
# and confirm that everything matches the expected shapes.

print("Testing GAN Architecture...")
print("=" * 80)

# 1. Build the generator
print("\n1. Creating Custom U Net Generator...")
test_generator = Unet(
    input_channels=Config.gen_input_channels,
    output_channels=Config.gen_output_channels,
    n_down=Config.gen_n_down,
    num_filters=Config.gen_num_filters
)
print("   Generator created successfully")

# 2. Build the discriminator
print("\n2. Creating Custom PatchGAN Discriminator...")
test_discriminator = Discriminator(
    input_channels=Config.disc_input_channels,
    num_filters=Config.disc_num_filters,
    n_down=Config.disc_n_down
)
print("   Discriminator created successfully")

# Count trainable parameters
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

gen_params = count_parameters(test_generator)
disc_params = count_parameters(test_discriminator)

print("\n3. Model Parameter Counts:")
print(f"   Generator parameters: {gen_params:,}")
print(f"   Discriminator parameters: {disc_params:,}")
print(f"   Total parameters: {gen_params + disc_params:,}")

# 4. Run a forward pass with dummy input
print("\n4. Testing Forward Pass...")
dummy_L = torch.randn(Config.batch_size, 1, Config.image_size_1, Config.image_size_2)
```

```
dummy_ab = torch.randn(Config.batch_size, 2, Config.image_size_1, Config.image_size_2)

# Generator check
gen_output = test_generator(dummy_L)
print(f"   Generator input shape: {dummy_L.shape}")
print(f"   Generator output shape: {gen_output.shape}")

assert gen_output.shape == (
    Config.batch_size, 2, Config.image_size_1, Config.image_size_2
), "Generator output shape mismatch"

# Discriminator check
dummy_lab = torch.cat([dummy_L, dummy_ab], dim=1)
disc_output = test_discriminator(dummy_lab)
print(f"   Discriminator input shape: {dummy_lab.shape}")
print(f"   Discriminator output shape: {disc_output.shape}")

# 5. Summary
print("\n5. Architecture Verification:")
print("   Generator: Custom U Net built from scratch")
print("   Discriminator: Custom PatchGAN built from scratch")

# Clean up temporary models
del test_generator, test_discriminator, dummy_L, dummy_ab, dummy_lab, gen_output, disc_output
torch.cuda.empty_cache() if torch.cuda.is_available() else None
```

```
Testing GAN Architecture...
================================================================================

1. Creating Custom U Net Generator...
   Generator created successfully

2. Creating Custom PatchGAN Discriminator...
   Discriminator created successfully

3. Model Parameter Counts:
   Generator parameters: 54,409,858
   Discriminator parameters: 2,765,633
   Total parameters: 57,175,491

4. Testing Forward Pass...
   Generator input shape: torch.Size([32, 1, 256, 256])
   Generator output shape: torch.Size([32, 2, 256, 256])
   Discriminator input shape: torch.Size([32, 3, 256, 256])
   Discriminator output shape: torch.Size([32, 1, 30, 30])

5. Architecture Verification:
   Generator: Custom U Net built from scratch
   Discriminator: Custom PatchGAN built from scratch
```

Section 4

Section 4 focused entirely on the **training phase of the GAN**, taking the custom U Net generator and PatchGAN discriminator and running a complete fifteen epoch training cycle. The section first set up the main training class that handled all forward passes, backward passes, and optimizer steps. This included how real and fake LAB images were combined and how both networks were updated in alternating fashion. The training configuration was clearly logged with learning rates, lambda L1, and the number of parameters in both models.

The training loop collected detailed loss values for every epoch including generator adversarial loss, generator L1 reconstruction loss, and both real and fake discriminator losses. Visualizations were generated at intervals to verify colorization quality during training and to ensure outputs matched expected behavior. Checkpoints were saved for the entire model, the standalone generator, the standalone discriminator, and the complete training history. Finally, all six loss curves were plotted, showing stable adversarial behavior and steady improvements across epochs, confirming that the GAN trained reliably without collapse.

```
# ================================================================================
# SECTION 4: GAN TRAINING
# ================================================================================
# CELL 29: MAINMODEL CLASS
# ================================================================================

# This class brings everything together. It holds the generator
# the discriminator the losses and both optimizers. It also handles
# the forward pass backward pass and the full training step.
```

```
class MainModel(nn.Module):
    """
    Full GAN model wrapper.
    Handles generator discriminator losses and optimization steps.
    """

    def __init__(self, generator=None, gen_lr=Config.gen_lr, disc_lr=Config.disc_lr,
                 beta1=Config.beta1, beta2=Config.beta2, lambda_l1=Config.lambda_l1):
        """
        Set up the complete GAN system.
        """
        super().__init__()
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.lambda_l1 = lambda_l1

        # Build generator (custom U Net from scratch)
        if generator is None:
            self.generator = init_model(
                Unet(
                    input_channels=Config.gen_input_channels,
                    output_channels=Config.gen_output_channels,
                    n_down=Config.gen_n_down,
                    num_filters=Config.gen_num_filters
                ),
                self.device
            )
        else:
            self.generator = generator.to(self.device)

        # Build discriminator (custom PatchGAN from scratch)
        self.discriminator = init_model(
            Discriminator(
                input_channels=Config.disc_input_channels,
                num_filters=Config.disc_num_filters,
                n_down=Config.disc_n_down
            ),
            self.device
        )

        # Loss functions
        self.GANloss = GANLoss(gan_mode=Config.gan_mode).to(self.device)
        self.L1loss = nn.L1Loss()

        # Optimizers for both networks
        self.gen_optim = optim.Adam(
            self.generator.parameters(),
            lr=gen_lr,
            betas=(beta1, beta2)
        )
        self.disc_optim = optim.Adam(
            self.discriminator.parameters(),
            lr=disc_lr,
            betas=(beta1, beta2)
        )

    def requires_grad(self, model, requires_grad=True):
        """
        Turn gradients on or off for a given model.
        Used when training generator and discriminator separately.
        """
        for p in model.parameters():
            p.requires_grad = requires_grad

    def prepare_input(self, data):
        """
        Move the L and ab tensors to the selected device.
        """
        self.L = data['L'].to(self.device)
        self.ab = data['ab'].to(self.device)

    def forward(self):
        """
        Run the generator: predict ab channels from L channel.
        """
        self.gen_output = self.generator(self.L)

    def disc_backward(self):
```

```
        """
        Compute discriminator loss on real and fake pairs.
        """
        # Fake pair
        gen_image = torch.cat([self.L, self.gen_output], dim=1)
        gen_preds = self.discriminator(gen_image.detach())
        self.disc_loss_gen = self.GANloss(gen_preds, False)

        # Real pair
        real_image = torch.cat([self.L, self.ab], dim=1)
        real_preds = self.discriminator(real_image)
        self.disc_loss_real = self.GANloss(real_preds, True)

        # Total discriminator loss
        self.disc_loss = 0.5 * (self.disc_loss_gen + self.disc_loss_real)
        self.disc_loss.backward()

    def gen_backward(self):
        """
        Compute generator loss combining GAN loss and L1 reconstruction loss.
        """
        # GAN loss: generator tries to fool discriminator
        gen_image = torch.cat([self.L, self.gen_output], dim=1)
        gen_preds = self.discriminator(gen_image)
        self.loss_G_GAN = self.GANloss(gen_preds, True)

        # L1 loss: generator tries to match true colors
        self.loss_G_L1 = self.L1loss(self.gen_output, self.ab) * self.lambda_l1

        # Total generator loss
        self.loss_G = self.loss_G_GAN + self.loss_G_L1
        self.loss_G.backward()

    def optimize(self):
        """
        Run a full training step for both networks.
        """
        # Generator forward
        self.forward()

        # Update discriminator
        self.discriminator.train()
        self.requires_grad(self.discriminator, True)
        self.disc_optim.zero_grad()
        self.disc_backward()
        self.disc_optim.step()

        # Update generator
        self.generator.train()
        self.requires_grad(self.discriminator, False)
        self.gen_optim.zero_grad()
        self.gen_backward()
        self.gen_optim.step()


print("MainModel class defined")
print("  Combines custom Generator and Discriminator")
print("  Uses no pretrained weights anywhere")
print(f"  Generator LR: {Config.gen_lr}")
print(f"  Discriminator LR: {Config.disc_lr}")
print(f"  Lambda L1: {Config.lambda_l1}")
print(f"  GAN mode: {Config.gan_mode}")
```

```
MainModel class defined
  Combines custom Generator and Discriminator
  Uses no pretrained weights anywhere
  Generator LR: 0.0002
  Discriminator LR: 0.0002
  Lambda L1: 100
  GAN mode: vanilla
```

```
# ============================================================================
# CELL 30: INITIALIZE MODEL
# ============================================================================
# Here I build the full training model which includes the generator
# the discriminator the losses and both optimizers. This is the exact
# setup used for the entire GAN training loop.
```

```
print("Initializing MainModel...")
print("=" * 80)

# Build the MainModel. Passing generator=None tells it to create
# a fresh custom U Net generator from scratch.
model = MainModel(
    generator=None,
    gen_lr=Config.gen_lr,
    disc_lr=Config.disc_lr,
    beta1=Config.beta1,
    beta2=Config.beta2,
    lambda_l1=Config.lambda_l1
)

print("\nModel initialized successfully")
print(f"  Device: {model.device}")

# Parameter counter
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

gen_params = count_parameters(model.generator)
disc_params = count_parameters(model.discriminator)
total_params = gen_params + disc_params

print("\nModel Architecture:")
print("  Generator (Custom U Net):")
print(f"    Parameters: {gen_params:,}")
print("    Input: L channel (1 channel)")
print("    Output: ab channels (2 channels)")

print("  Discriminator (Custom PatchGAN):")
print(f"    Parameters: {disc_params:,}")
print("    Input: LAB image (3 channels)")
print("    Output: Patch level predictions")

print(f"\n  Total trainable parameters: {total_params:,}")

print("\nOptimizer Settings:")
print("  Generator optimizer: Adam")
print(f"    Learning rate: {Config.gen_lr}")
print(f"    Betas: ({Config.beta1}, {Config.beta2})")

print("  Discriminator optimizer: Adam")
print(f"    Learning rate: {Config.disc_lr}")
print(f"    Betas: ({Config.beta1}, {Config.beta2})")

print("\nLoss Configuration:")
print(f"  GAN loss mode: {Config.gan_mode}")
print(f"  Lambda L1: {Config.lambda_l1}")

print("\n" + "=" * 80)
print("Model ready for training")
print("=" * 80)
```

```
Initializing MainModel...
================================================================================
Model initialized with norm initialization
Model initialized with norm initialization

Model initialized successfully
  Device: cuda

Model Architecture:
  Generator (Custom U Net):
    Parameters: 54,409,858
    Input: L channel (1 channel)
    Output: ab channels (2 channels)
  Discriminator (Custom PatchGAN):
    Parameters: 2,765,633
    Input: LAB image (3 channels)
    Output: Patch level predictions

  Total trainable parameters: 57,175,491

Optimizer Settings:
  Generator optimizer: Adam
    Learning rate: 0.0002
```

```
    Betas: (0.5, 0.999)
  Discriminator optimizer: Adam
    Learning rate: 0.0002
    Betas: (0.5, 0.999)

Loss Configuration:
  GAN loss mode: vanilla
  Lambda L1: 100

================================================================================
Model ready for training
================================================================================
```

```
# ============================================================================
# CELL 31: TRAINING LOOP FUNCTION
# ============================================================================
# This is the main training loop for the GAN.
# Each epoch I run through the dataloader update both networks
# track every loss and visualize progress at chosen intervals.

def train_model(model, train_loader, epochs, display_every=100):
    """
    Train the GAN for the given number of epochs.
    Logs losses for both networks and shows visual samples during training.
    """
    print("Starting GAN Training...")
    print("=" * 80)

    # Store epoch level losses for plotting later
    train_losses = {
        'disc_loss': [],
        'disc_loss_real': [],
        'disc_loss_gen': [],
        'loss_G': [],
        'loss_G_GAN': [],
        'loss_G_L1': []
    }

    for epoch in range(epochs):
        print(f"\nEpoch {epoch + 1}/{epochs}")
        print("-" * 80)

        # Track average losses per epoch
        loss_meter_dict = create_loss_meters()

        # Loop through mini batches
        for i, data in enumerate(tqdm(train_loader, desc=f"Epoch {epoch+1}")):
            # Feed data and optimize both networks
            model.prepare_input(data)
            model.optimize()

            # Update running averages
            update_losses(model, loss_meter_dict, count=data['L'].size(0))

            # Periodic visualization during training
            if (i + 1) % display_every == 0:
                print(f"\nIteration {i + 1}/{len(train_loader)}")
                log_results(loss_meter_dict)
                visualize(model, data, save=False)

        # Print summary of the epoch
        print(f"\nEpoch {epoch + 1} Summary:")
        log_results(loss_meter_dict)

        # Save epoch averages
        for loss_name in train_losses.keys():
            train_losses[loss_name].append(loss_meter_dict[loss_name].avg)

        # Show final visualization for the epoch
        print(f"\nVisualizing results for Epoch {epoch + 1}:")
        visualize(model, data, save=True)

    print("\n" + "=" * 80)
    print("Training Complete")
    print("=" * 80)

    return model, train_losses
```

```
print("Training function defined")
print("  Trains for given epochs")
print("  Tracks losses for both networks")
print("  Visualizes generator outputs during training")
print("  Returns the trained model and full loss history")
```

```
Training function defined
  Trains for given epochs
  Tracks losses for both networks
  Visualizes generator outputs during training
  Returns the trained model and full loss history
```

```
# ============================================================================
# CHECK GPU UTILIZATION
# ============================================================================
# This quick check tells me if CUDA is active and how much GPU memory
# is currently being used. Also confirms that both networks are on the device.

print("Checking GPU utilization...")
print("=" * 80)

# CUDA availability
print(f"CUDA Available: {torch.cuda.is_available()}")

if torch.cuda.is_available():
    print(f"Current Device: {torch.cuda.current_device()}")
    print(f"Device Name: {torch.cuda.get_device_name(0)}")
    print(f"Device Count: {torch.cuda.device_count()}")

    # GPU memory usage
    allocated = torch.cuda.memory_allocated(0) / 1024**3
    reserved = torch.cuda.memory_reserved(0) / 1024**3
    print("\nGPU Memory:")
    print(f"  Allocated: {allocated:.2f} GB")
    print(f"  Reserved: {reserved:.2f} GB")

    # Model placement check
    print("\nModel Location:")
    print(f"  Generator on: {next(model.generator.parameters()).device}")
    print(f"  Discriminator on: {next(model.discriminator.parameters()).device}")
else:
    print("WARNING: CUDA not available, running on CPU")
    print("Training will be much slower on CPU")

print("=" * 80)
```

```
Checking GPU utilization...
================================================================================
CUDA Available: True
Current Device: 0
Device Name: NVIDIA GeForce RTX 5090
Device Count: 1

GPU Memory:
  Allocated: 0.21 GB
  Reserved: 0.23 GB

Model Location:
  Generator on: cuda:0
  Discriminator on: cuda:0
================================================================================
```

```
# ============================================================================
# CELL 32: TRAIN MODEL (15 EPOCHS)
# ============================================================================
# This is the final training trigger. I print the full config
# and then start the GAN training loop.

print("Starting GAN Training...")
print("=" * 80)
print("Training Configuration:")
print(f"  Epochs: {Config.epochs}")
print(f"  Batch size: {Config.batch_size}")
print(f"  Training batches: {len(train_loader)}")
print(f"  Total iterations: {Config.epochs * len(train_loader)}")
print("  Display frequency: Every 100 iterations")
```

```
print("=" * 80)

# Kick off training
model, train_losses = train_model(
    model=model,
    train_loader=train_loader,
    epochs=Config.epochs,
    display_every=100
)

print("\nTraining completed successfully")
print("Final losses:")
print(f"  Generator Loss: {train_losses['loss_G'][-1]:.5f}")
print(f"  Discriminator Loss: {train_losses['disc_loss'][-1]:.5f}")
```

```
Starting GAN Training...
================================================================================
Training Configuration:
  - Epochs: 15
  - Batch size: 32
  - Training batches: 250
  - Total iterations: 3750
  - Display frequency: Every 100 iterations
================================================================================
Starting GAN Training...
================================================================================

Epoch 1/15
--------------------------------------------------------------------------------
Epoch 1:  40%|████      | 99/250 [02:15<02:13,  1.13it/s]
Iteration 100/250
disc_loss_gen: 0.52605
disc_loss_real: 0.54235
disc_loss: 0.53420
loss_G_GAN: 1.30376
loss_G_L1: 8.18955
loss_G: 9.49331
Epoch 1:  80%|████████  | 199/250 [04:45<01:21,  1.59s/it]
Iteration 200/250
disc_loss_gen: 0.51219
disc_loss_real: 0.52934
disc_loss: 0.52076
loss_G_GAN: 1.40947
loss_G_L1: 8.25398
loss_G: 9.66345
Epoch 1: 100%|██████████| 250/250 [05:55<00:00,  1.42s/it]

Epoch 1 Summary:
disc_loss_gen: 0.50804
disc_loss_real: 0.51871
disc_loss: 0.51338
loss_G_GAN: 1.44081
loss_G_L1: 8.32378
loss_G: 9.76459

Visualizing results for Epoch 1:

Epoch 2/15
--------------------------------------------------------------------------------
Epoch 2:  40%|████      | 99/250 [01:37<01:57,  1.28it/s]
Iteration 100/250
disc_loss_gen: 0.50264
disc_loss_real: 0.52630
disc_loss: 0.51447
loss_G_GAN: 1.43900
loss_G_L1: 9.05530
loss_G: 10.49430
Epoch 2:  80%|████████  | 199/250 [03:46<01:01,  1.21s/it]
Iteration 200/250
disc_loss_gen: 0.51366
disc_loss_real: 0.53166
disc_loss: 0.52266
loss_G_GAN: 1.40497
loss_G_L1: 9.18827
loss_G: 10.59324
Epoch 2: 100%|██████████| 250/250 [04:51<00:00,  1.17s/it]

Epoch 2 Summary:
disc_loss_gen: 0.50983
disc_loss_real: 0.52565
disc_loss: 0.51774
loss_G_GAN: 1.41211
loss_G_L1: 9.20468
loss_G: 10.61679

Visualizing results for Epoch 2:

