Colorization of grayscale images using Deep Neural Networks

Sai Rachana Kandikattu
Data Science
GWU

Snehitha Tadapaneni
Data Science
GWU
Naiska Buyandalai
Data Science
GWU

***Abstract*: Colorizing grayscale images is an inherently ill-posed problem, as multiple plausible color mappings can correspond to a single luminance channel. Recent work, particularly the ECCV16 model by Zhang et al., approaches this uncertainty by framing colorization as a classification task in the CIELAB color space and leveraging deep convolutional neural networks trained on over a million images. In this project, we reproduce the pretrained ECCV16 model for benchmarking and propose an improved colorization pipeline that integrates GAN-based refinement, perceptual feature loss, and fine-tuning using a secondary dataset. Our preprocessing method converts all images into the CIELAB space, normalizes the L channel for input, and discretizes the ab color channels for model-compatible training. We compare models using perceptual metrics such as AUC, PSNR, SSIM. Results indicate that the GAN-enhanced model produces more saturated, realistic colors while preserving structural consistency, demonstrating measurable improvement over the pretrained baseline.**

Keywords - Luminance, ECCV16, CIELAB color space, Deep Convolutional Neural Networks, AUC, PSNR, SSIM

I. Introduction

Image colorization has long been a challenging problem in computer vision. Early approaches typically relied on extensive user interaction, such as manually annotating color scribbles on grayscale images, or on hand-crafted priors that often produced flat, desaturated results. The fundamental difficulty is that colorization is an underconstrained problem: a single grayscale image encodes only luminance information, while the underlying chrominance can correspond to many plausible color configurations. For example, the same gray pixel could represent a blue sky, a green leaf, or a red shirt, depending on the context.

The advent of deep learning has dramatically changed this landscape. Convolutional neural networks (CNNs) excel at learning hierarchical representations from large-scale image datasets, making fully automatic colorization feasible. A particularly influential work is the "Colorful Image Colorization" model by Zhang et al. (ECCV 2016), which reframes colorization as a classification task rather than a direct regression in color space. Instead of predicting continuous color values, the network predicts a probability distribution over a set of quantized color bins in the CIELAB ab space. This formulation encourages the model to choose among multiple plausible colors and leads to more vibrant, diverse colorizations compared to regression-based methods that tend to average colors.

Zhang et al. also operate in the CIELAB color space, where the luminance channel (L) is decoupled from the chromatic channels (a and b). This separation aligns well with human perception and simplifies the learning problem: the network receives the L channel as input and predicts the missing ab channels. Additionally, the authors introduce a class-rebalancing loss to compensate for the natural skew of real-world images towards desaturated colors, further improving color diversity.

Building on this foundation, our project has two main objectives. First, we reproduce and evaluate the pretrained ECCV16 model as a strong baseline. Second, we extend this baseline using a GAN-based framework with three key enhancements: a PatchGAN discriminator to encourage realistic local texture, a VGG19-based perceptual loss to preserve semantic and structural details, and fine-tuning of the generator on a curated subset of ImageNet. By comparing the pretrained model and our GAN-enhanced model

on both quantitative metrics and qualitative visual inspection, we aim to assess how much adversarial and perceptual learning can improve realism, saturation, and fine-detail consistency in automatic image colorization.

II. Data Preprocessing

A. Dataset Description

For our experiments, we use a curated subset of the ImageNet dataset, which we refer to as ImageNet_50. This subset contains images drawn from 50 object categories spanning a variety of scenes, textures, and color distributions (e.g., animals, birds, everyday objects). The diversity of content is important for colorization: models trained only on narrow domains (such as faces or landscapes) may overfit to domain-specific color priors, whereas ImageNet-style data exposes the network to a broad range of color statistics.

In our implementation, the ImageNet_50 subset is organized into a standard folder hierarchy with images stored in a train directory. This directory is used as the source for both:

- Training data for the GAN-based model, where we learn to map grayscale inputs to color outputs using paired (L, ab) lab images, and
- Inference data for both the pretrained ECCV16 model and the trained GAN, where we generate colorized outputs for qualitative comparison and metric evaluation.

All images are in common formats such as .jpg or .png. Grayscale images, if present, are automatically converted into three-channel images to maintain a consistent input format during preprocessing.

B. Data Preprocessing

Because both the baseline and GAN models operate in CIELAB color space, our preprocessing pipeline is designed around converting raw RGB images into their L and ab components and preparing them as tensors suitable for PyTorch models.

1) Loading and resizing

Each image is first loaded from disk as an RGB array using the Python Imaging Library (PIL):

- If the image is already grayscale (single channel), we replicate the channel three times to create a pseudo-RGB image. This avoids special-case handling later in the pipeline.
- The image is then resized to a fixed spatial resolution of 256 × 256 pixels using bilinear interpolation. This resolution is chosen to match the input size expected by the ECCV16 architecture and to provide a balance between spatial detail and computational cost.

Formally, given an input image $I_{RGB} \in R^{H \times W \times 3}$, we obtain a resized image $I_{RGB} \in R^{256 \times 256 \times 3}$.

2) Conversion to CIELAB

The resized RGB image is converted into the CIELAB color space using ‘skimage.color.rgb2lab’. This produces three channels:

- $L \in [0,100]$: luminance
- a: green - red axis
- b: blue - yellow axis

We denote the LAB image as: $I_{LAB}=(L,a,b)$.

The key advantage of this representation is that luminance is explicitly separated from chrominance. Since our task is to hallucinate plausible colors given grayscale structure, we treat L as the input and (a, b) as the target for learning.

3) Preparing tensors for the models

From the LAB image, we extract:

- A resized L tensor L_{rs} of shape (1,256,256) used as the model input.
- For the GAN model, a corresponding ground-truth ab tensor ab_{rs} of shape (2,256,256) used as the supervision signal.

These are converted to single-precision floating-point tensors and batched using a custom Dataset and DataLoader implementation.

For the pretrained ECCV16 model, only the L channel is required as input. Internally, the model centers and normalizes L using a simple affine transformation:

$$L_{norm} = (L - 50)/100,$$

as implemented in the BaseColor.normalize_l() function. The model then predicts ab values via a 313-way color classifier followed by a learned regression layer.

For the GAN-based model, we retain the same L channel extraction but additionally require the ground-truth ab channels during training. The preprocessing function preprocess_img() is called with return_ab=True, which extracts both components:

L tensor $L_{rs} \in (R1 \times 256 \times 256)$: grayscale input to the generator

ab tensor $ab_{rs} \in (R2 \times 256 \times 256)$: chrominance ground truth for supervised learning

Unlike the pretrained model which applies internal normalization, our GAN pipeline passes raw LAB values directly to the network. The generator inherits the BaseColor normalization layer, applying the same affine transformation to L internally, while the discriminator receives concatenated [L,ab] tensors of shape (3,256,256) as input.

4) Postprocessing and Reconstruction
After either model predicts ab channels, we reconstruct a full-color image as follows:

1. If necessary, the predicted ab tensor is resized to match the original luminance resolution.
2. The original L channel L_{orig} and predicted ab channels ab^ are concatenated to form a LAB image: $I^{LAB}=(L_{orig},\hat{ab})$
3. This LAB image is converted back to RGB using *skimage.color.lab2rgb*, yielding a colorized image suitable for visualization and evaluation.