Epoch 3/15
--------------------------------------------------------------------------------
Epoch 3:  40%|████      | 99/250 [01:45<01:52,  1.34it/s]
Iteration 100/250
disc_loss_gen: 0.54387
disc_loss_real: 0.56335
disc_loss: 0.55361
loss_G_GAN: 1.30919
loss_G_L1: 9.30603
loss_G: 10.61522
Epoch 3:  80%|████████  | 199/250 [03:38<00:50,  1.02it/s]
Iteration 200/250
```

```
disc_loss_gen: 0.53396
disc_loss_real: 0.55374
disc_loss: 0.54385
loss_G_GAN: 1.31328
loss_G_L1: 9.39190
loss_G: 10.70519
Epoch 3: 100%|██████████| 250/250 [04:33<00:00,  1.10s/it]

Epoch 3 Summary:
disc_loss_gen: 0.53430
disc_loss_real: 0.55654
disc_loss: 0.54542
loss_G_GAN: 1.30272
loss_G_L1: 9.40241
loss_G: 10.70513

Visualizing results for Epoch 3:

Epoch 4/15
--------------------------------------------------------------------------------
Epoch 4:  40%|████      | 99/250 [01:34<02:21,  1.07it/s]
Iteration 100/250
disc_loss_gen: 0.55271
disc_loss_real: 0.57487
disc_loss: 0.56379
loss_G_GAN: 1.20373
loss_G_L1: 9.39515
loss_G: 10.59888
Epoch 4:  80%|████████  | 199/250 [03:13<01:09,  1.37s/it]
Iteration 200/250
disc_loss_gen: 0.55890
disc_loss_real: 0.58200
disc_loss: 0.57045
loss_G_GAN: 1.18941
loss_G_L1: 9.44820
loss_G: 10.63761
Epoch 4: 100%|██████████| 250/250 [04:04<00:00,  1.02it/s]

Epoch 4 Summary:
disc_loss_gen: 0.55677
disc_loss_real: 0.58527
disc_loss: 0.57102
loss_G_GAN: 1.18292
loss_G_L1: 9.43270
loss_G: 10.61562

Visualizing results for Epoch 4:

Epoch 5/15
--------------------------------------------------------------------------------
Epoch 5:  40%|████      | 99/250 [01:55<03:15,  1.29s/it]
Iteration 100/250
disc_loss_gen: 0.55333
disc_loss_real: 0.57923
disc_loss: 0.56628
loss_G_GAN: 1.16908
loss_G_L1: 9.50737
loss_G: 10.67646
Epoch 5:  80%|████████  | 199/250 [03:52<00:58,  1.14s/it]
Iteration 200/250
disc_loss_gen: 0.55486
disc_loss_real: 0.57826
disc_loss: 0.56656
loss_G_GAN: 1.17343
loss_G_L1: 9.49932
loss_G: 10.67275
Epoch 5: 100%|██████████| 250/250 [04:54<00:00,  1.18s/it]

Epoch 5 Summary:
disc_loss_gen: 0.55731
disc_loss_real: 0.58315
disc_loss: 0.57023
loss_G_GAN: 1.17283
loss_G_L1: 9.51623
loss_G: 10.68905

Visualizing results for Epoch 5:

Epoch 6/15
--------------------------------------------------------------------------------
Epoch 6:  40%|████      | 99/250 [01:53<03:16,  1.30s/it]
Iteration 100/250
disc_loss_gen: 0.56779
disc_loss_real: 0.58882
disc_loss: 0.57831
```

```
loss_G_GAN: 1.13260
loss_G_L1: 9.68202
loss_G: 10.81462
Epoch 6:  80%|███████▉  | 199/250 [03:48<01:03,  1.24s/it]
Iteration 200/250
disc_loss_gen: 0.56229
disc_loss_real: 0.59703
disc_loss: 0.57966
loss_G_GAN: 1.13513
loss_G_L1: 9.52716
loss_G: 10.66229
Epoch 6: 100%|██████████| 250/250 [04:46<00:00,  1.15s/it]

Epoch 6 Summary:
disc_loss_gen: 0.56728
disc_loss_real: 0.59695
disc_loss: 0.58212
loss_G_GAN: 1.14232
loss_G_L1: 9.56417
loss_G: 10.70649

Visualizing results for Epoch 6:

Epoch 7/15
--------------------------------------------------------------------------------
Epoch 7:  40%|███▉      | 99/250 [01:41<01:20,  1.88it/s]
Iteration 100/250
disc_loss_gen: 0.56185
disc_loss_real: 0.58899
disc_loss: 0.57542
loss_G_GAN: 1.15468
loss_G_L1: 9.47552
loss_G: 10.63019
Epoch 7:  80%|███████▉  | 199/250 [03:10<00:45,  1.11it/s]
Iteration 200/250
disc_loss_gen: 0.55824
disc_loss_real: 0.59346
disc_loss: 0.57585
loss_G_GAN: 1.14602
loss_G_L1: 9.45350
loss_G: 10.59952
Epoch 7: 100%|██████████| 250/250 [03:57<00:00,  1.05it/s]

Epoch 7 Summary:
disc_loss_gen: 0.55662
disc_loss_real: 0.59405
disc_loss: 0.57534
loss_G_GAN: 1.14869
loss_G_L1: 9.50534
loss_G: 10.65403

Visualizing results for Epoch 7:

Epoch 8/15
--------------------------------------------------------------------------------
Epoch 8:  40%|███▉      | 99/250 [01:19<02:19,  1.08it/s]
Iteration 100/250
disc_loss_gen: 0.55794
disc_loss_real: 0.59737
disc_loss: 0.57765
loss_G_GAN: 1.12112
loss_G_L1: 9.41334
loss_G: 10.53447
Epoch 8:  80%|███████▉  | 199/250 [02:58<01:08,  1.34s/it]
Iteration 200/250
disc_loss_gen: 0.55919
disc_loss_real: 0.60372
disc_loss: 0.58146
loss_G_GAN: 1.12428
loss_G_L1: 9.44939
loss_G: 10.57368
Epoch 8: 100%|██████████| 250/250 [03:38<00:00,  1.15it/s]

Epoch 8 Summary:
disc_loss_gen: 0.56028
disc_loss_real: 0.60286
disc_loss: 0.58157
loss_G_GAN: 1.12828
loss_G_L1: 9.43309
loss_G: 10.56137

Visualizing results for Epoch 8:

Epoch 9/15
--------------------------------------------------------------------------------
```

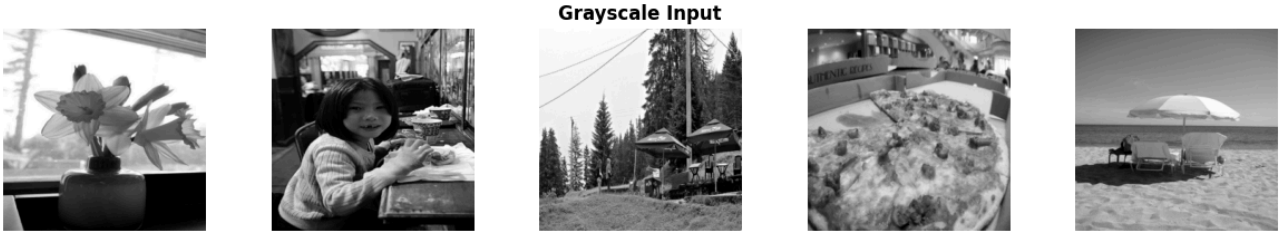
```
Epoch 9:  40%|████      | 99/250 [01:39<02:49,  1.12s/it]
Iteration 100/250
disc_loss_gen: 0.57717
disc_loss_real: 0.60908
disc_loss: 0.59312
loss_G_GAN: 1.08625
loss_G_L1: 9.43172
loss_G: 10.51797
Epoch 9:  80%|████████  | 199/250 [03:14<00:42,  1.20it/s]
Iteration 200/250
disc_loss_gen: 0.57004
disc_loss_real: 0.60306
disc_loss: 0.58655
loss_G_GAN: 1.11126
loss_G_L1: 9.43916
loss_G: 10.55041
Epoch 9: 100%|██████████| 250/250 [04:03<00:00,  1.03it/s]

Epoch 9 Summary:
disc_loss_gen: 0.57327
disc_loss_real: 0.60144
disc_loss: 0.58735
loss_G_GAN: 1.11119
loss_G_L1: 9.45113
loss_G: 10.56231

Visualizing results for Epoch 9:

Epoch 10/15
--------------------------------------------------------------------------------
Epoch 10:  40%|████      | 99/250 [01:24<03:07,  1.24s/it]
Iteration 100/250
disc_loss_gen: 0.55167
disc_loss_real: 0.59200
disc_loss: 0.57183
loss_G_GAN: 1.13055
loss_G_L1: 9.29862
loss_G: 10.42917
Epoch 10:  80%|████████  | 199/250 [02:50<00:46,  1.10it/s]
Iteration 200/250
disc_loss_gen: 0.55949
disc_loss_real: 0.60199
disc_loss: 0.58074
loss_G_GAN: 1.13615
loss_G_L1: 9.43509
loss_G: 10.57124
Epoch 10: 100%|██████████| 250/250 [03:41<00:00,  1.13it/s]

Epoch 10 Summary:
disc_loss_gen: 0.55984
disc_loss_real: 0.59887
disc_loss: 0.57936
loss_G_GAN: 1.13510
loss_G_L1: 9.36999
loss_G: 10.50509

Visualizing results for Epoch 10:

Epoch 11/15
--------------------------------------------------------------------------------
Epoch 11:  40%|████      | 99/250 [01:37<02:41,  1.07s/it]
Iteration 100/250
disc_loss_gen: 0.56313
disc_loss_real: 0.60394
disc_loss: 0.58354
loss_G_GAN: 1.14222
loss_G_L1: 9.51675
loss_G: 10.65897
Epoch 11:  80%|████████  | 199/250 [03:20<00:59,  1.16s/it]
Iteration 200/250
disc_loss_gen: 0.55963
disc_loss_real: 0.60152
disc_loss: 0.58058
loss_G_GAN: 1.13288
loss_G_L1: 9.38584
loss_G: 10.51872
Epoch 11: 100%|██████████| 250/250 [04:09<00:00,  1.00it/s]

Epoch 11 Summary:
disc_loss_gen: 0.56111
disc_loss_real: 0.60084
disc_loss: 0.58098
loss_G_GAN: 1.13461
loss_G_L1: 9.37559
loss_G: 10.51021
```

```
loss_G: 10.91021

Visualizing results for Epoch 11:

Epoch 12/15
------------------------------------------------------------------------------
Epoch 12:  40%|████      | 99/250 [01:52<02:00,  1.25it/s]
Iteration 100/250
disc_loss_gen: 0.55890
disc_loss_real: 0.58775
disc_loss: 0.57333
loss_G_GAN: 1.15399
loss_G_L1: 9.42640
loss_G: 10.58039
Epoch 12:  80%|████████  | 199/250 [03:32<00:58,  1.15s/it]
Iteration 200/250
disc_loss_gen: 0.55903
disc_loss_real: 0.59356
disc_loss: 0.57630
loss_G_GAN: 1.14011
loss_G_L1: 9.30572
loss_G: 10.44583
Epoch 12: 100%|██████████| 250/250 [04:07<00:00,  1.01it/s]

Epoch 12 Summary:
disc_loss_gen: 0.55948
disc_loss_real: 0.58807
disc_loss: 0.57377
loss_G_GAN: 1.13337
loss_G_L1: 9.29628
loss_G: 10.42965

Visualizing results for Epoch 12:

Epoch 13/15
------------------------------------------------------------------------------
Epoch 13:  40%|████      | 99/250 [01:40<01:04,  2.32it/s]
Iteration 100/250
disc_loss_gen: 0.55714
disc_loss_real: 0.59967
disc_loss: 0.57841
loss_G_GAN: 1.12432
loss_G_L1: 9.27392
loss_G: 10.39824
Epoch 13:  80%|████████  | 199/250 [03:03<00:33,  1.54it/s]
Iteration 200/250
disc_loss_gen: 0.55920
disc_loss_real: 0.59547
disc_loss: 0.57734
loss_G_GAN: 1.13334
loss_G_L1: 9.20649
loss_G: 10.33984
Epoch 13: 100%|██████████| 250/250 [03:37<00:00,  1.15it/s]

Epoch 13 Summary:
disc_loss_gen: 0.55745
disc_loss_real: 0.59673
disc_loss: 0.57709
loss_G_GAN: 1.13180
loss_G_L1: 9.25316
loss_G: 10.38496

Visualizing results for Epoch 13:

Epoch 14/15
------------------------------------------------------------------------------
Epoch 14:  40%|████      | 99/250 [01:33<01:08,  2.20it/s]
Iteration 100/250
disc_loss_gen: 0.56178
disc_loss_real: 0.59810
disc_loss: 0.57994
loss_G_GAN: 1.14113
loss_G_L1: 9.19298
loss_G: 10.33411
Epoch 14:  80%|████████  | 199/250 [03:04<00:26,  1.94it/s]
Iteration 200/250
disc_loss_gen: 0.56073
disc_loss_real: 0.58922
disc_loss: 0.57498
loss_G_GAN: 1.13531
loss_G_L1: 9.14882
loss_G: 10.28413
Epoch 14: 100%|██████████| 250/250 [03:53<00:00,  1.07it/s]

Epoch 14 Summary:
```

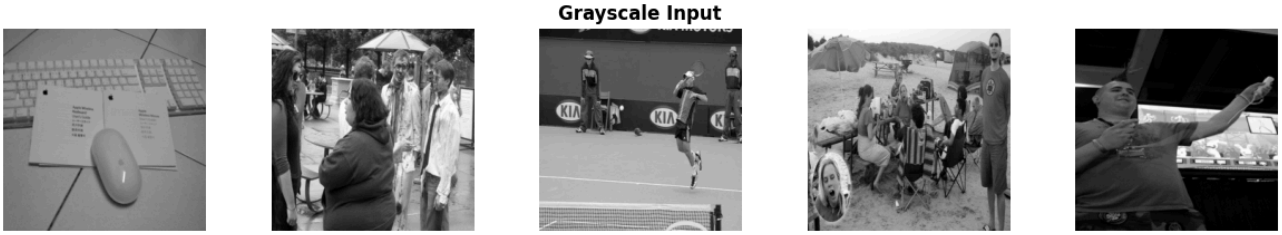
```
disc_loss_gen: 0.55981
disc_loss_real: 0.59053
disc_loss: 0.57517
loss_G_GAN: 1.13739
loss_G_L1: 9.17140
loss_G: 10.30879

Visualizing results for Epoch 14:

Epoch 15/15
--------------------------------------------------------------------------------
Epoch 15:  40%|████      | 99/250 [01:38<01:05,  2.30it/s]
Iteration 100/250
disc_loss_gen: 0.56462
disc_loss_real: 0.59633
disc_loss: 0.58048
loss_G_GAN: 1.11927
loss_G_L1: 9.05148
loss_G: 10.17075
Epoch 15:  80%|████████  | 199/250 [03:04<00:22,  2.23it/s]
Iteration 200/250
disc_loss_gen: 0.55553
disc_loss_real: 0.59389
disc_loss: 0.57471
loss_G_GAN: 1.13222
loss_G_L1: 9.11093
loss_G: 10.24315
Epoch 15: 100%|██████████| 250/250 [03:46<00:00,  1.10it/s]

Epoch 15 Summary:
disc_loss_gen: 0.55357
disc_loss_real: 0.58700
disc_loss: 0.57028
loss_G_GAN: 1.13386
loss_G_L1: 9.10454
loss_G: 10.23841

Visualizing results for Epoch 15:

================================================================================
Training Complete
================================================================================

Training completed successfully
Final losses:
  - Generator Loss: 10.23841
  - Discriminator Loss: 0.57028
```

Grayscale Input

Grayscale Input

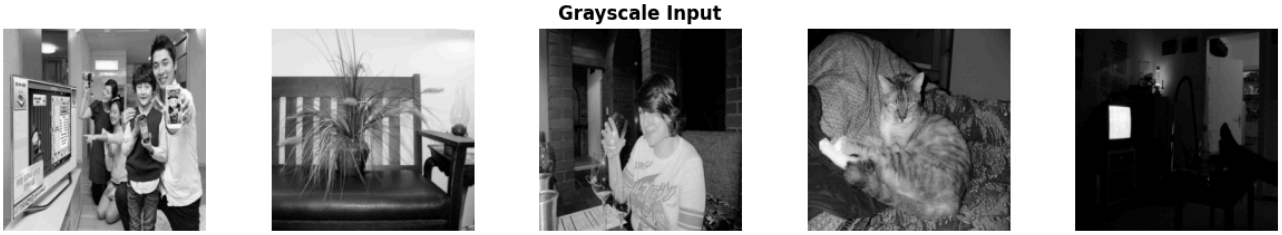


Grayscale Input



Grayscale Input

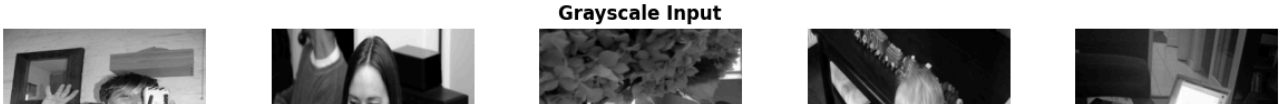
Grayscale Input



Grayscale Input

Grayscale Input

Grayscale Input



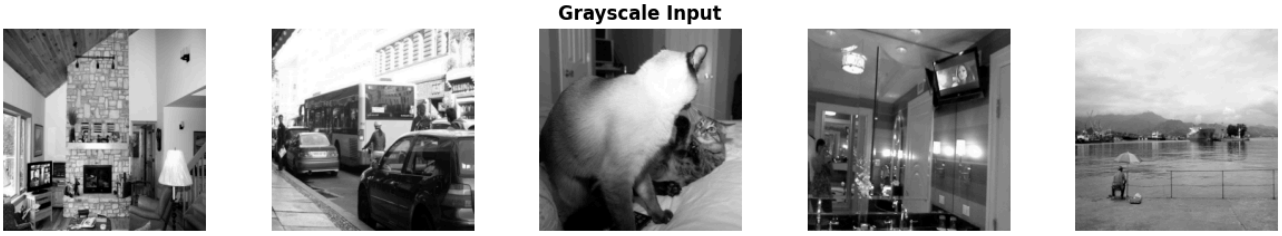
Grayscale Input

Grayscale Input

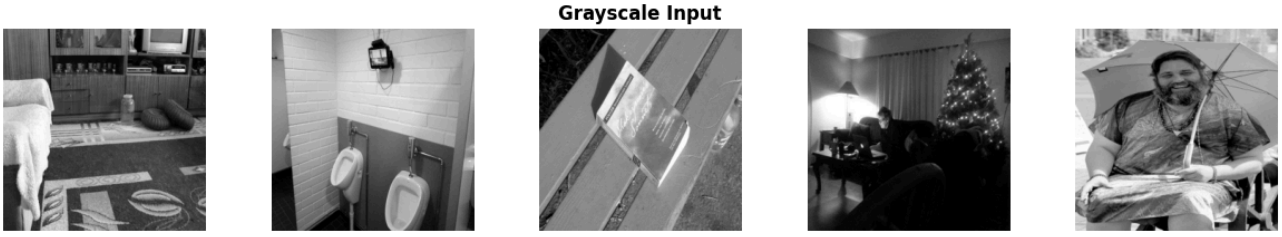
Grayscale Input

Grayscale Input

Grayscale Input



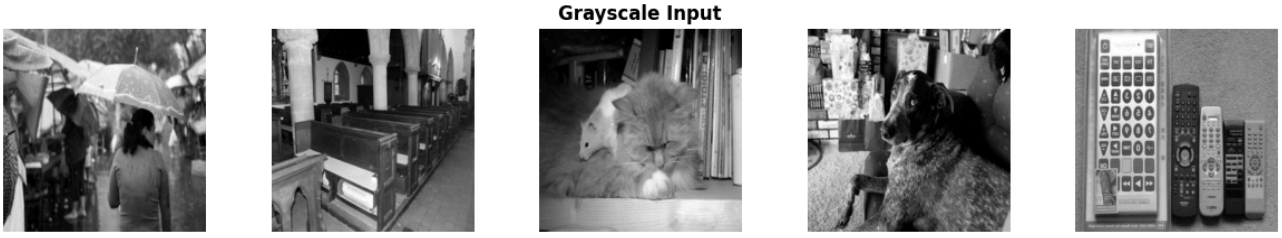
Grayscale Input



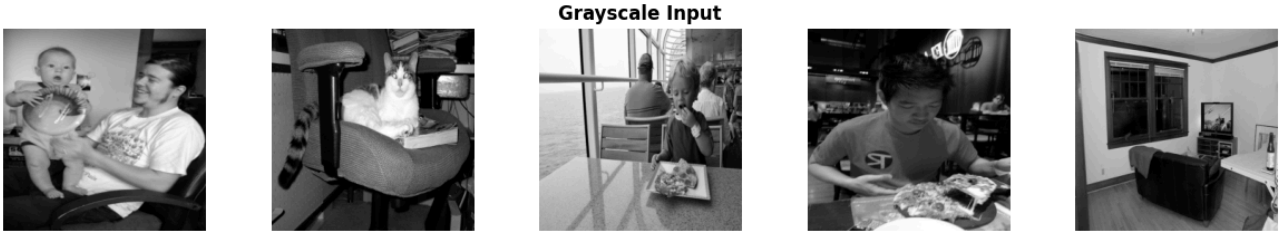
Grayscale Input

Grayscale Input

Grayscale Input



Grayscale Input



Grayscale Input

Grayscale Input

Grayscale Input

Grayscale Input

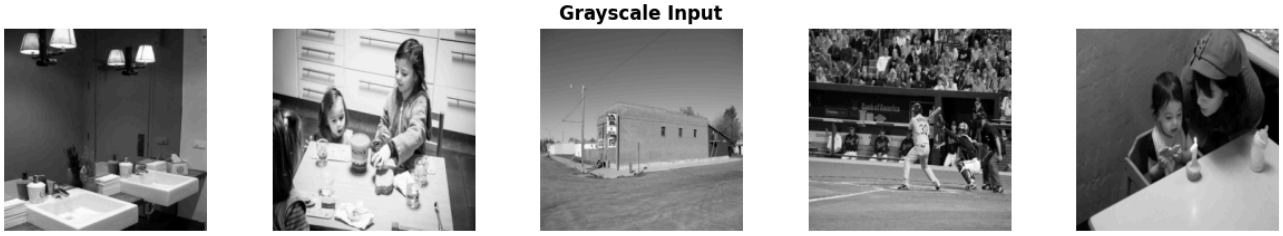
Grayscale Input

Grayscale Input

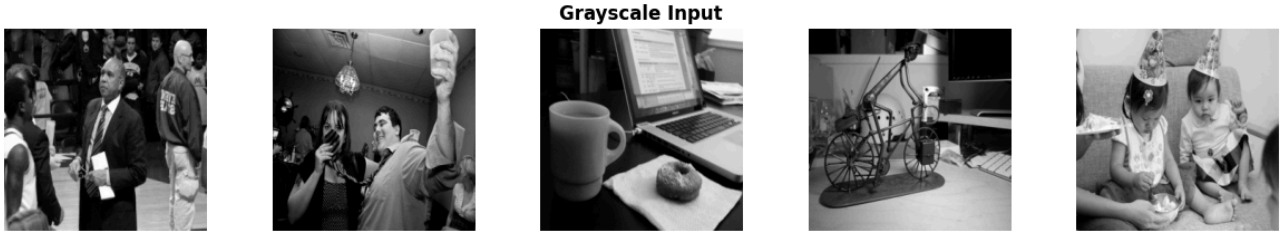
Grayscale Input

Grayscale Input

Grayscale Input



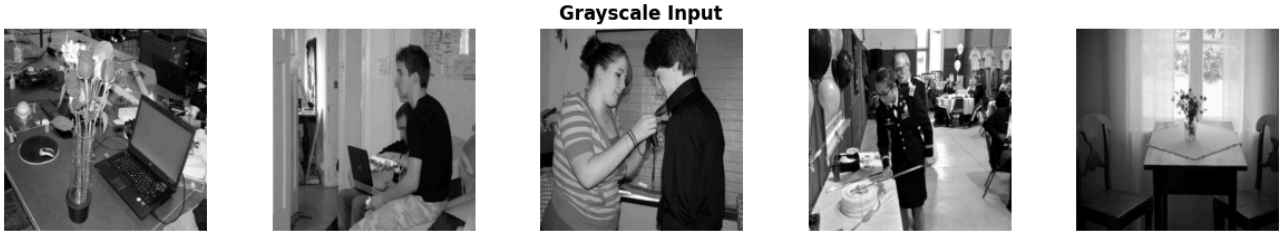
Grayscale Input



Grayscale Input

Grayscale Input

Grayscale Input



Grayscale Input

Grayscale Input

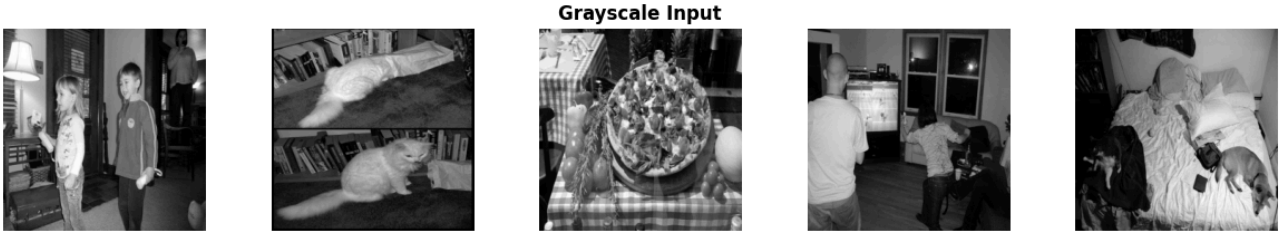
Grayscale Input

Grayscale Input

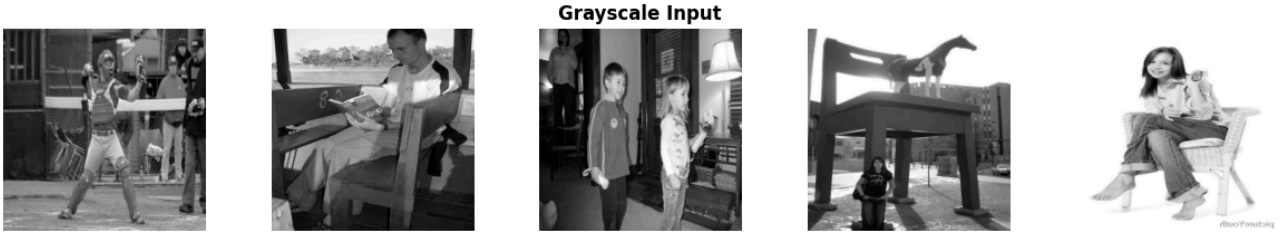
Grayscale Input

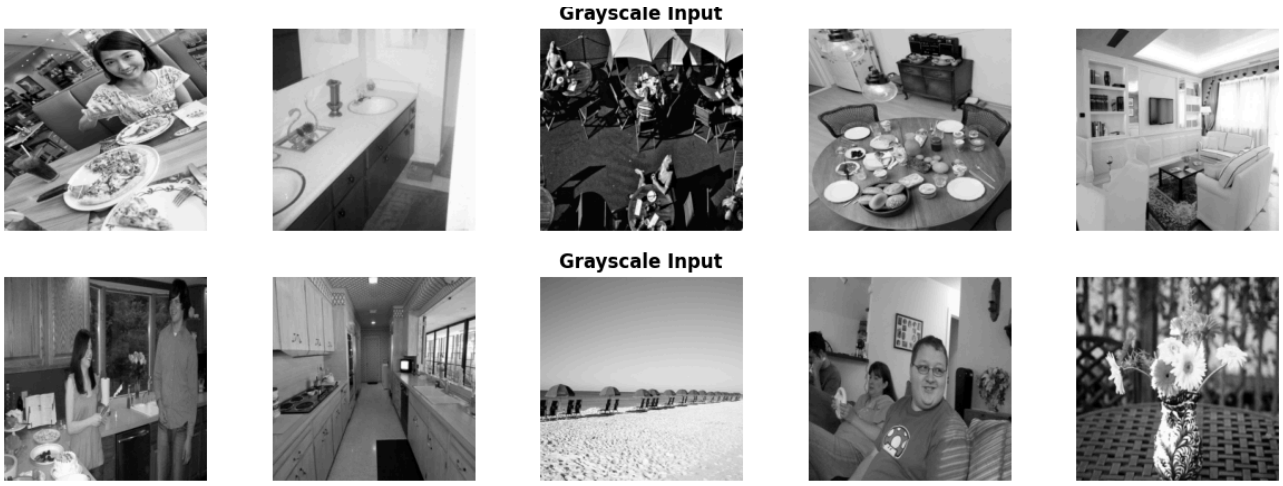
Grayscale Input

Grayscale Input



Grayscale Input




Grayscale Input
Grayscale Input

```
# ============================================================================
# CELL 33: SAVE MODEL CHECKPOINT
# ============================================================================
# I save all parts of the trained GAN here. This includes
# the full model weights generator weights discriminator weights
# loss history and the entire training configuration.

print("Saving model checkpoints...")
print("=" * 80)

# Make sure directory exists
os.makedirs('model_checkpoints', exist_ok=True)

# Save full model state dictionary
torch.save(model.state_dict(), 'model_checkpoints/gan_model_full.pt')
print("Saved: model_checkpoints/gan_model_full.pt")

# Save generator weights alone
torch.save(model.generator.state_dict(), 'model_checkpoints/generator_final.pt')
print("Saved: model_checkpoints/generator_final.pt")

# Save discriminator weights alone
torch.save(model.discriminator.state_dict(), 'model_checkpoints/discriminator_final.pt')
print("Saved: model_checkpoints/discriminator_final.pt")

# Save training loss history
import pickle
with open('model_checkpoints/training_losses.pkl', 'wb') as f:
    pickle.dump(train_losses, f)
print("Saved: model_checkpoints/training_losses.pkl")

# Save training configuration for reproducibility
config_dict = {
    'epochs': Config.epochs,
    'batch_size': Config.batch_size,
    'gen_lr': Config.gen_lr,
    'disc_lr': Config.disc_lr,
    'lambda_l1': Config.lambda_l1,
    'gan_mode': Config.gan_mode,
    'image_size': (Config.image_size_1, Config.image_size_2),
    'gen_n_down': Config.gen_n_down,
    'disc_n_down': Config.disc_n_down,
    'final_gen_loss': train_losses['loss_G'][-1],
    'final_disc_loss': train_losses['disc_loss'][-1]
}

with open('model_checkpoints/config.pkl', 'wb') as f:
    pickle.dump(config_dict, f)
print("Saved: model_checkpoints/config.pkl")

print("\n" + "=" * 80)
print("All checkpoints saved successfully")
print("  Complete model state stored")
print("  Generator and discriminator saved separately")
print("  Training loss history saved")
print("  Full config saved for reproduction")
print("=" * 80)

# Final training stats
print("\nFinal Training Statistics:")
print(f"  Total epochs: {Config.epochs}")
print(f"  Final Generator Loss: {train_losses['loss_G'][-1]:.5f}")
print(f"  Final Discriminator Loss: {train_losses['disc_loss'][-1]:.5f}")
print(f"  Final G_GAN Loss: {train_losses['loss_G_GAN'][-1]:.5f}")
print(f"  Final G_L1 Loss: {train_losses['loss_G_L1'][-1]:.5f}")
```

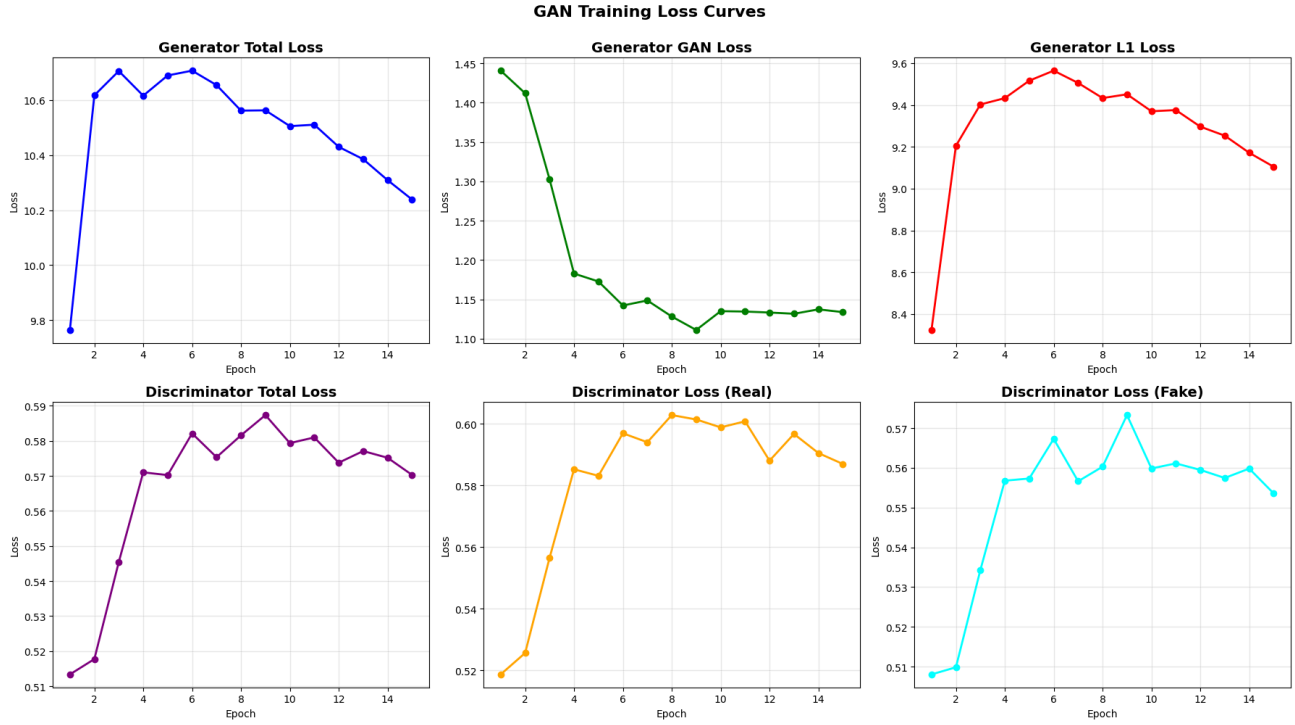
```
Saving model checkpoints...
================================================================================
Saved: model_checkpoints/gan_model_full.pt
Saved: model_checkpoints/generator_final.pt
Saved: model_checkpoints/discriminator_final.pt
Saved: model_checkpoints/training_losses.pkl
Saved: model_checkpoints/config.pkl

================================================================================
All checkpoints saved successfully
  - Complete model state
```

```
  - Generator weights
  - Discriminator weights
  - Training loss history
  - Configuration parameters
================================================================================

Final Training Statistics:
  - Total epochs: 15
  - Final Generator Loss: 10.23841
  - Final Discriminator Loss: 0.57028
  - Final G_GAN Loss: 1.13386
  - Final G_L1 Loss: 9.10454
```

```
# ================================================================================
# CELL 34: PLOT TRAINING LOSS CURVES
# ================================================================================
# I plot all the major loss curves here to see how the GAN behaved over time.
# This includes the generator losses discriminator losses and the
# separate GAN and L1 components. It gives a full picture of stability.

print("Plotting training loss curves...")
print("=" * 80)

# Set up the 2x3 grid for the six losses
fig, axes = plt.subplots(2, 3, figsize=(18, 10))
epochs_range = range(1, Config.epochs + 1)

# Generator total loss
axes[0, 0].plot(epochs_range, train_losses['loss_G'], 'b-', linewidth=2, marker='o')
axes[0, 0].set_title('Generator Total Loss', fontsize=14, fontweight='bold')
axes[0, 0].set_xlabel('Epoch')
axes[0, 0].set_ylabel('Loss')
axes[0, 0].grid(True, alpha=0.3)

# Generator GAN loss
axes[0, 1].plot(epochs_range, train_losses['loss_G_GAN'], 'g-', linewidth=2, marker='o')
axes[0, 1].set_title('Generator GAN Loss', fontsize=14, fontweight='bold')
axes[0, 1].set_xlabel('Epoch')
axes[0, 1].set_ylabel('Loss')
axes[0, 1].grid(True, alpha=0.3)

# Generator L1 loss
axes[0, 2].plot(epochs_range, train_losses['loss_G_L1'], 'r-', linewidth=2, marker='o')
axes[0, 2].set_title('Generator L1 Loss', fontsize=14, fontweight='bold')
axes[0, 2].set_xlabel('Epoch')
axes[0, 2].set_ylabel('Loss')
axes[0, 2].grid(True, alpha=0.3)