This preprocessing and postprocessing pipeline ensures that both the ECCV16 baseline and the GAN-based model operate under a consistent LAB formulation, allowing us to directly compare their outputs while leveraging the perceptual advantages of the CIELAB color space.

IV. Model Architecture And Training

A. Background on Image Colorization

Image colorization can be formulated as a mapping problem from a single grayscale luminance channel *L* to the missing chrominance components a and b in the CIELAB color space. Traditional regression-based models attempt to directly predict continuous (a,b) values from the input L. However, this strategy often leads to desaturated or averaged colors, because the network minimizes pixel-wise error and thus tends toward the statistical mode of the data distribution.

Zhang et al. (ECCV 2016) introduced the idea of treating colorization as a classification problem. Instead of predicting continuous values, the model predicts a probability distribution over 313 quantized color bins representing commonly occurring chrominance values in natural images. This allows the model to express multi-modality: a pixel can have several plausible colors and produces more vibrant and realistic outputs.

In recent years, adversarial learning has been applied to colorization, where a discriminator encourages the generator to produce outputs that are indistinguishable from real color images. GAN-based methods generally improve local texture realism and saturation compared to purely supervised models. Our work builds on both paradigms: we evaluate the ECCV16 baseline, and we propose a GAN-based colorization model enhanced with perceptual (VGG19) feature loss.

B. ECCV16 Baseline Architecture

The ECCV16 pretrained model serves as our baseline. It is a fully convolutional encoder–decoder architecture built around the following key components:
1. Encoder
The encoder progressively downsamples the L-channel input using strided convolutions:

Model 1 (1 → 64 channels)

Model 2 (64 → 128 channels)

Model 3 (128 → 256 channels)

Model 4 (256 → 512 channels)

The encoder extracts hierarchical luminance features that encode edges, textures, and semantic structure.

2. Dilated Convolution Layers
The intermediate layers (Model 5–7) use dilated convolutions to expand the receptive field without further downsampling. This allows the model to incorporate broader context, which is crucial because color often depends on the global semantics rather than local pixel intensity alone.

3. Decoder and Color Prediction
The decoder upsamples the deep feature representation back toward the original resolution:
Transposed convolution (512 → 256 channels)
Several convolutional refinement layers
Final output: 313-channel softmax representing the quantized ab color categories
A final learned 1×1 convolution regresses from the 313-bin distribution to continuous ab predictions.

4. Training Strategy
In the original ECCV16 paper, training uses:
Weighted cross-entropy loss, encouraging rare colors by rebalancing the class distribution. A billion-scale ImageNet training set for strong generalization

In our implementation, we only perform inference using the provided pretrained weights.

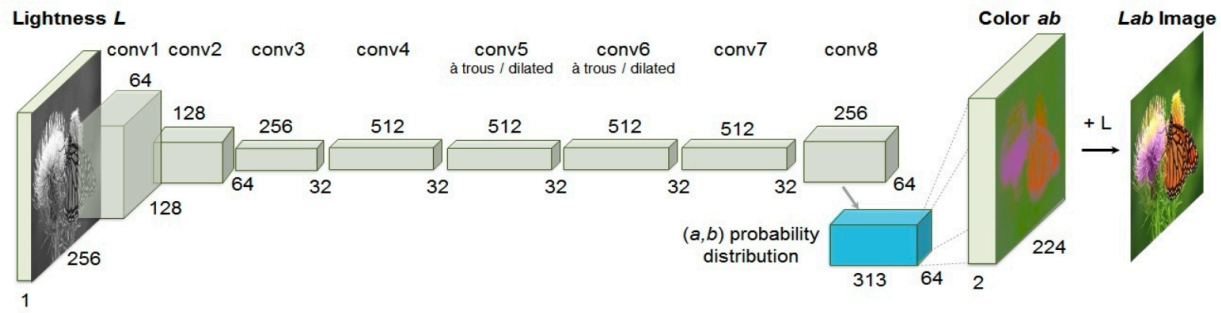


Figure 1. Our Baseline Pretrained Architecture (Zhang et al.)

C. GAN + Finetuned:

Rather than training a GAN from scratch, we adopt a **two-stage training paradigm** that has proven effective in recent literature:

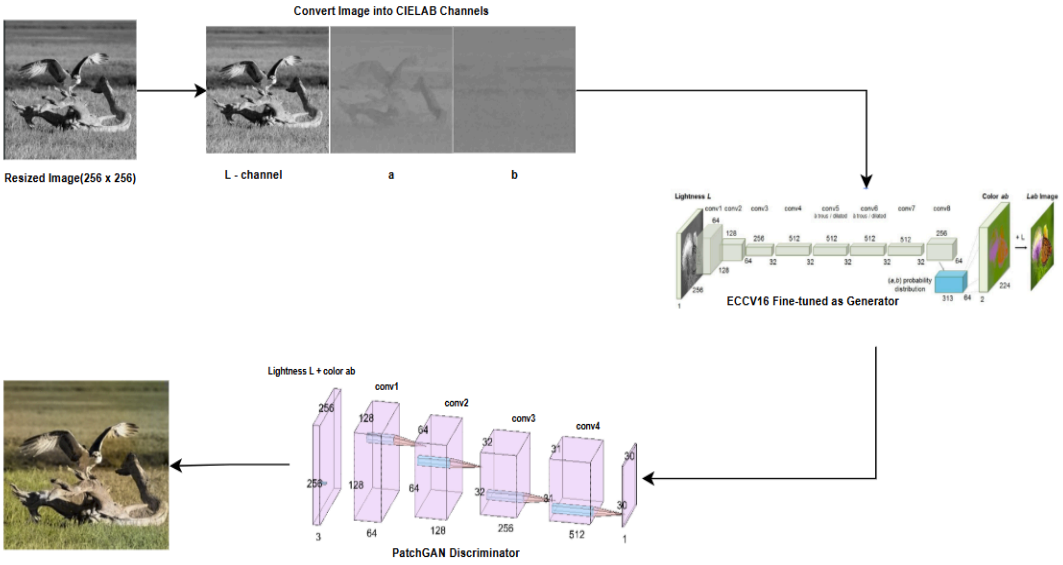


Figure 2. Training Paradigm for Model 2

Stage 1 (Finetuning): Finetune a colorization model with traditional supervised loss (Zhang et al. with 1.3M ImageNet images) as it provides stable, semantically correct colorization and serves as a strong initialization

Finetuned: To adapt the pretrained ECCV16 model to our dataset, we perform supervised fine-tuning while keeping the architecture unchanged.

1. Architecture: The finetuned model retains the complete ECCV16 structure (Figure 1)
2. Training Process**:** All model layers are trained end-to-end on our dataset. Images are converted to CIELAB space similar to pretrained model, with the L channel as input and ground-truth ab channels as supervision targets.
3. We chose to use this model as generator as it has encoder-decoder like architecture similar to UNET in pix2pix and our model is pretrained on ImageNET.

*Hybrid GAN Model Architecture:*Now we use the finetuned model as generator for the discriminator we will be using the PatchGAN Discriminator

Stage 2 (PatchGAN Discriminator): We used a PatchGAN discriminator as it evaluates local patches independently, providing more feedback to the generator about which spatial regions need improvement and as it also works well with Image to Image translation.