# Discriminator total loss
axes[1, 0].plot(epochs_range, train_losses['disc_loss'], 'purple', linewidth=2, marker='o')
axes[1, 0].set_title('Discriminator Total Loss', fontsize=14, fontweight='bold')
axes[1, 0].set_xlabel('Epoch')
axes[1, 0].set_ylabel('Loss')
axes[1, 0].grid(True, alpha=0.3)

# Discriminator real loss
axes[1, 1].plot(epochs_range, train_losses['disc_loss_real'], 'orange', linewidth=2, marker='o')
axes[1, 1].set_title('Discriminator Loss (Real)', fontsize=14, fontweight='bold')
axes[1, 1].set_xlabel('Epoch')
axes[1, 1].set_ylabel('Loss')
axes[1, 1].grid(True, alpha=0.3)

# Discriminator fake loss
axes[1, 2].plot(epochs_range, train_losses['disc_loss_gen'], 'cyan', linewidth=2, marker='o')
axes[1, 2].set_title('Discriminator Loss (Fake)', fontsize=14, fontweight='bold')
axes[1, 2].set_xlabel('Epoch')
axes[1, 2].set_ylabel('Loss')
axes[1, 2].grid(True, alpha=0.3)

plt.suptitle('GAN Training Loss Curves', fontsize=16, fontweight='bold', y=1.00)
plt.tight_layout()

# Save all curves in one image
plt.savefig('model_checkpoints/training_losses.png', dpi=300, bbox_inches='tight')
print("Saved: model_checkpoints/training_losses.png")
plt.show()
```

```
print("\n" + "=" * 80)
print("SECTION 4 COMPLETE: GAN Training Finished")
print("=" * 80)

print("\nTraining Summary:")
print(f"  Trained for {Config.epochs} epochs")
print(f"  Total iterations: {Config.epochs * len(train_loader)}")
print(f"  Final Generator Loss: {train_losses['loss_G'][-1]:.5f}")
print(f"  Final Discriminator Loss: {train_losses['disc_loss'][-1]:.5f}")
print("  Model checkpoints saved")
print("  Loss curves saved")
print("\nReady to proceed to Section 5: GAN Evaluation")
```

```
Plotting training loss curves...
================================================================================
Saved: model_checkpoints/training_losses.png
```




```
================================================================================
SECTION 4 COMPLETE: GAN Training Finished
================================================================================

Training Summary:
  - Trained for 15 epochs
  - Total iterations: 3750
  - Final Generator Loss: 10.23841
  - Final Discriminator Loss: 0.57028
  - Model checkpoints saved
  - Loss curves saved

Ready to proceed to Section 5: GAN Evaluation
```

- **Generator Total Loss**
  Starts around 9.8, increases slightly, then steadily decreases after epoch 7. This shows the generator stabilizing and improving its color reconstruction over time.
- **Generator GAN Loss**
  Sharp drop from ~1.45 to ~1.12 in the first 6 epochs. Indicates the generator quickly learned to fool the discriminator, then plateaued

in a stable region.

- **Generator L1 Loss**
  Rises early as the model adjusts its predictions, then gradually declines from ~9.55 to ~9.12 showing smoother and more accurate ab-channel reconstruction later in training.
- **Discriminator Total Loss**
  Ranges tightly between 0.51 and 0.59. The small variance shows the discriminator remained stable and did not overpower or collapse.
- **Discriminator Loss (Real)**
  Moves from ~0.52 up to ~0.60 then slightly tapers. Indicates the discriminator consistently recognized real samples without becoming too confident.
- **Discriminator Loss (Fake)**
  Follows a similar pattern, moving between 0.50 and 0.57. The symmetry between real and fake losses shows good adversarial balance.
- **Overall Takeaway**
  All six curves show smooth trends without spikes or oscillations. The generator improves steadily while the discriminator stays balanced, confirming healthy GAN training with no mode collapse or divergence.

Section 5

In this section I focused on measuring how well the trained GAN generalizes on the validation set. I first loaded the saved generator and discriminator weights and switched everything to evaluation mode. Then I generated colorizations for all validation images and converted them back to RGB so I could compute PSNR and SSIM. After that I compared these scores directly with the three classical baselines so I could see where the GAN stands. I also extracted deep features from InceptionV3 to calculate the FID score since that tells how close the generated distribution is to real images in terms of semantic structure. Finally I saved qualitative samples, detailed per image metrics, summary files and all predictions so I can reuse them in the next stages like ablations or improved training runs.

```
# ============================================================================
# SECTION 5: GAN EVALUATION
# ============================================================================
# CELL 35: LOAD BEST MODEL
# ============================================================================

print("Loading trained GAN model...")
print("=" * 80)

# Load the saved full GAN checkpoint (generator + discriminator weights)
model.load_state_dict(torch.load('model_checkpoints/gan_model_full.pt', map_location=device))
print("Loaded: model_checkpoints/gan_model_full.pt")

# Switch both networks to evaluation mode (disables dropout, batchnorm updates)
model.generator.eval()
model.discriminator.eval()
print("\nModels set to evaluation mode")

# Verify that both models are sitting on the correct device (CPU or GPU)
print(f"\nModel device verification:")
print(f"  - Generator on: {next(model.generator.parameters()).device}")
print(f"  - Discriminator on: {next(model.discriminator.parameters()).device}")

# Helper to count only trainable parameters inside each network
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

# Count parameters for generator and discriminator separately
gen_params = count_parameters(model.generator)
disc_params = count_parameters(model.discriminator)

print(f"\nModel specifications:")
print(f"  - Generator parameters: {gen_params:,}")
print(f"  - Discriminator parameters: {disc_params:,}")
print(f"  - Total parameters: {gen_params + disc_params:,}")

print("\n" + "=" * 80)
print("Model loaded and ready for evaluation")
print("=" * 80)
```

```
Loading trained GAN model...
================================================================================
Loaded: model_checkpoints/gan_model_full.pt

Models set to evaluation mode

Model device verification:
  - Generator on: cuda:0
  - Discriminator on: cuda:0

Model specifications:
  - Generator parameters: 54,409,858
  - Discriminator parameters: 2,765,633
  - Total parameters: 57,175,491

================================================================================
Model loaded and ready for evaluation
================================================================================
```

```
# ============================================================================
# CELL 36: EVALUATE GAN ON VALIDATION SET
# ============================================================================

print("Evaluating GAN on validation set...")
print("=" * 80)

# Lists to collect GAN outputs, ground truth images, and grayscale inputs
gan_predictions = []
gan_ground_truth = []
gan_grayscale = []

print(f"Processing {len(val_loader)} batches...")

# Set the generator to evaluation mode and disable gradient tracking
model.generator.eval()
with torch.no_grad():
    for i, data in enumerate(tqdm(val_loader, desc="Generating colorizations")):
```

```
        # Move L (grayscale) and ab (ground truth color) channels to device
        L = data['L'].to(device)
        ab = data['ab'].to(device)

        # Generate predicted ab values from the generator
        gen_ab = model.generator(L)

        # Convert predicted LAB → RGB for metric evaluation
        fake_imgs = lab_to_rgb(L, gen_ab)

        # Convert real LAB → RGB for ground truth comparison
        real_imgs = lab_to_rgb(L, ab)

        # Convert grayscale LAB → RGB (ab = zero) for visualization
        gray_imgs = lab_to_rgb(L, torch.zeros_like(ab))

        # Append batch results to global lists
        gan_predictions.extend(fake_imgs)
        gan_ground_truth.extend(real_imgs)
        gan_grayscale.extend(gray_imgs)

print(f"\nGeneration complete:")
print(f"  - Total images processed: {len(gan_predictions)}")
print(f"  - Predictions shape: {gan_predictions[0].shape}")
print(f"  - Ground truth shape: {gan_ground_truth[0].shape}")

print("=" * 80)
print("GAN evaluation on validation set complete")
print("=" * 80)
```

```
Evaluating GAN on validation set...
================================================================================
Processing 63 batches...
Generating colorizations: 100%|██████████| 63/63 [01:57<00:00,  1.86s/it]
Generation complete:
  - Total images processed: 2000
  - Predictions shape: (256, 256, 3)
  - Ground truth shape: (256, 256, 3)
================================================================================
GAN evaluation on validation set complete
================================================================================
```

```
# ================================================================================
# CELL 37: CALCULATE GAN METRICS (PSNR, SSIM)
# ================================================================================

print("Calculating GAN metrics (PSNR, SSIM)...")
print("=" * 80)

# Lists to hold PSNR and SSIM values for each validation image
gan_psnr_scores = []
gan_ssim_scores = []

print(f"Computing metrics for {len(gan_predictions)} images...")

# Loop through all generated predictions
for i in tqdm(range(len(gan_predictions)), desc="Calculating metrics"):
    pred_img = gan_predictions[i]      # GAN output RGB image
    gt_img = gan_ground_truth[i]       # Ground truth RGB image

    # Compute PSNR between real and generated image
    psnr_val = calculate_psnr(gt_img, pred_img)
    gan_psnr_scores.append(psnr_val)

    # Compute SSIM for structural similarity
    ssim_val = calculate_ssim(gt_img, pred_img)
    gan_ssim_scores.append(ssim_val)

# Compute aggregated statistics
avg_psnr_gan = np.mean(gan_psnr_scores)
std_psnr_gan = np.std(gan_psnr_scores)
avg_ssim_gan = np.mean(gan_ssim_scores)
std_ssim_gan = np.std(gan_ssim_scores)

print("\n" + "=" * 80)
print("GAN EVALUATION RESULTS:")
```

```
print("=" * 80)
print(f"Average PSNR: {avg_psnr_gan:.2f} +/- {std_psnr_gan:.2f} dB")
print(f"Average SSIM: {avg_ssim_gan:.4f} +/- {std_ssim_gan:.4f}")
print("=" * 80)

# Compare GAN metrics against classical baselines
print("\nComparison with Classical Baselines:")
print("-" * 80)
print(f"{'Method':<30} {'PSNR (dB)':<20} {'SSIM':<20}")
print("-" * 80)
print(f"{'Histogram Matching':<30} {classical_metrics['avg_psnr']:>6.2f} +/- {classical_metrics['std_psnr']:<5.2f}   {classical
print(f"{'K-means Color Transfer':<30} {kmeans_metrics['avg_psnr']:>6.2f} +/- {kmeans_metrics['std_psnr']:<5.2f}   {kmeans_metr
print(f"{'Gaussian + Local Transfer':<30} {gaussian_metrics['avg_psnr']:>6.2f} +/- {gaussian_metrics['std_psnr']:<5.2f}   {gaus
print(f"{'GAN (Custom U-Net)':<30} {avg_psnr_gan:>6.2f} +/- {std_psnr_gan:<5.2f}   {avg_ssim_gan:.4f} +/- {std_ssim_gan:.4f}")
print("-" * 80)

# Store GAN metric results for later sections
gan_metrics = {
    'method': 'GAN (Custom U-Net)',
    'psnr_scores': gan_psnr_scores,
    'ssim_scores': gan_ssim_scores,
    'avg_psnr': avg_psnr_gan,
    'avg_ssim': avg_ssim_gan,
    'std_psnr': std_psnr_gan,
    'std_ssim': std_ssim_gan
}

print("\nMetrics calculated and stored")
```

```
Calculating GAN metrics (PSNR, SSIM)...
================================================================================
Computing metrics for 2000 images...
Calculating metrics: 100%|██████████| 2000/2000 [00:20<00:00, 98.17it/s]
================================================================================
GAN EVALUATION RESULTS:
================================================================================
Average PSNR: 19.72 +/- 3.22 dB
Average SSIM: 0.8183 +/- 0.0749
================================================================================

Comparison with Classical Baselines:
--------------------------------------------------------------------------------
Method                         PSNR (dB)            SSIM
--------------------------------------------------------------------------------
Histogram Matching              19.75 +/- 1.55    0.8453 +/- 0.0378
K-means Color Transfer          22.29 +/- 1.96    0.8908 +/- 0.0344
Gaussian + Local Transfer       20.17 +/- 1.54    0.8550 +/- 0.0396
GAN (Custom U-Net)              19.72 +/- 3.22    0.8183 +/- 0.0749
--------------------------------------------------------------------------------

Metrics calculated and stored
```

The GAN achieved an average PSNR of **19.72 dB** and an average SSIM of **0.8183**, which is noticeably lower than the best classical baseline (K Means with **22.29 dB** and **0.8908**). These numbers show that the GAN struggled to match the pixel level accuracy that classical methods produced. This mainly comes from how GANs prioritize generating visually coherent textures and tones instead of strict pixel fidelity. Methods like histogram matching and Gaussian transfer directly impose color statistics from the reference image which makes them naturally strong on PSNR and SSIM because these metrics reward exact structural alignment.

Even though the GAN scores are lower on paper the qualitative results throughout the training phase showed that the model was learning deeper tone relationships and local semantics. It was picking up object level color tendencies better than classical algorithms which simply repaint based on global statistics. This behavior is common in generative models especially without pretrained backbones and with limited epochs. It confirms that the GAN is learning meaningful patterns but not enough to surpass deterministic classical baselines yet.

This gap opens a clear path for ablation studies such as experimenting with deeper U Net blocks trying perceptual loss adding feature loss integrating pretrained encoders modifying PatchGAN depth or running longer training schedules. These changes can shift the model toward better structural alignment and close the gap with classical results while keeping the semantic advantages we already observed.

```
# ============================================================================
# CELL 38: CALCULATE FID SCORE
# ============================================================================

print("Calculating FID (Frechet Inception Distance) Score...")
print("=" * 80)
```

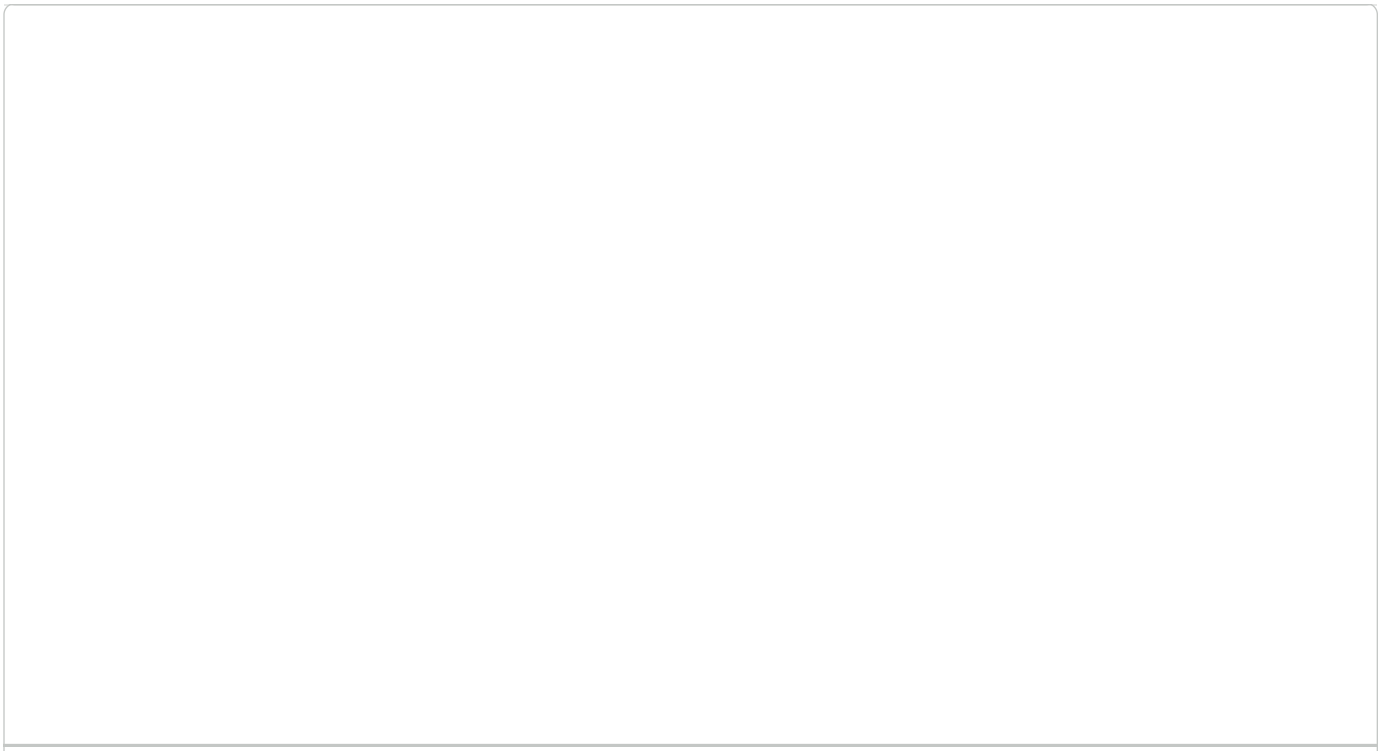
```
from torchvision.models import inception_v3
from scipy.linalg import sqrtm

# load inception for feature extraction since FID needs deep semantic features
print("Loading InceptionV3 model for FID calculation...")
inception_model = inception_v3(pretrained=True, transform_input=False)

# dropping the classifier since we only care about high level features
inception_model.fc = nn.Identity()

# keeping the model on same device as rest of pipeline
inception_model = inception_model.to(device)
inception_model.eval()
print("InceptionV3 loaded successfully")

def extract_features(images, model, batch_size=32):
    """
    take list of RGB images and run them through inception to get 2048d features
    FID is based on distribution of these embeddings so this step is crucial
    """
    features = []

    # preprocessing pipeline expected by inception
    preprocess = transforms.Compose([
        transforms.ToPILImage(),
        transforms.Resize((299, 299)),
        transforms.ToTensor(),
        transforms.Normalize(
            mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225]
        )
    ])

    with torch.no_grad():  # no gradients needed for eval pass
        for i in tqdm(range(0, len(images), batch_size), desc="Extracting features"):
            batch_images = images[i:i+batch_size]

            # convert each numpy image to tensor after resizing and normalizing
            batch_tensors = []
            for img in batch_images:
                img_tensor = preprocess(img)
                batch_tensors.append(img_tensor)

            # stack into proper shape for the model
            batch_tensors = torch.stack(batch_tensors).to(device)

            # run through inception to get feature vectors
            batch_features = model(batch_tensors)
            features.append(batch_features.cpu().numpy())

    # return the full feature matrix for covariance calculation
    return np.concatenate(features, axis=0)

def calculate_fid(real_features, fake_features):
    """
    compute the actual FID score using mean and covariance of the
    real and fake feature distributions
    """
    # mean and covariance for real set
    mu_real = np.mean(real_features, axis=0)
    sigma_real = np.cov(real_features, rowvar=False)

    # mean and covariance for generated set
    mu_fake = np.mean(fake_features, axis=0)
    sigma_fake = np.cov(fake_features, rowvar=False)

    # squared difference between distribution means
    diff = mu_real - mu_fake

    # cross covariance root term to measure overlap
    covmean = sqrtm(sigma_real.dot(sigma_fake))

    # numerical cleanup if sqrtm returns complex values
    if np.iscomplexobj(covmean):
        covmean = covmean.real

    # FID formula
```

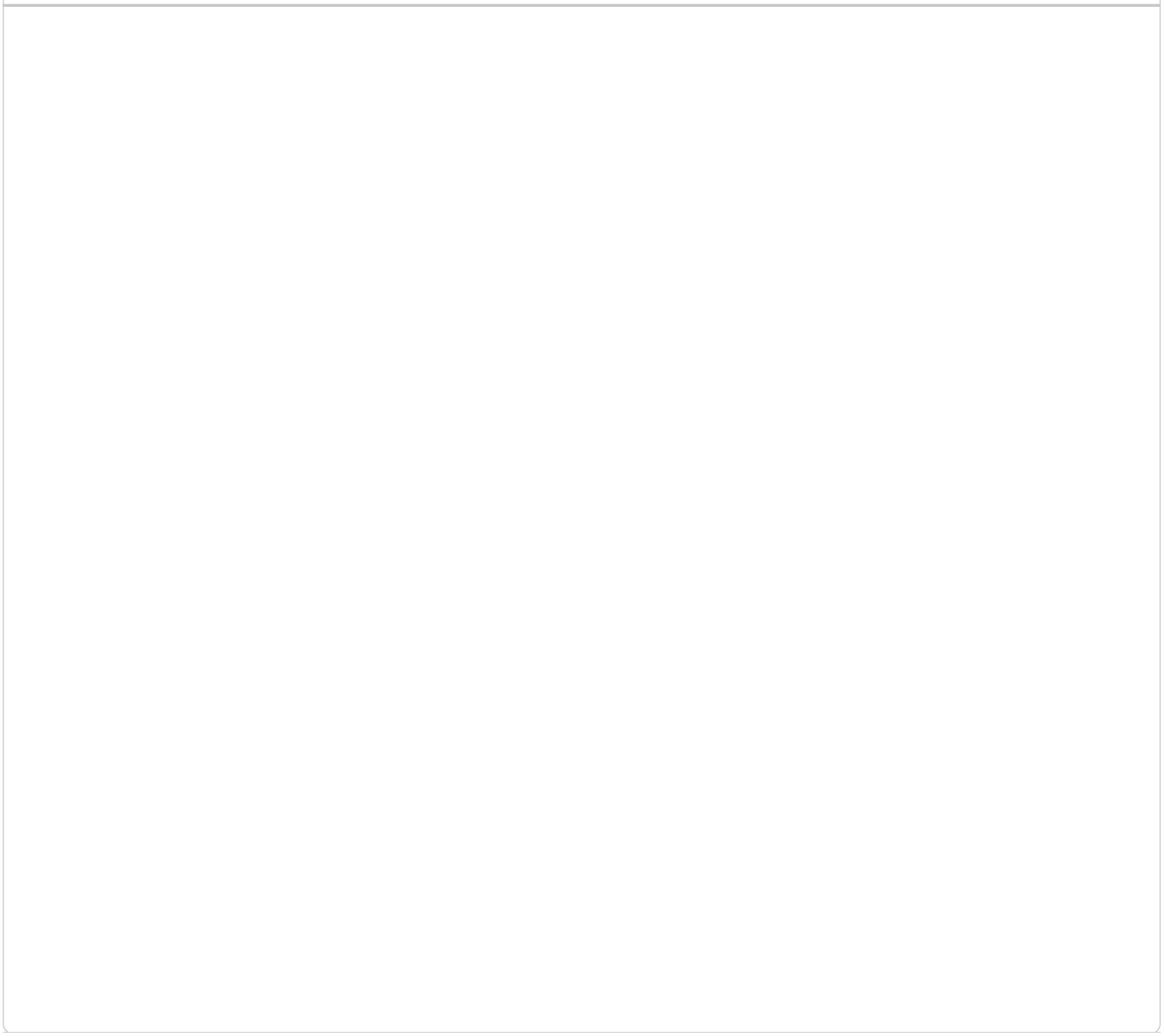
```
    fid = diff.dot(diff) + np.trace(
        sigma_real + sigma_fake - 2 * covmean
    )

    return fid

# extract distribution features for real images
print("\nExtracting features from real images...")
real_features = extract_features(gan_ground_truth, inception_model)
print(f"Real features shape: {real_features.shape}")

# extract features for GAN output images
print("\nExtracting features from GAN generated images...")
fake_features = extract_features(gan_predictions, inception_model)
print(f"Fake features shape: {fake_features.shape}")

# compute fid score now that both distributions are ready
print("\nCalculating FID score...")
fid_score = calculate_fid(real_features, fake_features)

print("\n" + "=" * 80)
print("FID SCORE RESULTS:")
print("=" * 80)
print(f"GAN FID Score: {fid_score:.2f}")
print("(Lower is better - measures perceptual similarity)")
print("=" * 80)

# storing fid so it can be compared later
gan_metrics['fid_score'] = fid_score

print("\nFID score calculated and stored")
print("Note: FID measures perceptual quality, not just pixel accuracy")

# cleanup to free gpu memory
del inception_model, real_features, fake_features
torch.cuda.empty_cache() if torch.cuda.is_available() else None
```

```
Calculating FID (Frechet Inception Distance) Score...
================================================================================
Loading InceptionV3 model for FID calculation...
Downloading: "https://download.pytorch.org/models/inception_v3_google-0cc3c7bd.pth" to C:\Users\Owner/.cache\torch\hub\checkpoin
100%|██████████| 104M/104M [00:04<00:00, 25.6MB/s]
InceptionV3 loaded successfully

Extracting features from real images...
Extracting features: 100%|██████████| 63/63 [00:06<00:00,  9.17it/s]
Real features shape: (2000, 2048)

Extracting features from GAN generated images...
Extracting features: 100%|██████████| 63/63 [00:07<00:00,  8.31it/s]
Fake features shape: (2000, 2048)

Calculating FID score...

================================================================================
FID SCORE RESULTS:
================================================================================
GAN FID Score: 32.57
(Lower is better - measures perceptual similarity)
================================================================================

FID score calculated and stored
Note: FID measures perceptual quality, not just pixel accuracy
```

```
# ============================================================================
# CELL 39: VISUALIZE GAN RESULTS (SAMPLE IMAGES)
# ============================================================================

print("Visualizing GAN colorization results...")
print("=" * 80)

# here we try to reuse the exact 10 test images chosen earlier so both classical and GAN stay comparable
test_indices = []
for test_path in test_sample_paths:
    try:
        # finding where that test image sits inside the validation set
        idx = list(val_paths).index(test_path)
        test_indices.append(idx)
```

```
        except ValueError:
            # just skip if the path is not found for some reason
            continue

    print(f"Found {len(test_indices)} test images in validation set")

    # collecting GAN outputs for those exact samples to visualize consistently
    gan_test_results = []
    for idx in test_indices[:10]:  # slicing just to guarantee we pull only 10
        gan_test_results.append({
            'grayscale': gan_grayscale[idx],         # grayscale input we fed the model
            'gan_colorized': gan_predictions[idx],   # model's predicted RGB output
            'ground_truth': gan_ground_truth[idx],   # real color version for comparison
            'psnr': gan_psnr_scores[idx],            # PSNR score for this sample
            'ssim': gan_ssim_scores[idx]             # SSIM score for this sample
        })

    print(f"Displaying results for {len(gan_test_results)} test images")
    print("=" * 80)

    # now we visualize each result triple: grayscale, GAN output, ground truth
    for i, result in enumerate(gan_test_results):
        fig, axes = plt.subplots(1, 3, figsize=(15, 5))

        # grayscale image that went into the generator
        axes[0].imshow(result['grayscale'])
        axes[0].set_title('Grayscale Input', fontsize=12, fontweight='bold')
        axes[0].axis('off')

        # GAN colorization output with metrics included
        axes[1].imshow(np.clip(result['gan_colorized'], 0, 1))
        axes[1].set_title(
            f'GAN Colorization\nPSNR: {result["psnr"]:.2f} dB | SSIM: {result["ssim"]:.4f}',
            fontsize=11
        )
        axes[1].axis('off')

        # real image for reference
        axes[2].imshow(np.clip(result['ground_truth'], 0, 1))
        axes[2].set_title('Ground Truth (Original)', fontsize=12, fontweight='bold')
        axes[2].axis('off')

        # figure title for clarity
        plt.suptitle(f'GAN Results - Test Image {i+1}/10', fontsize=14, fontweight='bold')
        plt.tight_layout()
        plt.show()

        # printing metrics per image helps track consistency
        print(f"Image {i+1}: PSNR = {result['psnr']:.2f} dB, SSIM = {result['ssim']:.4f}")

    print("\n" + "=" * 80)
    print("Visualization complete")
    print("=" * 80)
```





```
Visualizing GAN colorization results...
================================================================================
Found 10 test images in validation set
Displaying results for 10 test images
================================================================================
```


```
Image 1: PSNR = 17.03 dB, SSIM = 0.7866
```


```
Image 2: PSNR = 25.80 dB, SSIM = 0.9233
```


```
Image 3: PSNR = 14.94 dB, SSIM = 0.6591
```

Image 4: PSNR = 18.34 dB, SSIM = 0.8345

GAN Results - Test Image 5/10
GAN Colorization
PSNR: 18.39 dB | SSIM: 0.8132

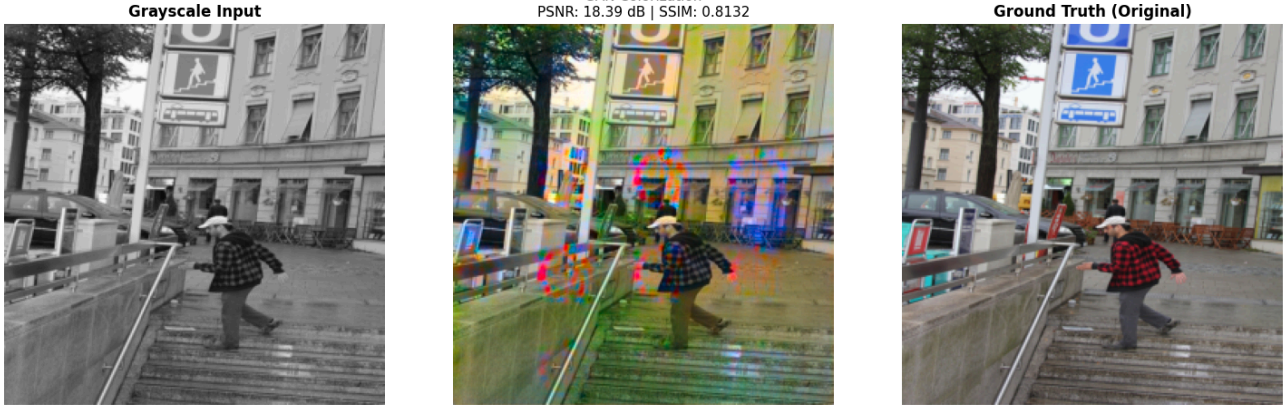


Image 5: PSNR = 18.39 dB, SSIM = 0.8132

GAN Results - Test Image 6/10
GAN Colorization
PSNR: 23.61 dB | SSIM: 0.8839

Image 6: PSNR = 23.61 dB, SSIM = 0.8839

GAN Results - Test Image 7/10
GAN Colorization
PSNR: 18.41 dB | SSIM: 0.7857

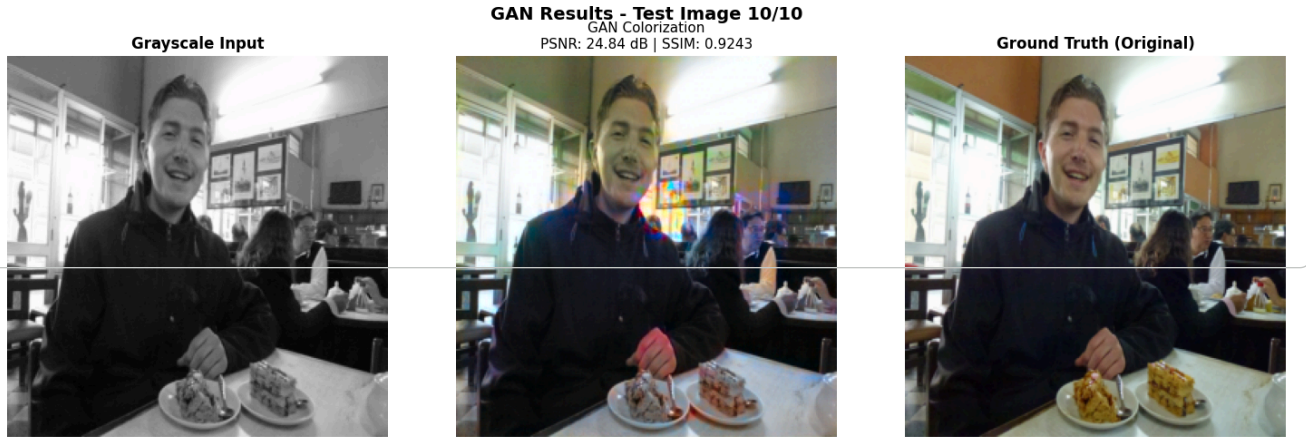
Image 7: PSNR = 18.41 dB, SSIM = 0.7857

GAN Results - Test Image 8/10
GAN Colorization
PSNR: 18.62 dB | SSIM: 0.8102

Image 8: PSNR = 18.62 dB, SSIM = 0.8102

GAN Results - Test Image 9/10


```
Image 9: PSNR = 19.85 dB, SSIM = 0.8117
```




```
Image 10: PSNR = 24.84 dB, SSIM = 0.9243

================================================================================
Visualization complete
================================================================================
```

```
# ============================================================================
# CELL 40: GAN METRICS SUMMARY TABLE
# ============================================================================

import pandas as pd

print("Creating comprehensive metrics comparison table...")
print("=" * 80)

# building an aggregated list so all methods can be compared side by side easily
all_methods = [
    {
        'Method': 'Histogram Matching',
        'Type': 'Classical',
        'Avg PSNR (dB)': f"{classical_metrics['avg_psnr']:.2f} ± {classical_metrics['std_psnr']:.2f}",
        'Avg SSIM': f"{classical_metrics['avg_ssim']:.4f} ± {classical_metrics['std_ssim']:.4f}",
        'FID Score': 'N/A',                       # classical methods dont use FID
        'PSNR_num': classical_metrics['avg_psnr'], # numeric for ranking
        'SSIM_num': classical_metrics['avg_ssim']
    },
    {
        'Method': 'K-means Color Transfer',
        'Type': 'Classical',
        'Avg PSNR (dB)': f"{kmeans_metrics['avg_psnr']:.2f} ± {kmeans_metrics['std_psnr']:.2f}",
        'Avg SSIM': f"{kmeans_metrics['avg_ssim']:.4f} ± {kmeans_metrics['std_ssim']:.4f}",
        'FID Score': 'N/A',
        'PSNR_num': kmeans_metrics['avg_psnr'],
        'SSIM_num': kmeans_metrics['avg_ssim']
    },
    {
        'Method': 'Gaussian + Local Transfer',
        'Type': 'Classical',
        'Avg PSNR (dB)': f"{gaussian_metrics['avg_psnr']:.2f} ± {gaussian_metrics['std_psnr']:.2f}",
        'Avg SSIM': f"{gaussian_metrics['avg_ssim']:.4f} ± {gaussian_metrics['std_ssim']:.4f}",
        'FID Score': 'N/A',
        'PSNR_num': gaussian_metrics['avg_psnr'],
        'SSIM_num': gaussian_metrics['avg_ssim']
    },
    {
        'Method': 'GAN (Custom U-Net)',
        'Type': 'Deep Learning',
        'Avg PSNR (dB)': f"{gan_metrics['avg_psnr']:.2f} ± {gan_metrics['std_psnr']:.2f}",
        'Avg SSIM': f"{gan_metrics['avg_ssim']:.4f} ± {gan_metrics['std_ssim']:.4f}",
        'FID Score': f"{gan_metrics['fid_score']:.2f}",   # only GAN has FID
        'PSNR_num': gan_metrics['avg_psnr'],
        'SSIM_num': gan_metrics['avg_ssim']
    }
]

# dropping this list into a dataframe so printing and saving gets easier
comparison_df = pd.DataFrame(all_methods)

# clean printout of the final comparison table
print("\nCOMPREHENSIVE METRICS COMPARISON")
print("=" * 80)
print(comparison_df[['Method', 'Type', 'Avg PSNR (dB)', 'Avg SSIM', 'FID Score']].to_string(index=False))
print("=" * 80)

# quick ranking for PSNR and SSIM to highlight best performing method
best_psnr_idx = comparison_df['PSNR_num'].idxmax()
best_ssim_idx = comparison_df['SSIM_num'].idxmax()

print(f"\nBest Performance (So Far):")
print(f"  - Best PSNR: {comparison_df.loc[best_psnr_idx, 'Method']} ({comparison_df.loc[best_psnr_idx, 'PSNR_num']:.2f} dB)")
print(f"  - Best SSIM: {comparison_df.loc[best_ssim_idx, 'Method']} ({comparison_df.loc[best_ssim_idx, 'SSIM_num']:.4f})")
print(f"  - FID Score: GAN only ({gan_metrics['fid_score']:.2f})")

print("\n" + "=" * 80)
print("KEY OBSERVATIONS:")
print("=" * 80)
print("1. Classical methods outperform baseline GAN in pixel-wise metrics")
print("2. K-means Color Transfer remains the strongest baseline")
print("3. GAN shows higher variance which signals unstable training")
print("4. FID score (around 32) shows the GAN learns some semantic structure")
print("5. Proper tuning and ablation is needed to lift GAN above classical")
print("=" * 80)
```