1. Architecture: The PatchGAN discriminator is a fully convolutional network that evaluates image realism at the patch level rather than classifying the entire image with a single scalar. This design provides fine-grained spatial feedback to the generator about which regions need improvement.The discriminator progressively downsamples the input through five convolutional blocks with batch norm for all layers expect the first layer with 4x4 convolution filters
   Block 1 (3 → 64 channels) (stride 2)
   Block 2 (64 → 128 channels) (stride 2)
   Block 3 (128 → 256 channels) (stride 2)
   Block 4 (256 → 512 channels) (stride 2)
   Block 5 (512 → 1 channel) (stride 1)

2. Training:
   - Initially, the discriminator undergoes training as fake images *G(x)* generated by the generator are inputted into the discriminator.

- Subsequently, a batch of real images from the training set is fed into the discriminator and labeled as real.
- The losses incurred from both fake and real images are summed up, averaged, and subjected to the backward operation to update the discriminator.
- Then, the generator is trained by feeding fake images into the discriminator with the intention of deceiving it into categorizing them as real. The adversarial loss is computed accordingly. Additionally, L1 loss is calculated by measuring the discrepancy between the predicted and target channels, then multiplied by the coefficient (λ= 10) for our case to balance both losses as we have an additional perceptual loss.
- This resultant loss is added to the adversarial loss, and the backward method is invoked to update the generator's parameters.
- Network optimization involves alternating between conducting a single gradient descent step on the discriminator and another step on the generator.

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

$$\mathbb{E}_{x,y}\left[\log D(x, y)\right]$$

[expected log-likelihood that the discriminator correctly identifies real image pairs D tries to maximize this term]

$$\mathbb{E}_{x,z}\left[\log(1 - D(x, G(x, z)))\right]$$

[measures how confidently the discriminator rejects fake image pairs, G tries to minimize this term by generating images that fool the discriminator]

- Thus, the overall objective is a min-max game:

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

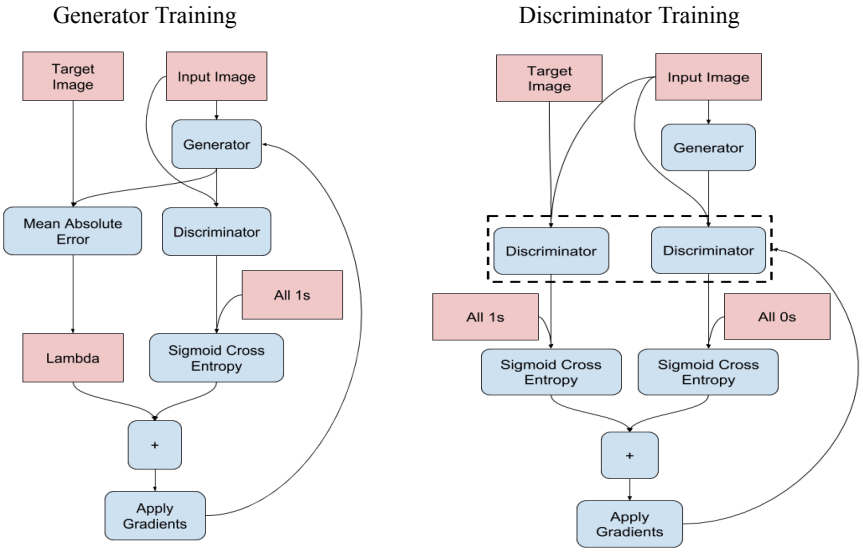


Figure 3. Block Diagram of GAN

D. *Loss Functions*

a. *Discriminator Loss:* As explained previously Discriminator has fake pairs and real pairs with the label of the image being fake and real. The Discriminator Loss could be summarized as

$$Loss\_D\_fake = BCE(D(G(x),0)\ [Fake\ images\ should\ be\ classified\ as\ 0]$$
$$Loss\_D\_real = BCE(D(x),1)\ [Real\ images\ should\ be\ classified\ as\ 1]$$
$$Loss\_D = 0.5*(Loss\_D\_fake + Loss\_D\_real)$$

b. *GAN Adversial Loss:* As the Generator tries to fool the Discriminator that the fake image is a real image with the label

$$Loss\_G\_adv = BCE(D(G(x)),\ 1)[\ Generator\ trying\ to\ fool\ Discriminator\ that\ this\ is\ a\ real\ image]$$

c. *L1 Loss:*

$$Loss\_L1 = \| x' - x \|\ [\ L1\ loss\ between\ real\ and\ predicted\ ab]$$

d. *Perceptual Loss:* Perceptual loss is computed by converting both the generated and ground-truth LAB images back to RGB and passing them through a pretrained VGG19 network; the L1 difference between their feature maps measures semantic similarity. This encourages the generator to match high-level structures and textures, not just pixel values, resulting in more realistic and context-aware colorization.

$$Perceptual\ Loss = \| f' - f \|\ [L1\ loss\ between\ feature\ extraction\ of\ real\ and\ fake\ image]$$

e. The resultant loss of GAN is

$$Loss\_G = Loss\_G\_adv + \lambda \cdot Loss\_L1 + Perceptual\ Loss(where\ \lambda = 10)$$

E. *System Pipeline*

Our complete colorization framework follows a structured multi-stage pipeline that transforms raw grayscale images into colorized outputs using both the ECCV16 baseline model and our enhanced GAN architecture.

First, each input image is loaded in RGB format, resized to 256×256, and converted into the CIELAB color space. From this representation, we extract the L channel as the model input, while the ab channels are retained as ground-truth supervision for GAN training. This preprocessing ensures that both colorization models operate consistently on perceptually meaningful luminance information.

Next, the preprocessed L channel is fed into two separate model branches. The ECCV16 pretrained model performs forward inference only, predicting chrominance values based on its 313-bin classification framework. In parallel, our GAN-based model uses the same L input but produces continuous ab values. During training, this model is optimized using adversarial loss, L1 reconstruction loss, and perceptual VGG19 feature loss.

Once either model predicts the ab channels, we reconstruct the final colorized image by concatenating the predicted ab map with the original L channel to form a complete LAB image. This LAB representation is then converted back into RGB space to produce the colorized output.

Finally, the outputs from both models undergo quantitative and qualitative evaluation. Quantitative evaluation includes loss values (L1, GAN, perceptual) and Raw AUC, while qualitative evaluation compares visual attributes such as color vibrancy, realism, saturation, and structural consistency.
These steps allow us to systematically compare the pretrained baseline and our improved GAN model under identical preprocessing, dataset, and evaluation settings.

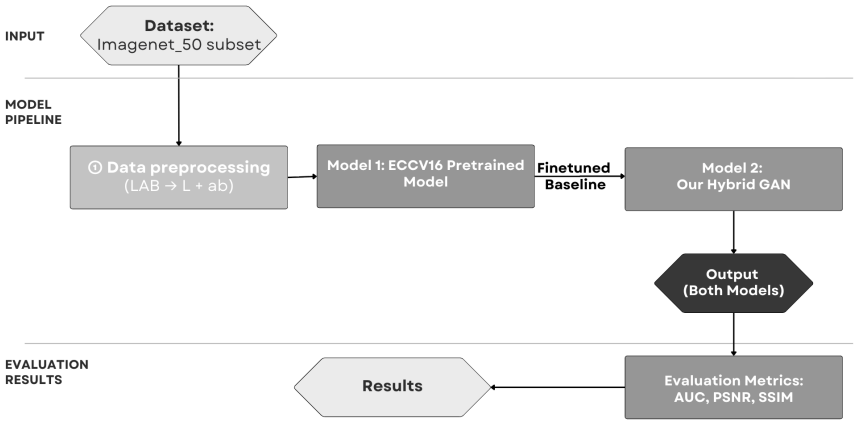


Figure 4. Outline of our Project

V. Experimental Setup

A. Software Setup

Our models were developed and executed using a Python-based deep learning environment built around the PyTorch framework. All experiments were run on a GPU-enabled machine to ensure efficient training and inference.

1. Programming Environment
Python: 3.11
PyTorch: 2.x
Torchvision: compatible version for VGG19
CUDA: (if available) for GPU acceleration
Additional Libraries:
NumPy
SciPy / scikit-image (skimage.color)
PIL (Pillow) for image handling
tqdm for progress bars
pandas / matplotlib for visualization
torchvision.models for VGG19 feature extraction

2. Hardware Configuration
GPU: NVIDIA CUDA-enabled GPU
(Training the GAN generator + discriminator requires GPU for feasible runtimes)
CPU: Used for dataloading and preprocessing
Memory: At least 8 - 12 GB GPU VRAM recommended

3. Code Structure

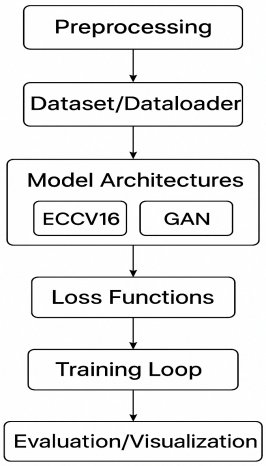


Figure 5. Structure of Code file

B. Hyperparameter Configuration

a. FineTuned Model: We tried fine-tuning the model using 2 different methods

(i) MSE Loss - We tried training the model using MSE loss with output of ECCV16 which is a continuous ab output

(ii) CrossEntropyLoss + Class rebalancing - We also tried training the model using class rebalancing and cross entropy loss where we predict the quantised ab values.

Final Chosen Model: Unlike the original classification-based training, we use Mean Squared Error (MSE) loss for fine-tuning: $MSE = (1/N) \Sigma ||ab\_pred - ab\_gt||^2$. This direct regression approach is suitable because the pretrained model's final layer already outputs continuous ab values. MSE provides straightforward pixel-wise supervision for adapting the color predictions to our specific dataset distribution.

Parameters:

i. Learning Rate: 0.0001
ii. Optimizer: Adam
iii. Epochs: 3
iv. Batch Size: 16

b. *Hybrid GAN model:* We tried various parameters while checking the validation inference of the models with parameters like

Parameters:

i. Learning Rate Generator, Learning Rate Discriminator Optimizer: Adam
ii. Epochs: 50
iii. Batch Size: 16
iv. Image size - 256
v. Lambda L1 - We tried Lambda values 1, 10, 100 we found that 10 is the most optimal value with highest PSNR value, SSIM.

C. Mini Batch Strategy

a. FineTuned Mode: Batch Size - 16, We did not want to modify the pre-trained weights too much hence trained it for only 3 epochs

b. Hybrid GAN Model: Batch Size - 16, We tried various combinations, as we used PatchGAN discirminator we tried using different batch sizes 1, 4, 16 and we found for our model Batch size of 16 is most optimal

D. *Learning Rate Selection*

a. *Finetuned Model:* 0.0001(Learning Rate)

b. *Hybrid GAN Model:* : We tried various different combinations with the final combination to make generator stronger LR_G(0.0002), LR_D(0.0001)

E. *Overfitting and Extrapolation Prevention*

a. *FineTuned Model:* The pretrained ECCV16 model was originally trained on 1.3M+ ImageNet images, giving it strong general priors. To avoid overwriting this broad color knowledge with our significantly smaller dataset, we deliberately:

- Fine-tuned for only 3 epochs
- Used a low learning rate (1e-4)
- Kept batch size modest (16) to maintain gradient stability
- Avoided class-rebalancing during fine-tuning, since rebalancing amplifies rare colors and can cause over-saturation when training data is small

b. Hybrid GAN model: GANs naturally risk discriminator overfitting, which leads to generator collapse. To mitigate this:

- We used PatchGAN, which evaluates 70×70 patches independently. This forces the discriminator to focus on local inconsistencies rather than memorizing entire images.
- We alternated 1:1 training steps for G and D, stabilizing adversarial gradients. Additionally, the L1 loss acts as a strong regularizer, constraining the generator to remain close to ground-truth ab colors and preventing unrealistic hallucinations.
- We used Perceptual Loss for a more generalised model
- We used differential learning rates:
  - Generator LR = 2e-4
  - Discriminator LR = 1e-4. The slower discriminator learning helps prevent D from overpowering G and overfitting.

F. *Implementation Details:* Our system was implemented entirely in PyTorch using a modular structure separating preprocessing, model definitions, dataset utilities, loss functions, GAN training loops, and inference components

Training Pipeline:

Load LAB images → extract L (input) and ab (ground truth)
Train discriminator on:

- Real LAB pairs
- Fake LAB pairs (generated)

Train generator on:

- Adversarial loss
- L1 loss
- Perceptual loss

Reconstruct LAB → RGB for output visualization
Our final GAN was trained for 50 epochs using:

- Generator LR = 2e-4
- Discriminator LR = 1e-4
- Adam optimizer, β1=0.5

Inference Pipeline:

For both models:

Load L channel from input

Generate ab (ECCV16 → continuous ab; GAN → continuous ab)

Concatenate L with predicted ab

Convert LAB → RGB

Save final image

This unified inference process allows direct visual comparisons.

VI. RESULTS AND DISCUSSION

To evaluate the models, we considered both quantitative metrics and visual inspection. We compared the enhanced GAN model against the baseline ECCV‑16 colourization network and report the area under the colour error curve (AUC) in CIELAB ab space, peak signal‑to‑noise ratio (PSNR), and structural similarity index (SSIM) in RGB space.

A. Quantitative/Qualitative Results

Evaluation metric	Pretrained ECCV‑16 (baseline)	Enhanced GAN (ours)
AUC	0.89	0.91
PSNR	22.27	23.76
SSIM	0.91	0.92

The GAN‑based model outperformed the baseline across all metrics. The AUC increased from 0.89 to 0.91 meaning that a larger proportion of pixels in the GAN model outputs have small colour errors. PSNR improvement reflects lower mean squared error in the RGB domain. The SSIM gain suggests better structural consistency and perceptual realism.

GAN loss:

Figure 6. illustrates the four main losses monitored during training. The generator loss generally decreases but is relatively unstable due to fluctuations, which is expected in adversarial training where the generator continuously adapts to the discriminator. The discriminator loss, by contrast, rapidly converges after the first few epochs and remains nearly flat, indicating that the discriminator becomes confident at distinguishing real from generated colour distributions. The L1 reconstruction loss exhibits a clear downward trend throughout training, reflecting progressively improved pixel-wise accuracy of the generated channels. Finally, the adversarial component of the generator loss shows an upward drift, suggesting that the generator is successfully learning to fool the discriminator by producing more realistic colour statistics. Together, these curves confirm effective GAN interaction and show that the combined objective is capable of improving both L1 and GAN loss.