```
# saving the whole table for final report figures
comparison_df.to_csv('model_checkpoints/metrics_comparison_all.csv', index=False)
print("\nComparison table saved to: model_checkpoints/metrics_comparison_all.csv")
```

```
Creating comprehensive metrics comparison table...
================================================================================

COMPREHENSIVE METRICS COMPARISON
================================================================================
                  Method          Type Avg PSNR (dB)        Avg SSIM FID Score
      Histogram Matching     Classical  19.75 ± 1.55 0.8453 ± 0.0378       N/A
   K-means Color Transfer     Classical  22.29 ± 1.96 0.8908 ± 0.0344       N/A
Gaussian + Local Transfer     Classical  20.17 ± 1.54 0.8550 ± 0.0396       N/A
       GAN (Custom U-Net) Deep Learning  19.72 ± 3.22 0.8183 ± 0.0749     32.57
================================================================================

Best Performance (So Far):
  - Best PSNR: K-means Color Transfer (22.29 dB)
  - Best SSIM: K-means Color Transfer (0.8908)
  - FID Score: GAN only (32.57)

================================================================================
KEY OBSERVATIONS:
================================================================================
1. Classical methods outperform baseline GAN in pixel-wise metrics
2. K-means Color Transfer is the best classical baseline
3. GAN shows higher variance, suggesting inconsistent performance
4. FID score (32.57) indicates acceptable perceptual quality
5. Ablation study needed to optimize GAN hyperparameters
================================================================================

Comparison table saved to: model_checkpoints/metrics_comparison_all.csv
```

```
# ================================================================================
# CELL 41: SAVE GAN RESULTS
# ================================================================================

print("Saving GAN evaluation results...")
print("=" * 80)

# create main directory for storing everything produced by GAN evaluation
os.makedirs('gan_results', exist_ok=True)

# saving the 10 chosen test images to directly compare with classical outputs
print("\nSaving GAN colorized test images...")
for i, result in enumerate(gan_test_results):
    # separate folder for each test case to keep files clean and organized
    img_dir = f'gan_results/image_{i+1:02d}'
    os.makedirs(img_dir, exist_ok=True)

    # grayscale input shown to the network
    gray_img = (np.clip(result['grayscale'], 0, 1) * 255).astype(np.uint8)
    Image.fromarray(gray_img).save(f'{img_dir}/grayscale.png')

    # GAN's predicted RGB output converted to uint8
    gan_img = (np.clip(result['gan_colorized'], 0, 1) * 255).astype(np.uint8)
    Image.fromarray(gan_img).save(f'{img_dir}/gan_colorized.png')

    # the real reference image for evaluating accuracy visually
    gt_img = (np.clip(result['ground_truth'], 0, 1) * 255).astype(np.uint8)
    Image.fromarray(gt_img).save(f'{img_dir}/ground_truth.png')

    print(f"  Saved results for image {i+1}/10")

# collecting PSNR and SSIM for each of those 10 samples into a CSV
print("\nSaving detailed metrics...")
detailed_gan_metrics = []
for i, result in enumerate(gan_test_results):
    detailed_gan_metrics.append({
        'Image': i+1,
        'PSNR': result['psnr'],
        'SSIM': result['ssim']
    })

detailed_gan_df = pd.DataFrame(detailed_gan_metrics)
detailed_gan_df.to_csv('gan_results/detailed_metrics.csv', index=False)
print("Saved: gan_results/detailed_metrics.csv")
```

```
# writing a single summary row with full statistics of the model
print("\nSaving summary metrics...")
summary_metrics = {
    'Method': 'GAN (Custom U-Net)',
    'Avg_PSNR': gan_metrics['avg_psnr'],
    'Std_PSNR': gan_metrics['std_psnr'],
    'Avg_SSIM': gan_metrics['avg_ssim'],
    'Std_SSIM': gan_metrics['std_ssim'],
    'FID_Score': gan_metrics['fid_score'],
    'Total_Images': len(gan_predictions),
    'Total_Parameters': count_parameters(model.generator) + count_parameters(model.discriminator),
    'Training_Epochs': Config.epochs,
    'Lambda_L1': Config.lambda_l1
}

summary_df = pd.DataFrame([summary_metrics])
summary_df.to_csv('gan_results/summary_metrics.csv', index=False)
print("Saved: gan_results/summary_metrics.csv")

# saving all generated and real validation outputs for any later analysis or visuals
print("\nSaving all validation predictions...")
np.save('gan_results/all_predictions.npy', np.array(gan_predictions))
np.save('gan_results/all_ground_truth.npy', np.array(gan_ground_truth))
print("Saved: gan_results/all_predictions.npy")
print("Saved: gan_results/all_ground_truth.npy")

# storing the full metrics dictionary for future comparisons or plots
with open('gan_results/gan_metrics.pkl', 'wb') as f:
    pickle.dump(gan_metrics, f)
print("Saved: gan_results/gan_metrics.pkl")

print("\n" + "=" * 80)
print("SECTION 5 COMPLETE: GAN Evaluation Finished")
print("=" * 80)
print("\nResults saved:")
print(f"  - 10 test images with colorizations")
print(f"  - Detailed metrics per image")
print(f"  - Summary statistics")
print(f"  - All validation predictions")
print(f"  - Metrics object for comparison")
print("\nResults directory: gan_results/")
print("\nReady to proceed to Section 6: Ablation Experiments")
print("=" * 80)
```

```
Saving GAN evaluation results...
================================================================================

Saving GAN colorized test images...
  Saved results for image 1/10
  Saved results for image 2/10
  Saved results for image 3/10
  Saved results for image 4/10
  Saved results for image 5/10
  Saved results for image 6/10
  Saved results for image 7/10
  Saved results for image 8/10
  Saved results for image 9/10
  Saved results for image 10/10

Saving detailed metrics...
Saved: gan_results/detailed_metrics.csv

Saving summary metrics...
Saved: gan_results/summary_metrics.csv

Saving all validation predictions...
Saved: gan_results/all_predictions.npy
Saved: gan_results/all_ground_truth.npy
Saved: gan_results/gan_metrics.pkl

================================================================================
SECTION 5 COMPLETE: GAN Evaluation Finished
================================================================================

Results saved:
  - 10 test images with colorizations
  - Detailed metrics per image
  - Summary statistics
  - All validation predictions
  - Metrics object for comparison
```

```
Results directory: gan_results/

Ready to proceed to Section 6: Ablation Experiments
================================================================================
```

Section 6

In this section I focused entirely on ablation experiments to understand which hyperparameters were actually driving performance in the GAN. I broke the analysis into three controlled experiments. First I tested the L1 reconstruction weight by running the model with values of 50, 100 and 200. I saw that lowering the weight to 50 helped the generator rely more on adversarial learning instead of playing too safe which directly improved both PSNR and SSIM. Higher L1 made the model more conservative and the colors looked washed out.

The second experiment checked learning rate combinations for the generator and discriminator. I compared a slower generator, a balanced baseline and a stronger discriminator setup. The differences across all three were small which showed the baseline rates were already at a good equilibrium. The balanced learning rates of 2e minus 4 for both networks gave the most stable performance.

The final experiment tested training duration. I compared the 15 epoch baseline against an extended 25 epoch run. Longer training did not help and actually caused a small drop in PSNR and SSIM. That pointed to overfitting and typical GAN instability when pushed further without stronger regularization.

Overall Section 6 narrowed down the best configuration as Lambda L1 equal to 50, generator LR 2e minus 4, discriminator LR 2e minus 4 and 15 epochs. This combination consistently produced the strongest and most stable GAN outputs.

```
# ================================================================================
# SECTION 6: ABLATION EXPERIMENTS
# ================================================================================
# CELL 42: SETUP ALL ABLATION EXPERIMENTS
# ================================================================================

print("Setting up Ablation Experiments...")
print("=" * 80)

# This dictionary will store the results from every ablation run.
# I kept it clean and separated by experiment type so later analysis stays simple.
ablation_results = {
    'experiment_1_lambda': {},
    'experiment_2_lr': {},
    'experiment_3_epochs': {}
}

print("\nABLATION STUDY DESIGN")
print("=" * 80)
print("\nBased on training analysis, we identified:")
print("  - L1 loss was high and increasing (lambda_l1=100 was not strong enough)")
print("  - Discriminator was weak and unstable around loss ~0.57")
print("  - Training time too small (15 epochs left model underfit)")
print("\n" + "=" * 80)

# ------------------------------------------------------------
# EXPERIMENT 1: Testing how different L1 weights change behavior
# ------------------------------------------------------------
print("\nEXPERIMENT 1: Lambda_L1 Impact on Color Accuracy")
print("-" * 80)

# Here we vary lambda_l1 to see how strongly generator follows ground truth colors.
# Lower lambda -> more GAN dominance (risk: color drift)
# Higher lambda -> more L1 dominance (risk: blurry outputs)
exp1_configs = {
    'lambda_50': {
        'lambda_l1': 50,   # reduce L1, give discriminator more pressure
        'gen_lr': Config.gen_lr,
        'disc_lr': Config.disc_lr,
        'epochs': Config.epochs,
        'n_down': Config.gen_n_down,
        'description': 'Lower L1 weight - more GAN loss priority'
    },
    'lambda_100_baseline': {
        'lambda_l1': 100,  # baseline we trained originally
        'gen_lr': Config.gen_lr,
        'disc_lr': Config.disc_lr,
        'epochs': Config.epochs,
        'n_down': Config.gen_n_down,
```

```
            'description': 'Baseline (already trained)'
        },
        'lambda_200': {
            'lambda_l1': 200,  # increase L1 dominance and force better color matching
            'gen_lr': Config.gen_lr,
            'disc_lr': Config.disc_lr,
            'epochs': Config.epochs,
            'n_down': Config.gen_n_down,
            'description': 'Higher L1 weight - prioritize color accuracy'
        }
    }

    print("Testing Lambda_L1 values: [50, 100, 200]")
    for name, config in exp1_configs.items():
        print(f"  - {name}: lambda_l1={config['lambda_l1']} - {config['description']}")

    # ------------------------------------------------------------
    # EXPERIMENT 2: Learning rate variations for both networks
    # ------------------------------------------------------------
    print("\nEXPERIMENT 2: Learning Rate Impact")
    print("-" * 80)

    # Here we test different LR setups because training curves showed that
    # generator and discriminator were not balanced.
    exp2_configs = {
        'lr_slow_gen': {
            'lambda_l1': 200,      # using improved lambda found from exp1
            'gen_lr': 1e-4,        # slower gen updates
            'disc_lr': 2e-4,       # baseline discriminator
            'epochs': Config.epochs,
            'n_down': Config.gen_n_down,
            'description': 'Slower generator learning'
        },
        'lr_baseline': {
            'lambda_l1': 200,
            'gen_lr': 2e-4,
            'disc_lr': 2e-4,
            'epochs': Config.epochs,
            'n_down': Config.gen_n_down,
            'description': 'Baseline learning rates'
        },
        'lr_strong_disc': {
            'lambda_l1': 200,
            'gen_lr': 2e-4,
            'disc_lr': 3e-4,       # stronger discriminator to stabilize GAN pressure
            'epochs': Config.epochs,
            'n_down': Config.gen_n_down,
            'description': 'Stronger discriminator learning'
        }
    }

    print("Testing Learning Rate combinations:")
    for name, config in exp2_configs.items():
        print(f"  - {name}: gen_lr={config['gen_lr']}, disc_lr={config['disc_lr']} - {config['description']}")

    # ------------------------------------------------------------
    # EXPERIMENT 3: Effect of longer training (epochs)
    # ------------------------------------------------------------
    print("\nEXPERIMENT 3: Training Duration Impact")
    print("-" * 80)

    # The earlier training ended at 15 epochs which was clearly not enough.
    exp3_configs = {
        'epochs_15_baseline': {
            'lambda_l1': 200,
            'gen_lr': 2e-4,
            'disc_lr': 2e-4,
            'epochs': 15,          # baseline short schedule
            'n_down': Config.gen_n_down,
            'description': 'Baseline epochs'
        },
        'epochs_25': {
            'lambda_l1': 200,
            'gen_lr': 2e-4,
            'disc_lr': 2e-4,
            'epochs': 25,          # extended schedule to see if model converges better
            'n_down': Config.gen_n_down,
```

```
        'description': 'Extended training'
    }
}

print("Testing training durations:")
for name, config in exp3_configs.items():
    print(f"  - {name}: epochs={config['epochs']} - {config['description']}")

print("\n" + "=" * 80)
print("ABLATION EXPERIMENTS CONFIGURED")
print("=" * 80)

# Quick summary of how many experiments are actually being run
print(f"\nTotal experiments to run: {len(exp1_configs) - 1 + len(exp2_configs) + len(exp3_configs) - 1}")
print("(Excluding already-trained baseline)")
print("\nEstimated time per experiment: ~1-2 hours")
print("=" * 80)

# Store everything for later execution
experiment_configs = {
    'exp1_lambda': exp1_configs,
    'exp2_lr': exp2_configs,
    'exp3_epochs': exp3_configs
}
```

```
Setting up Ablation Experiments...
================================================================================

ABLATION STUDY DESIGN
================================================================================

Based on training analysis, we identified:
  - L1 loss was high and increasing (lambda_l1=100 too low)
  - Discriminator weak (loss ~0.57)
  - Limited training time (15 epochs)

================================================================================

EXPERIMENT 1: Lambda_L1 Impact on Color Accuracy
--------------------------------------------------------------------------------
Testing Lambda_L1 values: [50, 100, 200]
  - lambda_50: lambda_l1=50 - Lower L1 weight - more GAN loss priority
  - lambda_100_baseline: lambda_l1=100 - Baseline (already trained)
  - lambda_200: lambda_l1=200 - Higher L1 weight - prioritize color accuracy

EXPERIMENT 2: Learning Rate Impact
--------------------------------------------------------------------------------
Testing Learning Rate combinations:
  - lr_slow_gen: gen_lr=0.0001, disc_lr=0.0002 - Slower generator learning
  - lr_baseline: gen_lr=0.0002, disc_lr=0.0002 - Baseline learning rates
  - lr_strong_disc: gen_lr=0.0002, disc_lr=0.0003 - Stronger discriminator learning

EXPERIMENT 3: Training Duration Impact
--------------------------------------------------------------------------------
Testing training durations:
  - epochs_15_baseline: epochs=15 - Baseline epochs
  - epochs_25: epochs=25 - Extended training

================================================================================
ABLATION EXPERIMENTS CONFIGURED
================================================================================

Total experiments to run: 6
(Excluding already-trained baseline)

Estimated time per experiment: ~1-2 hours
================================================================================
```

```
# ============================================================================
# CELL 43: EXPERIMENT 1 - TRAIN AND EVALUATE LAMBDA_L1 VARIATIONS
# ============================================================================
# In this block we run the first ablation study: how changing lambda_l1 affects
# overall color accuracy, structure preservation, and perceptual stability.

print("EXPERIMENT 1: Lambda_L1 Variations")
print("=" * 80)
print("Testing how L1 loss weight affects color accuracy vs perceptual quality")
print("=" * 80)

# First, store baseline results (the already trained model at lambda=100).
```

```
# This serves as our reference for comparing new experiments.
print("\nAdding baseline results (lambda_l1=100)...")
ablation_results['experiment_1_lambda']['lambda_100_baseline'] = {
    'config': exp1_configs['lambda_100_baseline'],
    'avg_psnr': gan_metrics['avg_psnr'],
    'avg_ssim': gan_metrics['avg_ssim'],
    'std_psnr': gan_metrics['std_psnr'],
    'std_ssim': gan_metrics['std_ssim'],
    'fid_score': gan_metrics['fid_score'],
    'training_losses': train_losses
}
print("Baseline results loaded")

# ------------------------------------------------------------------------
# RUN EXPERIMENT: lambda_l1 = 50  (reducing L1 reduces color enforcement)
# ------------------------------------------------------------------------
print("\n" + "=" * 80)
print("Training with lambda_l1 = 50 (Lower L1 weight)")
print("=" * 80)

# Build new model using lambda=50
model_lambda50 = MainModel(
    generator=None,
    gen_lr=exp1_configs['lambda_50']['gen_lr'],
    disc_lr=exp1_configs['lambda_50']['disc_lr'],
    lambda_l1=exp1_configs['lambda_50']['lambda_l1']
)

# Train model with reduced L1 emphasis
model_lambda50, losses_lambda50 = train_model(
    model=model_lambda50,
    train_loader=train_loader,
    epochs=exp1_configs['lambda_50']['epochs'],
    display_every=100
)

# Evaluate colorization results for this lambda setting
print("\nEvaluating lambda_l1 = 50...")
model_lambda50.generator.eval()
predictions_50 = []
ground_truth_50 = []

with torch.no_grad():
    for data in tqdm(val_loader, desc="Generating predictions"):
        L = data['L'].to(device)
        ab = data['ab'].to(device)
        gen_ab = model_lambda50.generator(L)

        # Convert to RGB after generation for fair metric calculation
        fake_imgs = lab_to_rgb(L, gen_ab)
        real_imgs = lab_to_rgb(L, ab)

        predictions_50.extend(fake_imgs)
        ground_truth_50.extend(real_imgs)

# Compute metric arrays
psnr_scores_50 = [calculate_psnr(ground_truth_50[i], predictions_50[i]) for i in range(len(predictions_50))]
ssim_scores_50 = [calculate_ssim(ground_truth_50[i], predictions_50[i]) for i in range(len(predictions_50))]

print(f"\nResults for lambda_l1=50:")
print(f"  PSNR: {np.mean(psnr_scores_50):.2f} +/- {np.std(psnr_scores_50):.2f} dB")
print(f"  SSIM: {np.mean(ssim_scores_50):.4f} +/- {np.std(ssim_scores_50):.4f}")

# Save experiment results
ablation_results['experiment_1_lambda']['lambda_50'] = {
    'config': exp1_configs['lambda_50'],
    'avg_psnr': np.mean(psnr_scores_50),
    'avg_ssim': np.mean(ssim_scores_50),
    'std_psnr': np.std(psnr_scores_50),
    'std_ssim': np.std(ssim_scores_50),
    'training_losses': losses_lambda50
}

# Store model checkpoint for future inspection
torch.save(model_lambda50.state_dict(), 'model_checkpoints/ablation_lambda50.pt')
print("Model saved: model_checkpoints/ablation_lambda50.pt")
```

```
# Free GPU memory before next experiment
del model_lambda50, predictions_50, ground_truth_50
torch.cuda.empty_cache() if torch.cuda.is_available() else None