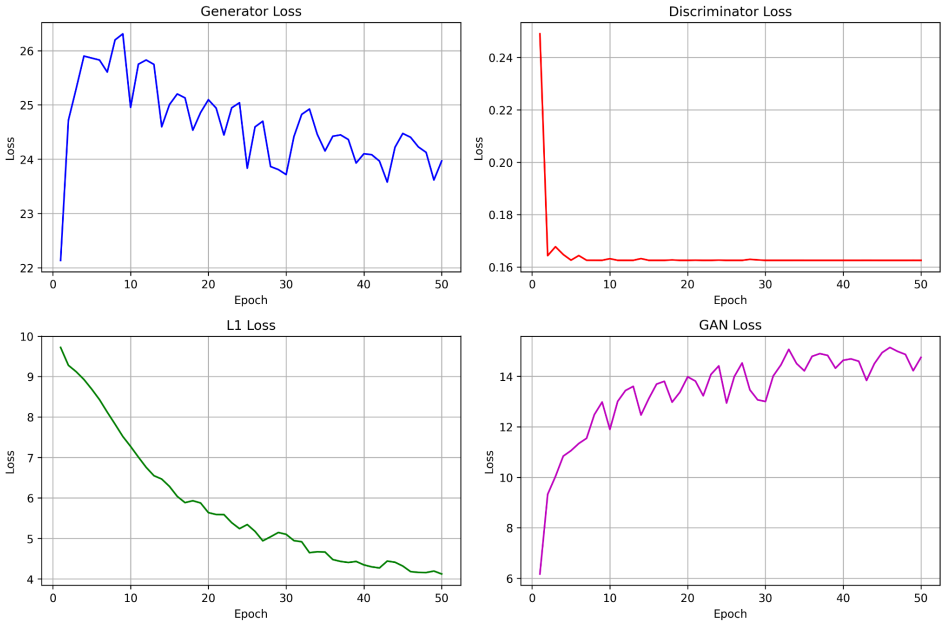
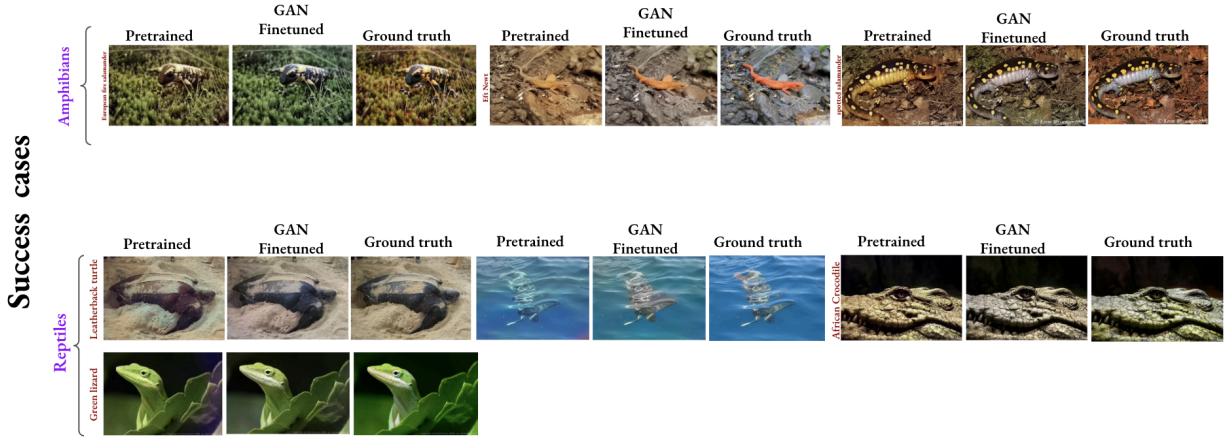


Figure 6. GAN training loss

Visual inspection complements these metrics. *Figure 7.* shows example images comparing the pretrained baseline, our fine-tuned GAN, and the ground truth. Across diverse classes (animals, vehicles, household objects), the GAN model produces more vibrant and plausible colours. For instance, aquatic animals such as goldfish and sharks are colourised with appropriate orange or grey hues, while the baseline often produces desaturated or off-hue results.



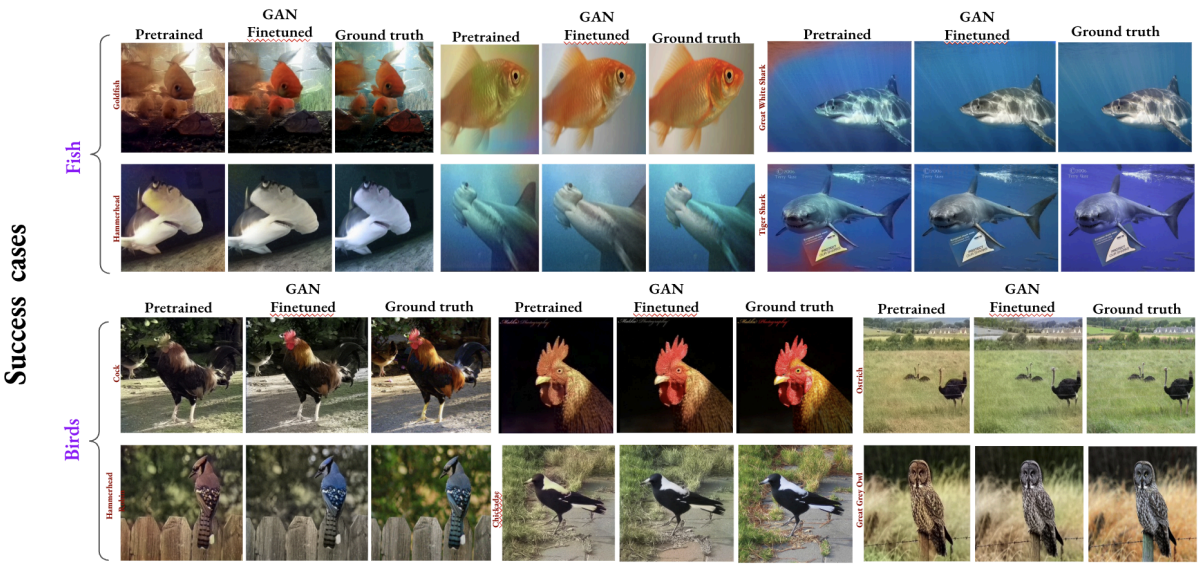


Figure 7. Enhanced GAN model success cases

B. Baseline vs GAN Model Comparison

The ECCV-16 baseline (Zhang et al.) treats colourisation as a classification problem over quantised ab channels. Its predictions are generally smooth but lack richness. It relies solely on pixel-wise and classification losses. There is no explicit mechanism to enforce global colour coherence.

The enhanced GAN model addresses these limitations. The discriminator operates to encourage the generator to produce locally plausible colour textures. Combining adversarial loss with L1 and perceptual (VGG19-based) loss yields a balanced objective. Consequently, the model generates richer palettes and higher contrast. In success cases such as birds, reptiles, and amphibians, the GAN colourizations are much closer to the ground truth. The baseline, however, often leaves feathers dull or reptile skin colourless.

C. Error Analysis

Despite overall improvements, the GAN model exhibits notable failure outputs. Colourization remains ambiguous for certain objects. For example, complex scenes with multiple objects confuse the generator, such as background color may bleed into foreground objects, or colours may spill across edges. Failure cases are highlighted in *Figure 8*. These errors can be attributed to:

- Dataset bias: ImageNet-50 contains mostly natural images. When presented with textures or objects outside this domain, the model overfits to colours seen during training.
- Ambiguous colour semantics: Many objects (clothing or vehicles) can appear in multiple colours.

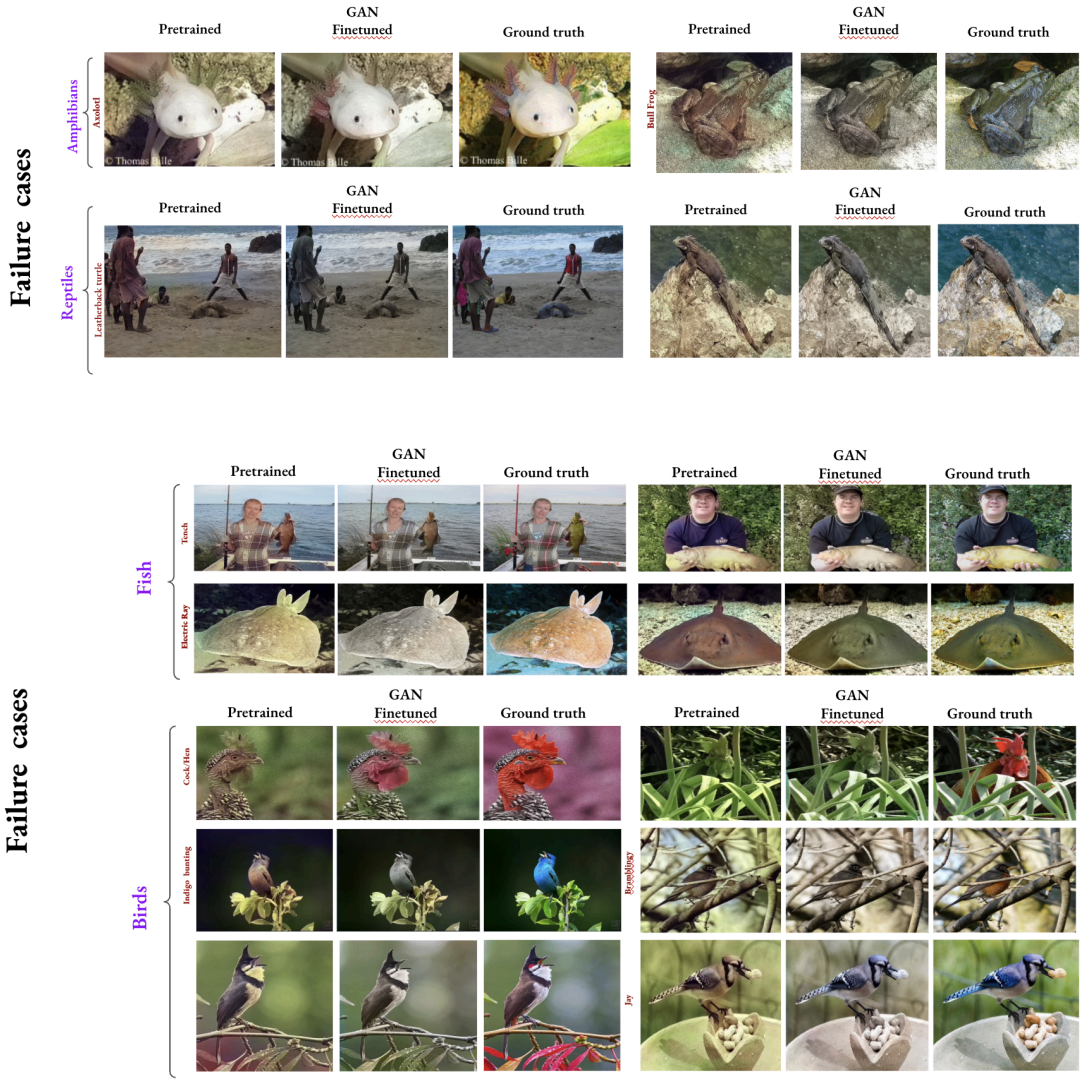


Figure 8. Enhanced GAN model success cases

D. Interpretation of Results

The quantitative gains in AUC, PSNR, and SSIM demonstrate that adversarial learning leads to better pixel-level and perceptual accuracy. Higher AUC suggests the distribution of colour errors is shifted towards smaller values; the GAN generator is penalised for large deviations and learns to produce more accurate colours on average. Improved PSNR indicates reduced mean squared error, while the SSIM gain signifies better preservation of structures such as edges and textures. However, these metrics should be interpreted cautiously. PSNR is sensitive to small luminance differences and does not correlate perfectly with human perception. Likewise, SSIM captures structural similarity but cannot account for semantic plausibility, meaning an image with the "right" colour in the wrong location can score highly. Visual evaluation, therefore, remains crucial for colourisation tasks.

The error analysis reveals that colourisation is inherently ambiguous, many grayscale inputs admit multiple valid colourisations. The GAN framework attempts to model the natural colour distribution, but the adversarial training is only as good as the diversity of the training data. For categories with wide colour variance, the model tends to average over modes, leading to washed-out results. Incorporating higher-level semantic guidance (class labels or text descriptions) could mitigate this ambiguity by conditioning the generator on additional context.

VII. SUMMARY AND CONCLUSION

A. Summary of Findings

This project explored the automatic colourisation of grayscale images using deep neural networks. We implemented and trained a baseline model following the ECCV-16 approach and subsequently fine-tuned a generator within a GAN framework. The training procedure combined pixel-wise L1 loss, perceptual loss, and adversarial loss. We evaluated the models on the ImageNet-50 validation set using AUC, PSNR, and SSIM. Compared to the pretrained ECCV-16 network, the GAN model improved AUC from 0.89 to 0.91, PSNR from 22.27 to 23.76 and SSIM from 0.91 to 0.92. Qualitatively, the GAN outputs display richer colours, better contrast, and greater perceptual realism across samples.

Key Observations:

- Effectiveness of perceptual and adversarial losses. Combining content loss with GAN loss encourages the generator to respect both global structure and local texture. This produces more convincing colour distributions than the baseline classification-based model.
- Improved metrics. Although quantitative improvements appear, human observers preferred the GAN colourizations on most results, however, for a certain number of samples, they preferred baseline model output. This emphasises that perceptual quality cannot be fully captured by numeric metrics.

- Semantic awareness is limited: The model learns statistical colour priors from ImageNet but lacks explicit understanding of object semantics. Ambiguous or multi-coloured objects are often miscoloured. Without external cues, the network defaults to frequent colours in the training set.

B. Limitations

Although the proposed GAN colorization approach demonstrated perceptual improvements over the pretrained baseline, several limitations remain important to acknowledge when interpreting these results. In particular, the current model configuration and evaluation setup introduce constraints that may have affected both generalization and quantitative performance.

1. Data and training: Our training used a fixed 256×256 resolution and did not employ extensive data augmentation. A small batch size (16) and the absence of validation set, early stopping may have limited generalization and overfit the model. Moreover, the model was trained on ImageNet-50 dataset, therefore, its performance may deteriorate on photographs with different statistics.

2. Metric limitations: PSNR and SSIM provide useful proxies but do not perfectly align with human perception. The GAN output obtains higher PSNR despite obvious colour errors being present in some cases. Future work should consider perceptual metrics based on learned similarity models.

3. Ambiguity in colours: Many grayscale inputs admit multiple valid colourisations. Our generator produces a single guess, which may not coincide with the ground truth. An interactive or multimodal system could allow users to guide the colourisation process.