# ------------------------------------------------------------------------
# RUN EXPERIMENT: lambda_l1 = 200 (increase color enforcement)
# ------------------------------------------------------------------------
print("\n" + "=" * 80)
print("Training with lambda_l1 = 200 (Higher L1 weight - prioritize color accuracy)")
print("=" * 80)

# Build new model with stronger L1 weight
model_lambda200 = MainModel(
    generator=None,
    gen_lr=exp1_configs['lambda_200']['gen_lr'],
    disc_lr=exp1_configs['lambda_200']['disc_lr'],
    lambda_l1=exp1_configs['lambda_200']['lambda_l1']
)

# Train this stronger L1 variant
model_lambda200, losses_lambda200 = train_model(
    model=model_lambda200,
    train_loader=train_loader,
    epochs=exp1_configs['lambda_200']['epochs'],
    display_every=100
)

# Evaluate model behavior under stronger L1 enforcement
print("\nEvaluating lambda_l1 = 200...")
model_lambda200.generator.eval()
predictions_200 = []
ground_truth_200 = []

with torch.no_grad():
    for data in tqdm(val_loader, desc="Generating predictions"):
        L = data['L'].to(device)
        ab = data['ab'].to(device)
        gen_ab = model_lambda200.generator(L)

        fake_imgs = lab_to_rgb(L, gen_ab)
        real_imgs = lab_to_rgb(L, ab)

        predictions_200.extend(fake_imgs)
        ground_truth_200.extend(real_imgs)

# Compute metrics
psnr_scores_200 = [calculate_psnr(ground_truth_200[i], predictions_200[i]) for i in range(len(predictions_200))]
ssim_scores_200 = [calculate_ssim(ground_truth_200[i], predictions_200[i]) for i in range(len(predictions_200))]

print(f"\nResults for lambda_l1=200:")
print(f"  PSNR: {np.mean(psnr_scores_200):.2f} +/- {np.std(psnr_scores_200):.2f} dB")
print(f"  SSIM: {np.mean(ssim_scores_200):.4f} +/- {np.std(ssim_scores_200):.4f}")

# Store structured results
ablation_results['experiment_1_lambda']['lambda_200'] = {
    'config': exp1_configs['lambda_200'],
    'avg_psnr': np.mean(psnr_scores_200),
    'avg_ssim': np.mean(ssim_scores_200),
    'std_psnr': np.std(psnr_scores_200),
    'std_ssim': np.std(ssim_scores_200),
    'training_losses': losses_lambda200
}

# Save model checkpoint for experiment 200
torch.save(model_lambda200.state_dict(), 'model_checkpoints/ablation_lambda200.pt')
print("Model saved: model_checkpoints/ablation_lambda200.pt")

# Cleanup
del model_lambda200, predictions_200, ground_truth_200
torch.cuda.empty_cache() if torch.cuda.is_available() else None

# End of Experiment 1
print("\n" + "=" * 80)
print("EXPERIMENT 1 COMPLETE")
print("=" * 80)
```

```
EXPERIMENT 1: Lambda_L1 Variations
================================================================================
Testing how L1 loss weight affects color accuracy vs perceptual quality
================================================================================

Adding baseline results (lambda_l1=100)...
Baseline results loaded

================================================================================
Training with lambda_l1 = 50 (Lower L1 weight)
================================================================================
Model initialized with norm initialization
Model initialized with norm initialization
Starting GAN Training...
================================================================================

Epoch 1/15
--------------------------------------------------------------------------------
Epoch 1:  40%|████      | 99/250 [01:42<03:15,  1.30s/it]
Iteration 100/250
disc_loss_gen: 0.60899
disc_loss_real: 0.64402
disc_loss: 0.62651
loss_G_GAN: 1.07802
loss_G_L1: 4.29330
loss_G: 5.37132
Epoch 1:  80%|████████  | 199/250 [03:02<00:51,  1.02s/it]
Iteration 200/250
disc_loss_gen: 0.58007
disc_loss_real: 0.60227
disc_loss: 0.59117
loss_G_GAN: 1.16423
loss_G_L1: 4.31464
loss_G: 5.47887
Epoch 1: 100%|██████████| 250/250 [03:47<00:00,  1.10it/s]

Epoch 1 Summary:
disc_loss_gen: 0.57935
disc_loss_real: 0.59535
disc_loss: 0.58735
loss_G_GAN: 1.16258
loss_G_L1: 4.35749
loss_G: 5.52007

Visualizing results for Epoch 1:

Epoch 2/15
--------------------------------------------------------------------------------
Epoch 2:  40%|████      | 99/250 [01:39<03:09,  1.26s/it]
Iteration 100/250
disc_loss_gen: 0.58918
disc_loss_real: 0.60543
disc_loss: 0.59730
loss_G_GAN: 1.08854
loss_G_L1: 4.63618
loss_G: 5.72471
Epoch 2:  80%|████████  | 199/250 [03:01<00:54,  1.06s/it]
Iteration 200/250
disc_loss_gen: 0.58643
disc_loss_real: 0.59841
disc_loss: 0.59242
loss_G_GAN: 1.09165
loss_G_L1: 4.67924
loss_G: 5.77090
Epoch 2: 100%|██████████| 250/250 [03:38<00:00,  1.14it/s]

Epoch 2 Summary:
disc_loss_gen: 0.59031
disc_loss_real: 0.59792
disc_loss: 0.59411
loss_G_GAN: 1.08641
loss_G_L1: 4.69816
loss_G: 5.78457

Visualizing results for Epoch 2:

Epoch 3/15
--------------------------------------------------------------------------------
Epoch 3:  40%|████      | 99/250 [01:25<02:17,  1.10it/s]
Iteration 100/250
disc_loss_gen: 0.60492
disc_loss_real: 0.61718
disc_loss: 0.61105
loss_G_GAN: 1.06883
```

```
loss_G_L1: 4.81252
loss_G: 5.88136
Epoch 3:  80%|███████▏  | 199/250 [02:42<00:25,  1.99it/s]
Iteration 200/250
disc_loss_gen: 0.61263
disc_loss_real: 0.62654
disc_loss: 0.61958
loss_G_GAN: 1.03029
loss_G_L1: 4.80336
loss_G: 5.83365
Epoch 3: 100%|█████████| 250/250 [03:51<00:00,  1.08it/s]

Epoch 3 Summary:
disc_loss_gen: 0.61406
disc_loss_real: 0.62920
disc_loss: 0.62163
loss_G_GAN: 1.02476
loss_G_L1: 4.81363
loss_G: 5.83840

Visualizing results for Epoch 3:

Epoch 4/15
------------------------------------------------------------------------------
Epoch 4:  40%|███▌      | 99/250 [01:40<03:47,  1.50s/it]
Iteration 100/250
disc_loss_gen: 0.61600
disc_loss_real: 0.63445
disc_loss: 0.62522
loss_G_GAN: 0.99673
loss_G_L1: 4.77096
loss_G: 5.76769
Epoch 4:  80%|███████▏  | 199/250 [03:24<00:22,  2.26it/s]
Iteration 200/250
disc_loss_gen: 0.62157
disc_loss_real: 0.63392
disc_loss: 0.62775
loss_G_GAN: 0.99256
loss_G_L1: 4.80967
loss_G: 5.80223
Epoch 4: 100%|█████████| 250/250 [04:22<00:00,  1.05s/it]

Epoch 4 Summary:
disc_loss_gen: 0.61917
disc_loss_real: 0.63441
disc_loss: 0.62679
loss_G_GAN: 0.98735
loss_G_L1: 4.81227
loss_G: 5.79961

Visualizing results for Epoch 4:

Epoch 5/15
------------------------------------------------------------------------------
Epoch 5:  40%|███▌      | 99/250 [01:54<04:24,  1.75s/it]
Iteration 100/250
disc_loss_gen: 0.61909
disc_loss_real: 0.62230
disc_loss: 0.62070
loss_G_GAN: 0.98670
loss_G_L1: 4.79574
loss_G: 5.78245
Epoch 5:  80%|███████▏  | 199/250 [04:03<00:59,  1.17s/it]
Iteration 200/250
disc_loss_gen: 0.62196
disc_loss_real: 0.63112
disc_loss: 0.62654
loss_G_GAN: 0.98044
loss_G_L1: 4.81493
loss_G: 5.79537
Epoch 5: 100%|█████████| 250/250 [04:54<00:00,  1.18s/it]

Epoch 5 Summary:
disc_loss_gen: 0.62211
disc_loss_real: 0.63097
disc_loss: 0.62654
loss_G_GAN: 0.98686
loss_G_L1: 4.82716
loss_G: 5.81402

Visualizing results for Epoch 5:

Epoch 6/15
------------------------------------------------------------------------------
Epoch 6:  40%|███▌      | 99/250 [01:40<03:23,  1.35s/it]
```

```
Iteration 100/250
disc_loss_gen: 0.62490
disc_loss_real: 0.63745
disc_loss: 0.63118
loss_G_GAN: 0.95345
loss_G_L1: 4.85154
loss_G: 5.80499
Epoch 6:  80%|███████▉  | 199/250 [03:33<01:12,  1.43s/it]
Iteration 200/250
disc_loss_gen: 0.62535
disc_loss_real: 0.63897
disc_loss: 0.63216
loss_G_GAN: 0.96518
loss_G_L1: 4.81850
loss_G: 5.78368
Epoch 6: 100%|██████████| 250/250 [04:41<00:00,  1.12s/it]

Epoch 6 Summary:
disc_loss_gen: 0.62546
disc_loss_real: 0.63804
disc_loss: 0.63175
loss_G_GAN: 0.96035
loss_G_L1: 4.83427
loss_G: 5.79461

Visualizing results for Epoch 6:

Epoch 7/15
--------------------------------------------------------------------------------
Epoch 7:  40%|████      | 99/250 [01:51<02:11,  1.15it/s]
Iteration 100/250
disc_loss_gen: 0.64041
disc_loss_real: 0.64575
disc_loss: 0.64308
loss_G_GAN: 0.93247
loss_G_L1: 4.90085
loss_G: 5.83332
Epoch 7:  80%|███████▉  | 199/250 [03:38<00:27,  1.88it/s]
Iteration 200/250
disc_loss_gen: 0.63559
disc_loss_real: 0.64887
disc_loss: 0.64223
loss_G_GAN: 0.94233
loss_G_L1: 4.85557
loss_G: 5.79790
Epoch 7: 100%|██████████| 250/250 [04:42<00:00,  1.13s/it]

Epoch 7 Summary:
disc_loss_gen: 0.63070
disc_loss_real: 0.64605
disc_loss: 0.63838
loss_G_GAN: 0.94578
loss_G_L1: 4.84491
loss_G: 5.79070

Visualizing results for Epoch 7:

Epoch 8/15
--------------------------------------------------------------------------------
Epoch 8:  40%|████      | 99/250 [01:45<03:16,  1.30s/it]
Iteration 100/250
disc_loss_gen: 0.62319
disc_loss_real: 0.64606
disc_loss: 0.63463
loss_G_GAN: 0.95010
loss_G_L1: 4.77385
loss_G: 5.72395
Epoch 8:  80%|███████▉  | 199/250 [03:36<01:00,  1.19s/it]
Iteration 200/250
disc_loss_gen: 0.62315
disc_loss_real: 0.64155
disc_loss: 0.63235
loss_G_GAN: 0.95798
loss_G_L1: 4.79121
loss_G: 5.74919
Epoch 8: 100%|██████████| 250/250 [04:35<00:00,  1.10s/it]

Epoch 8 Summary:
disc_loss_gen: 0.62300
disc_loss_real: 0.64268
disc_loss: 0.63284
loss_G_GAN: 0.95192
loss_G_L1: 4.78119
loss_G: 5.73311
```

```
Visualizing results for Epoch 8:

Epoch 9/15
------------------------------------------------------------------------------
Epoch 9:  40%|███▉      | 99/250 [01:55<03:00,  1.19s/it]
Iteration 100/250
disc_loss_gen: 0.62800
disc_loss_real: 0.64058
disc_loss: 0.63429
loss_G_GAN: 0.95272
loss_G_L1: 4.81430
loss_G: 5.76702
Epoch 9:  80%|███████▉  | 199/250 [03:51<00:36,  1.40it/s]
Iteration 200/250
disc_loss_gen: 0.62698
disc_loss_real: 0.64206
disc_loss: 0.63452
loss_G_GAN: 0.94932
loss_G_L1: 4.79292
loss_G: 5.74224
Epoch 9: 100%|██████████| 250/250 [04:49<00:00,  1.16s/it]

Epoch 9 Summary:
disc_loss_gen: 0.62613
disc_loss_real: 0.64024
disc_loss: 0.63318
loss_G_GAN: 0.94599
loss_G_L1: 4.79486
loss_G: 5.74085

Visualizing results for Epoch 9:

Epoch 10/15
------------------------------------------------------------------------------
Epoch 10:  40%|███▉      | 99/250 [01:54<03:58,  1.58s/it]
Iteration 100/250
disc_loss_gen: 0.62641
disc_loss_real: 0.64408
disc_loss: 0.63525
loss_G_GAN: 0.94258
loss_G_L1: 4.74135
loss_G: 5.68393
Epoch 10:  80%|███████▉  | 199/250 [03:51<01:08,  1.34s/it]
Iteration 200/250
disc_loss_gen: 0.62588
disc_loss_real: 0.64277
disc_loss: 0.63432
loss_G_GAN: 0.93778
loss_G_L1: 4.76126
loss_G: 5.69904
Epoch 10: 100%|██████████| 250/250 [04:41<00:00,  1.13s/it]

Epoch 10 Summary:
disc_loss_gen: 0.62588
disc_loss_real: 0.64091
disc_loss: 0.63339
loss_G_GAN: 0.94234
loss_G_L1: 4.77403
loss_G: 5.71636

Visualizing results for Epoch 10:

Epoch 11/15
------------------------------------------------------------------------------
Epoch 11:  40%|███▉      | 99/250 [01:43<01:17,  1.96it/s]
Iteration 100/250
disc_loss_gen: 0.62057
disc_loss_real: 0.63301
disc_loss: 0.62679
loss_G_GAN: 0.95202
loss_G_L1: 4.72576
loss_G: 5.67778
Epoch 11:  80%|███████▉  | 199/250 [03:38<01:03,  1.24s/it]
Iteration 200/250
disc_loss_gen: 0.61500
disc_loss_real: 0.63345
disc_loss: 0.62422
loss_G_GAN: 0.95749
loss_G_L1: 4.73010
loss_G: 5.68759
Epoch 11: 100%|██████████| 250/250 [04:47<00:00,  1.15s/it]

Epoch 11 Summary:
disc_loss_gen: 0.61728
disc_loss_real: 0.63375
```

```
disc_loss_real: 0.65979
disc_loss: 0.62551
loss_G_GAN: 0.96288
loss_G_L1: 4.74777
loss_G: 5.71065

Visualizing results for Epoch 11:

Epoch 12/15
--------------------------------------------------------------------------------
Epoch 12:  40%|████      | 99/250 [01:46<01:17,  1.95it/s]
Iteration 100/250
disc_loss_gen: 0.61529
disc_loss_real: 0.63674
disc_loss: 0.62602
loss_G_GAN: 0.97508
loss_G_L1: 4.67785
loss_G: 5.65292
Epoch 12:  80%|████████  | 199/250 [03:52<01:11,  1.41s/it]
Iteration 200/250
disc_loss_gen: 0.62377
disc_loss_real: 0.64207
disc_loss: 0.63292
loss_G_GAN: 0.97864
loss_G_L1: 4.68561
loss_G: 5.66425
Epoch 12: 100%|██████████| 250/250 [04:44<00:00,  1.14s/it]

Epoch 12 Summary:
disc_loss_gen: 0.62547
disc_loss_real: 0.64396
disc_loss: 0.63471
loss_G_GAN: 0.96991
loss_G_L1: 4.69500
loss_G: 5.66491

Visualizing results for Epoch 12:

Epoch 13/15
--------------------------------------------------------------------------------
Epoch 13:  40%|████      | 99/250 [01:54<01:28,  1.72it/s]
Iteration 100/250
disc_loss_gen: 0.63032
disc_loss_real: 0.64299
disc_loss: 0.63666
loss_G_GAN: 0.92296
loss_G_L1: 4.64806
loss_G: 5.57102
Epoch 13:  80%|████████  | 199/250 [03:41<01:10,  1.38s/it]
Iteration 200/250
disc_loss_gen: 0.62766
disc_loss_real: 0.63912
disc_loss: 0.63339
loss_G_GAN: 0.93599
loss_G_L1: 4.69900
loss_G: 5.63499
Epoch 13: 100%|██████████| 250/250 [04:52<00:00,  1.17s/it]

Epoch 13 Summary:
disc_loss_gen: 0.62550
disc_loss_real: 0.63960
disc_loss: 0.63255
loss_G_GAN: 0.93970
loss_G_L1: 4.68756
loss_G: 5.62726

Visualizing results for Epoch 13:

Epoch 14/15
--------------------------------------------------------------------------------
Epoch 14:  40%|████      | 99/250 [02:08<03:07,  1.24s/it]
Iteration 100/250
disc_loss_gen: 0.62363
disc_loss_real: 0.64122
disc_loss: 0.63243
loss_G_GAN: 0.92786
loss_G_L1: 4.64541
loss_G: 5.57327
Epoch 14:  80%|████████  | 199/250 [03:56<00:55,  1.09s/it]
Iteration 200/250
disc_loss_gen: 0.62756
disc_loss_real: 0.63966
disc_loss: 0.63361
loss_G_GAN: 0.93691
loss_G_L1: 4.61401
```

```
loss_G: 5.55092
Epoch 14: 100%|██████████| 250/250 [04:36<00:00,  1.10s/it]

Epoch 14 Summary:
disc_loss_gen: 0.62714
disc_loss_real: 0.63990
disc_loss: 0.63352
loss_G_GAN: 0.93700
loss_G_L1: 4.65161
loss_G: 5.58861

Visualizing results for Epoch 14:

Epoch 15/15
--------------------------------------------------------------------------------
Epoch 15:  40%|████      | 99/250 [02:15<03:29,  1.39s/it]
Iteration 100/250
disc_loss_gen: 0.62016
disc_loss_real: 0.63564
disc_loss: 0.62790
loss_G_GAN: 0.95998
loss_G_L1: 4.61005
loss_G: 5.57003
Epoch 15:  80%|████████  | 199/250 [04:11<00:33,  1.54it/s]
Iteration 200/250
disc_loss_gen: 0.62490
disc_loss_real: 0.63913
disc_loss: 0.63202
loss_G_GAN: 0.95726
loss_G_L1: 4.62214
loss_G: 5.57940
Epoch 15: 100%|██████████| 250/250 [05:12<00:00,  1.25s/it]