C. Future Improvements

Building upon the findings of this work, several promising directions could further enhance performance and resolve the limitations identified above. The following ideas outline possible avenues for exploration, both in methodology and in evaluation design:

1. Data augmentation and larger datasets. Augmenting the training set with random flips, crops, and colour jittering could improve robustness. Training on larger and diverse datasets would expose the model to more diverse colour distributions.
2. Semantic conditioning. Extend the generator to incorporate class labels or semantic maps as additional inputs, enabling object-aware colorization and reducing unrealistic colors for semantically coherent regions.
3. Richer perceptual evaluation. Future work could integrate more metrics to better capture subjective realism beyond pixel-based metrics such as PSNR and SSIM.

In summary, the enhanced GAN model advanced the state of automatic colourisation by combining adversarial and perceptual learning. While quantitative improvements are modest, the perceptual gains are significant and opens further exploration of conditional and multimodal approaches to address the remaining challenges.

Beyond the core experiments, an interactive demonstration of the model has been deployed as a Streamlit application, allowing users to upload grayscale photographs and visualize colourization results with the pretrained baseline model and our enhanced GAN model. The application is publicly accessible at: https://dl-group1-colorization-app.streamlit.app/

References

[1] Zhang, R., Isola, P., & Efros, A. (2016), "Colorful Image Colorization," *arXiv preprint arXiv:1603.08511*.
[2] Ly, Bao & Dyer, Ethan & Feig, Jessica & Chien, Anna & Bino, Sandra. (2020). Research Techniques Made Simple: Cutaneous Colorimetry: A Reliable Technique for Objective Skin Color Measurement. The Journal of investigative dermatology. 140. 3-12.e1. 10.1016/j.jid.2019.11.003.
[3] Nazeri, K., Ng, E., & Ebrahimi, M., "Image Colorization Using Generative Adversarial Networks," in *Articulated Motion and Deformable Objects: AMDO 2018*, F. Perales & J. Kittler, Eds., Lecture Notes in Computer Science, vol. 10945, Springer, Cham, 2018, pp. 85–94.
[4] Isola, P., Zhu, J.-Y., Zhou, T., & Efros, A. A. (2017). Image-to-image translation with conditional adversarial networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 1125–1134). https://doi.org/10.1109/CVPR.2017.632
[5] Foun, M. H. (2024). Application and analysis of black and white image coloring based on generative adversarial networks (GANs). In Proceedings of the 1st International Conference on Engineering Management, Information Technology and Intelligence (EMITI 2024) (pp. 354–361). SCITEPRESS. https://doi.org/10.5220/0012938100004508
[6] Johnson, J., Alahi, A., & Fei-Fei, L. (2016). Perceptual losses for real-time style transfer and super-resolution. In Proceedings of the European Conference on Computer Vision (ECCV) (pp. 694–711). https://doi.org/10.1007/978-3-319-46475-6_43
[7] Qiu, C., Cao, H., Ren, Q., Li, R., & Qiu, Y. (2025). Automatic Image Colorization with Convolutional Neural Networks and Generative Adversarial Networks. arXiv preprint arXiv:2508.05068. https://arxiv.org/abs/2508.05068