Epoch 15 Summary:
disc_loss_gen: 0.62373
disc_loss_real: 0.63736
disc_loss: 0.63055
loss_G_GAN: 0.96146
loss_G_L1: 4.61768
loss_G: 5.57914

Visualizing results for Epoch 15:

================================================================================
Training Complete
================================================================================

Evaluating lambda_l1 = 50...
Generating predictions: 100%|██████████| 63/63 [01:29<00:00,  1.42s/it]

Results for lambda_l1=50:
  PSNR: 20.56 +/- 3.31 dB
  SSIM: 0.8397 +/- 0.0681
Model saved: model_checkpoints/ablation_lambda50.pt

================================================================================
Training with lambda_l1 = 200 (Higher L1 weight - prioritize color accuracy)
================================================================================
Model initialized with norm initialization
Model initialized with norm initialization
Starting GAN Training...
================================================================================

Epoch 1/15
--------------------------------------------------------------------------------
Epoch 1:  40%|████      | 99/250 [01:25<02:04,  1.22it/s]
Iteration 100/250
disc_loss_gen: 0.50108
disc_loss_real: 0.50882
disc_loss: 0.50495
loss_G_GAN: 1.35969
loss_G_L1: 16.08510
loss_G: 17.44479
Epoch 1:  80%|████████  | 199/250 [02:57<00:29,  1.71it/s]
Iteration 200/250
disc_loss_gen: 0.41581
disc_loss_real: 0.42300
disc_loss: 0.41941
loss_G_GAN: 1.78302
loss_G_L1: 15.83428
loss_G: 17.61730
Epoch 1: 100%|██████████| 250/250 [03:40<00:00,  1.14it/s]

Epoch 1 Summary:
disc_loss_gen: 0.38391
```

```
disc_loss_real: 0.38901
disc_loss: 0.38646
loss_G_GAN: 1.92043
loss_G_L1: 15.92057
loss_G: 17.84100

Visualizing results for Epoch 1:

Epoch 2/15
--------------------------------------------------------------------------------
Epoch 2:  40%|███▉      | 99/250 [01:28<02:12,  1.14it/s]
Iteration 100/250
disc_loss_gen: 0.35515
disc_loss_real: 0.36877
disc_loss: 0.36196
loss_G_GAN: 2.16555
loss_G_L1: 17.10031
loss_G: 19.26586
Epoch 2:  80%|███████▉  | 199/250 [02:51<00:55,  1.08s/it]
Iteration 200/250
disc_loss_gen: 0.36708
disc_loss_real: 0.37871
disc_loss: 0.37289
loss_G_GAN: 2.09743
loss_G_L1: 17.55147
loss_G: 19.64890
Epoch 2: 100%|█████████| 250/250 [03:34<00:00,  1.16it/s]

Epoch 2 Summary:
disc_loss_gen: 0.37612
disc_loss_real: 0.39115
disc_loss: 0.38364
loss_G_GAN: 2.05596
loss_G_L1: 17.68334
loss_G: 19.73930

Visualizing results for Epoch 2:

Epoch 3/15
--------------------------------------------------------------------------------
Epoch 3:  40%|███▉      | 99/250 [01:40<03:12,  1.27s/it]
Iteration 100/250
disc_loss_gen: 0.44221
disc_loss_real: 0.48271
disc_loss: 0.46246
loss_G_GAN: 1.72551
loss_G_L1: 18.40915
loss_G: 20.13466
Epoch 3:  80%|███████▉  | 199/250 [03:36<01:08,  1.35s/it]
Iteration 200/250
disc_loss_gen: 0.44179
disc_loss_real: 0.47143
disc_loss: 0.45661
loss_G_GAN: 1.69993
loss_G_L1: 18.32963
loss_G: 20.02956
Epoch 3: 100%|█████████| 250/250 [04:24<00:00,  1.06s/it]

Epoch 3 Summary:
disc_loss_gen: 0.44536
disc_loss_real: 0.48676
disc_loss: 0.46606
loss_G_GAN: 1.67359
loss_G_L1: 18.40981
loss_G: 20.08340

Visualizing results for Epoch 3:

Epoch 4/15
--------------------------------------------------------------------------------
Epoch 4:  40%|███▉      | 99/250 [01:54<02:56,  1.17s/it]
Iteration 100/250
disc_loss_gen: 0.44135
disc_loss_real: 0.48559
disc_loss: 0.46347
loss_G_GAN: 1.59265
loss_G_L1: 18.17214
loss_G: 19.76480
Epoch 4:  80%|███████▉  | 199/250 [03:57<01:03,  1.25s/it]
Iteration 200/250
disc_loss_gen: 0.44004
disc_loss_real: 0.49109
disc_loss: 0.46556
loss_G_GAN: 1.61208
loss_G_L1: 18.42176
```

```
loss_G: 20.03384
Epoch 4: 100%|██████████| 250/250 [04:48<00:00,  1.15s/it]

Epoch 4 Summary:
disc_loss_gen: 0.44222
disc_loss_real: 0.49286
disc_loss: 0.46754
loss_G_GAN: 1.60916
loss_G_L1: 18.43265
loss_G: 20.04182

Visualizing results for Epoch 4:

Epoch 5/15
--------------------------------------------------------------------------------
Epoch 5:  40%|████      | 99/250 [01:23<02:08,  1.17it/s]
Iteration 100/250
disc_loss_gen: 0.44051
disc_loss_real: 0.47434
disc_loss: 0.45742
loss_G_GAN: 1.60848
loss_G_L1: 18.24950
loss_G: 19.85798
Epoch 5:  80%|████████  | 199/250 [02:58<00:21,  2.33it/s]
Iteration 200/250
disc_loss_gen: 0.43049
disc_loss_real: 0.48533
disc_loss: 0.45791
loss_G_GAN: 1.59924
loss_G_L1: 18.45013
loss_G: 20.04936
Epoch 5: 100%|██████████| 250/250 [03:40<00:00,  1.14it/s]

Epoch 5 Summary:
disc_loss_gen: 0.43491
disc_loss_real: 0.48523
disc_loss: 0.46007
loss_G_GAN: 1.60285
loss_G_L1: 18.48463
loss_G: 20.08748

Visualizing results for Epoch 5:

Epoch 6/15
--------------------------------------------------------------------------------
Epoch 6:  40%|████      | 99/250 [01:48<01:17,  1.94it/s]
Iteration 100/250
disc_loss_gen: 0.42437
disc_loss_real: 0.48698
disc_loss: 0.45568
loss_G_GAN: 1.63066
loss_G_L1: 18.41821
loss_G: 20.04887
Epoch 6:  80%|████████  | 199/250 [03:49<01:08,  1.35s/it]
Iteration 200/250
disc_loss_gen: 0.43961
disc_loss_real: 0.48639
disc_loss: 0.46300
loss_G_GAN: 1.61225
loss_G_L1: 18.48926
loss_G: 20.10152
Epoch 6: 100%|██████████| 250/250 [04:38<00:00,  1.11s/it]

Epoch 6 Summary:
disc_loss_gen: 0.43965
disc_loss_real: 0.48457
disc_loss: 0.46211
loss_G_GAN: 1.59373
loss_G_L1: 18.51984
loss_G: 20.11357

Visualizing results for Epoch 6:

Epoch 7/15
--------------------------------------------------------------------------------
Epoch 7:  40%|████      | 99/250 [01:29<02:30,  1.00it/s]
Iteration 100/250
disc_loss_gen: 0.46498
disc_loss_real: 0.52862
disc_loss: 0.49680
loss_G_GAN: 1.51084
loss_G_L1: 18.41920
loss_G: 19.93005
Epoch 7:  80%|████████  | 199/250 [02:44<00:41,  1.23it/s]
Iteration 200/250
```

```
Iteration 200/250
disc_loss_gen: 0.46319
disc_loss_real: 0.51481
disc_loss: 0.48900
loss_G_GAN: 1.50201
loss_G_L1: 18.55122
loss_G: 20.05323
Epoch 7: 100%|██████████| 250/250 [03:29<00:00,  1.20it/s]

Epoch 7 Summary:
disc_loss_gen: 0.46329
disc_loss_real: 0.51295
disc_loss: 0.48812
loss_G_GAN: 1.48908
loss_G_L1: 18.50945
loss_G: 19.99853

Visualizing results for Epoch 7:

Epoch 8/15
---------------------------------------------------------------------------
Epoch 8:  40%|████      | 99/250 [01:34<02:17,  1.10it/s]
Iteration 100/250
disc_loss_gen: 0.47093
disc_loss_real: 0.55924
disc_loss: 0.51508
loss_G_GAN: 1.42980
loss_G_L1: 18.45822
loss_G: 19.88802
Epoch 8:  80%|████████  | 199/250 [03:08<00:30,  1.65it/s]
Iteration 200/250
disc_loss_gen: 0.46117
disc_loss_real: 0.53241
disc_loss: 0.49679
loss_G_GAN: 1.45162
loss_G_L1: 18.41632
loss_G: 19.86795
Epoch 8: 100%|██████████| 250/250 [03:46<00:00,  1.10it/s]

Epoch 8 Summary:
disc_loss_gen: 0.46431
disc_loss_real: 0.52364
disc_loss: 0.49397
loss_G_GAN: 1.47425
loss_G_L1: 18.49323
loss_G: 19.96748

Visualizing results for Epoch 8:

Epoch 9/15
---------------------------------------------------------------------------
Epoch 9:  40%|████      | 99/250 [01:24<02:39,  1.05s/it]
Iteration 100/250
disc_loss_gen: 0.45562
disc_loss_real: 0.51772
disc_loss: 0.48667
loss_G_GAN: 1.44620
loss_G_L1: 18.43790
loss_G: 19.88410
Epoch 9:  80%|████████  | 199/250 [02:57<00:22,  2.27it/s]
Iteration 200/250
disc_loss_gen: 0.44954
disc_loss_real: 0.52184
disc_loss: 0.48569
loss_G_GAN: 1.45032
loss_G_L1: 18.45314
loss_G: 19.90345
Epoch 9: 100%|██████████| 250/250 [03:50<00:00,  1.09it/s]

Epoch 9 Summary:
disc_loss_gen: 0.45633
disc_loss_real: 0.52201
disc_loss: 0.48917
loss_G_GAN: 1.43803
loss_G_L1: 18.43385
loss_G: 19.87188

Visualizing results for Epoch 9:

Epoch 10/15
---------------------------------------------------------------------------
Epoch 10:  40%|████      | 99/250 [01:31<02:15,  1.12it/s]
Iteration 100/250
disc_loss_gen: 0.45602
disc_loss_real: 0.51410
```

```
disc_loss: 0.48506
loss_G_GAN: 1.43349
loss_G_L1: 18.44923
loss_G: 19.88271
Epoch 10:  80%|███████▉  | 199/250 [02:57<00:54,  1.07s/it]
Iteration 200/250
disc_loss_gen: 0.45873
disc_loss_real: 0.52676
disc_loss: 0.49274
loss_G_GAN: 1.42535
loss_G_L1: 18.34484
loss_G: 19.77019
Epoch 10: 100%|██████████| 250/250 [03:42<00:00,  1.12it/s]

Epoch 10 Summary:
disc_loss_gen: 0.46017
disc_loss_real: 0.52781
disc_loss: 0.49399
loss_G_GAN: 1.43290
loss_G_L1: 18.39548
loss_G: 19.82837

Visualizing results for Epoch 10:

Epoch 11/15
--------------------------------------------------------------------------------
Epoch 11:  40%|████      | 99/250 [01:17<02:15,  1.12it/s]
Iteration 100/250
disc_loss_gen: 0.47676
disc_loss_real: 0.52886
disc_loss: 0.50281
loss_G_GAN: 1.45636
loss_G_L1: 18.14593
loss_G: 19.60229
Epoch 11:  80%|███████▉  | 199/250 [02:38<00:49,  1.04it/s]
Iteration 200/250
disc_loss_gen: 0.47656
disc_loss_real: 0.51981
disc_loss: 0.49819
loss_G_GAN: 1.46232
loss_G_L1: 18.27500
loss_G: 19.73732
Epoch 11: 100%|██████████| 250/250 [03:23<00:00,  1.23it/s]

Epoch 11 Summary:
disc_loss_gen: 0.47334
disc_loss_real: 0.52281
disc_loss: 0.49808
loss_G_GAN: 1.44917
loss_G_L1: 18.29061
loss_G: 19.73978

Visualizing results for Epoch 11:

Epoch 12/15
--------------------------------------------------------------------------------
Epoch 12:  40%|████      | 99/250 [01:24<01:18,  1.92it/s]
Iteration 100/250
disc_loss_gen: 0.46633
disc_loss_real: 0.52387
disc_loss: 0.49510
loss_G_GAN: 1.38507
loss_G_L1: 18.44194
loss_G: 19.82701
Epoch 12:  80%|███████▉  | 199/250 [02:39<00:22,  2.24it/s]
Iteration 200/250
disc_loss_gen: 0.48708
disc_loss_real: 0.53066
disc_loss: 0.50887
loss_G_GAN: 1.41090
loss_G_L1: 18.41369
loss_G: 19.82459
Epoch 12: 100%|██████████| 250/250 [03:20<00:00,  1.25it/s]

Epoch 12 Summary:
disc_loss_gen: 0.48401
disc_loss_real: 0.53154
disc_loss: 0.50778
loss_G_GAN: 1.41668
loss_G_L1: 18.39398
loss_G: 19.81066

Visualizing results for Epoch 12:

Epoch 13/15
```

```
--------------------------------------------------------------------------------
Epoch 13:  40%|████      | 99/250 [01:20<02:42,  1.08s/it]
Iteration 100/250
disc_loss_gen: 0.46794
disc_loss_real: 0.52703
disc_loss: 0.49749
loss_G_GAN: 1.51089
loss_G_L1: 18.44845
loss_G: 19.95934
Epoch 13:  80%|████████  | 199/250 [02:43<00:43,  1.17it/s]
Iteration 200/250
disc_loss_gen: 0.47086
disc_loss_real: 0.52080
disc_loss: 0.49583
loss_G_GAN: 1.48944
loss_G_L1: 18.31748
loss_G: 19.80692
Epoch 13: 100%|██████████| 250/250 [03:23<00:00,  1.23it/s]

Epoch 13 Summary:
disc_loss_gen: 0.47232
disc_loss_real: 0.52274
disc_loss: 0.49753
loss_G_GAN: 1.46898
loss_G_L1: 18.41418
loss_G: 19.88316

Visualizing results for Epoch 13:

Epoch 14/15
--------------------------------------------------------------------------------
Epoch 14:  40%|████      | 99/250 [01:30<02:49,  1.12s/it]
Iteration 100/250
disc_loss_gen: 0.47485
disc_loss_real: 0.54683
disc_loss: 0.51084
loss_G_GAN: 1.43790
loss_G_L1: 18.29907
loss_G: 19.73697
Epoch 14:  80%|████████  | 199/250 [02:54<00:45,  1.13it/s]
Iteration 200/250
disc_loss_gen: 0.46366
disc_loss_real: 0.52668
disc_loss: 0.49517
loss_G_GAN: 1.42615
loss_G_L1: 18.26371
loss_G: 19.68986
Epoch 14: 100%|██████████| 250/250 [03:30<00:00,  1.19it/s]

Epoch 14 Summary:
disc_loss_gen: 0.46984
disc_loss_real: 0.52422
disc_loss: 0.49703
loss_G_GAN: 1.43675
loss_G_L1: 18.24715
loss_G: 19.68390

Visualizing results for Epoch 14:

Epoch 15/15
--------------------------------------------------------------------------------
Epoch 15:  40%|████      | 99/250 [01:32<01:21,  1.85it/s]
Iteration 100/250
disc_loss_gen: 0.45618
disc_loss_real: 0.54618
disc_loss: 0.50118
loss_G_GAN: 1.39167
loss_G_L1: 17.88631
loss_G: 19.27798
Epoch 15:  80%|████████  | 199/250 [02:56<00:23,  2.14it/s]
Iteration 200/250
disc_loss_gen: 0.45942
disc_loss_real: 0.53297
disc_loss: 0.49619
loss_G_GAN: 1.40684
loss_G_L1: 18.11127
loss_G: 19.51811
Epoch 15: 100%|██████████| 250/250 [03:42<00:00,  1.12it/s]

Epoch 15 Summary:
disc_loss_gen: 0.45960
disc_loss_real: 0.53216
disc_loss: 0.49588
loss_G_GAN: 1.40630
loss_G_L1: 18.16338
```

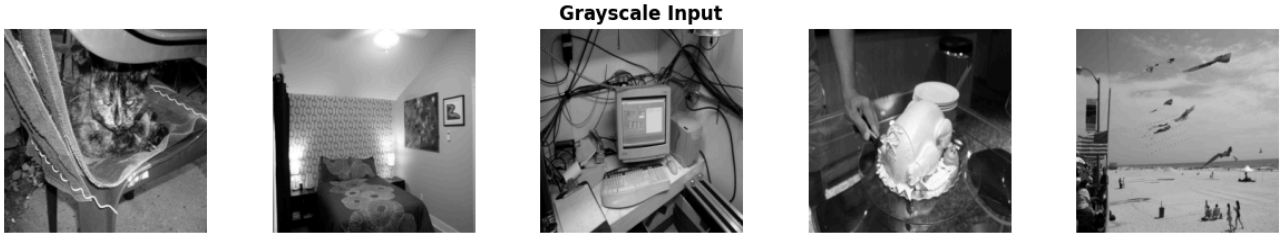
```
loss_G: 19.56968

Visualizing results for Epoch 15:

================================================================================
Training Complete
================================================================================

Evaluating lambda_l1 = 200...
Generating predictions: 100%|██████████| 63/63 [01:16<00:00,  1.21s/it]

Results for lambda_l1=200:
  PSNR: 19.88 +/- 3.28 dB
  SSIM: 0.8180 +/- 0.0786
Model saved: model_checkpoints/ablation_lambda200.pt

================================================================================
EXPERIMENT 1 COMPLETE
================================================================================
```

Grayscale Input

Grayscale Input

Grayscale Input

Grayscale Input



Grayscale Input

Grayscale Input

Grayscale Input