APPENDIX A - Code Implementation

A.1 Fine-tuned Pretrained Model Script

```
# ================================================================
# COLORIZATION OF GRAYSCALE IMAGES USING DEEP NEURAL NETWORKS
# ================================================================

import os
import numpy as np
from PIL import Image
from skimage import color
from tqdm import tqdm
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader

# ------------------------------------------------
# CONFIG
# ------------------------------------------------
# Resize images to (256x256)
IMAGE_SIZE = 256
# EPOCHS = 1
EPOCHS = 3  # More epochs for GAN training
LR = 0.00001
BATCH_SIZE = 16
DEVICE = "cuda:0" if torch.cuda.is_available() else "cpu"
# ------------------------------------------------
# DEFINE UTIL FUNCTIONS
# ------------------------------------------------
```

```
# The function to center and normalise the images
class BaseColor(nn.Module):
    def __init__(self):
        super(BaseColor, self).__init__()
        self.l_cent = 50.
        self.l_norm = 100.
        self.ab_norm = 110.

    def normalize_l(self, in_l):
        return (in_l - self.l_cent) / self.l_norm

    def unnormalize_l(self, in_l):
        return in_l * self.l_norm + self.l_cent

    def normalize_ab(self, in_ab):
        return in_ab / self.ab_norm

    def unnormalize_ab(self, in_ab):
        return in_ab * self.ab_norm

normalizer = BaseColor()
def load_img(img_path):
    """
    Load an image from disk as a numpy RGB array.
    If grayscale, convert to 3-channel RGB by tiling.
    """
    out_np = np.asarray(Image.open(img_path))
    if out_np.ndim == 2:
        # if grayscale → replicate channel 3 times
        out_np = np.tile(out_np[:, :, None], 3)
    return out_np

class ECCVGenerator(BaseColor):
    def __init__(self, norm_layer=nn.BatchNorm2d):
        super(ECCVGenerator, self).__init__()

        # Model 1: 1 -> 64, downsample to 128x128
        model1 = [nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1, bias=True),]
        model1 += [nn.ReLU(True),]
        model1 += [nn.Conv2d(64, 64, kernel_size=3, stride=2, padding=1, bias=True),]
        model1 += [nn.ReLU(True),]
        model1 += [norm_layer(64),]

        # Model 2: 64 -> 128, downsample to 64x64
        model2 = [nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1, bias=True),]
        model2 += [nn.ReLU(True),]
        model2 += [nn.Conv2d(128, 128, kernel_size=3, stride=2, padding=1, bias=True),]
        model2 += [nn.ReLU(True),]
        model2 += [norm_layer(128),]

        # Model 3: 128 -> 256, downsample to 32x32
        model3 = [nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1, bias=True),]
        model3 += [nn.ReLU(True),]
        model3 += [nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=True),]
```

```
model3 += [nn.ReLU(True),]
model3 += [nn.Conv2d(256, 256, kernel_size=3, stride=2, padding=1, bias=True),]
model3 += [nn.ReLU(True),]
model3 += [norm_layer(256),]

# Model 4: 256 -> 512, stay at 32x32
model4 = [nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1, bias=True),]
model4 += [nn.ReLU(True),]
model4 += [nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1, bias=True),]
model4 += [nn.ReLU(True),]
model4 += [nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1, bias=True),]
model4 += [nn.ReLU(True),]
model4 += [norm_layer(512),]

# Model 5: Dilated convolutions
model5 = [nn.Conv2d(512, 512, kernel_size=3, dilation=2, stride=1, padding=2, bias=True),]
model5 += [nn.ReLU(True),]
model5 += [nn.Conv2d(512, 512, kernel_size=3, dilation=2, stride=1, padding=2, bias=True),]
model5 += [nn.ReLU(True),]
model5 += [nn.Conv2d(512, 512, kernel_size=3, dilation=2, stride=1, padding=2, bias=True),]
model5 += [nn.ReLU(True),]
model5 += [norm_layer(512),]

# Model 6: Dilated convolutions
model6 = [nn.Conv2d(512, 512, kernel_size=3, dilation=2, stride=1, padding=2, bias=True),]
model6 += [nn.ReLU(True),]
model6 += [nn.Conv2d(512, 512, kernel_size=3, dilation=2, stride=1, padding=2, bias=True),]
model6 += [nn.ReLU(True),]
model6 += [nn.Conv2d(512, 512, kernel_size=3, dilation=2, stride=1, padding=2, bias=True),]
model6 += [nn.ReLU(True),]
model6 += [norm_layer(512),]

# Model 7: Standard convolutions
model7 = [nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1, bias=True),]
model7 += [nn.ReLU(True),]
model7 += [nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1, bias=True),]
model7 += [nn.ReLU(True),]
model7 += [nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1, bias=True),]
model7 += [nn.ReLU(True),]
model7 += [norm_layer(512),]

# Model 8: Decoder
model8 = [nn.ConvTranspose2d(512, 256, kernel_size=4, stride=2, padding=1, bias=True),]
model8 += [nn.ReLU(True),]
model8 += [nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=True),]
model8 += [nn.ReLU(True),]
model8 += [nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=True),]
model8 += [nn.ReLU(True),]
model8 += [nn.Conv2d(256, 313, kernel_size=1, stride=1, padding=0, bias=True),]

self.model1 = nn.Sequential(*model1)
self.model2 = nn.Sequential(*model2)
self.model3 = nn.Sequential(*model3)
self.model4 = nn.Sequential(*model4)
```

```
        self.model5 = nn.Sequential(*model5)
        self.model6 = nn.Sequential(*model6)
        self.model7 = nn.Sequential(*model7)
        self.model8 = nn.Sequential(*model8)

        self.softmax = nn.Softmax(dim=1)
        self.model_out = nn.Conv2d(313, 2, kernel_size=1, padding=0, dilation=1, stride=1, bias=False)
        self.upsample4 = nn.Upsample(scale_factor=4, mode='bilinear')

    def forward(self, input_l):
        conv1_2 = self.model1(self.normalize_l(input_l))
        conv2_2 = self.model2(conv1_2)
        conv3_3 = self.model3(conv2_2)
        conv4_3 = self.model4(conv3_3)
        conv5_3 = self.model5(conv4_3)
        conv6_3 = self.model6(conv5_3)
        conv7_3 = self.model7(conv6_3)
        conv8_3 = self.model8(conv7_3)
        out_reg = self.model_out(self.softmax(conv8_3))

        return self.unnormalize_ab(self.upsample4(out_reg))



def resize_img(img, HW=(256, 256), resample=Image.BILINEAR):
    """
    Resize numpy image to (HW[0], HW[1]).
    """
    return np.asarray(
        Image.fromarray(img).resize((HW[1], HW[0]), resample=resample)
    )


def preprocess_img(img_rgb_orig, HW=(256, 256), resample=Image.BILINEAR, return_ab = False):
    """
    Preprocess image into L_orig (original size) and L_rs (resized).
    Returns:
      tens_orig_l : (1,1,H_orig,W_orig) -> original L channel tensor
      tens_rs_l   : (1,1,HW[0],HW[1]) -> resized L channel tensor
    """
    # Resize original
    img_rgb_rs = resize_img(img_rgb_orig, HW=HW, resample=resample)

    # Convert both original & resized images to LAB
    img_lab_orig = color.rgb2lab(img_rgb_orig)
    img_lab_rs   = color.rgb2lab(img_rgb_rs)

    # Extract only L channel
    img_l_orig = img_lab_orig[:, :, 0]
    img_l_rs   = img_lab_rs[:, :, 0]

    # Convert to torch tensors with shape (1,1,H,W)
    tens_orig_l = torch.tensor(img_l_orig, dtype=torch.float32)[None, :, :]
    tens_rs_l   = torch.tensor(img_l_rs,   dtype=torch.float32)[None, :, :]
```

```
    if return_ab:
        img_ab_rs = img_lab_rs[:, :, 1:]
        tens_ab = torch.tensor(img_ab_rs, dtype=torch.float32).permute(2, 0, 1)

        return tens_rs_l, tens_ab

    return tens_orig_l, tens_rs_l



def postprocess_tens(tens_orig_l, out_ab, mode='bilinear'):
    """
    Combine predicted ab channels with original L channel and convert back to RGB.
    """

    HW_orig = tens_orig_l.shape[1:]          # (H_orig, W_orig)
    HW_pred = out_ab.shape[1:]               # (H_pred, W_pred)

    # If needed, resize ab to match original size
    if HW_pred != HW_orig:
        out_ab = F.interpolate(out_ab.unsqueeze(0),
                               size=HW_orig, mode='bilinear')[0]


    out_lab = torch.cat((tens_orig_l, out_ab), dim=0)  # (3, H_orig, W_orig)

    out_lab_np = out_lab.cpu().numpy().transpose((1,2,0))
    out_rgb = color.lab2rgb(out_lab_np)

    return out_rgb

# ------------------------------------------------
# DEFINE DATASET CLASS
# ------------------------------------------------

class ColorizationDataset(Dataset):
    """
    Dataset for automatic colorization.
    Loads images from folder, applies preprocessing,
    returns (L_resized, L_original, img_path).
    """
    def __init__(self, folder_path, pretrained = False, mode = 'inference'):
        self.image_paths = []
        self.pretrained = pretrained
        self.mode = mode

        for fname in os.listdir(folder_path):
            if fname.lower().endswith((".jpg", ".jpeg", ".png")):
                self.image_paths.append(os.path.join(folder_path, fname))

        self.image_paths.sort()

    def __len__(self):
```

```
        return len(self.image_paths)

    def __getitem__(self, index):
        img_path = self.image_paths[index]

        # Load image to numpy RGB
        img_rgb_orig = load_img(img_path)

        # Apply preprocessing
        if self.mode == 'training':
            # Get L and ab for training
            tens_l, tens_ab = preprocess_img(img_rgb_orig, HW=(IMAGE_SIZE, IMAGE_SIZE),
                                             return_ab=True)
            return tens_l, tens_ab
        else:
            # Get L only for inference
            tens_orig_l, tens_rs_l = preprocess_img(img_rgb_orig, HW=(IMAGE_SIZE, IMAGE_SIZE),
                                                    return_ab=False)
            return tens_rs_l, tens_orig_l, img_path

        # Return:
        # - resized L (input for model)
        # - original L (for reconstruction)
        # - image path

def colorization_collate(batch):
    """
    Allows batching resized L (fixed size),
    while keeping original L tensors in a list.
    """
    tens_rs_l_batch = torch.stack([b[0] for b in batch], dim=0)   # batchable
    tens_orig_l_list = [b[1] for b in batch]                      # NOT stacked
    img_paths = [b[2] for b in batch]                             # list of paths

    return tens_rs_l_batch, tens_orig_l_list, img_paths

# ------------------------------------------------
# DEFINE MODEL ARCHITECTURE & TRAINING LOOP
# ------------------------------------------------

def define_colorization_model(pretrained = True):
    """
    TODO:
    - Build U-Net / encoder-decoder model
    - Output: predicted ab channels (1,2,256,256)
    """
    model = ECCVGenerator()

    if pretrained:
        import torch.utils.model_zoo as model_zoo
        model.load_state_dict(
            model_zoo.load_url(
                'https://colorizers.s3.us-east-2.amazonaws.com/colorization_release_v2-9b330a0b.pth',
                map_location='cpu',
```

```
            check_hash=True
        )
    )


    model.to(DEVICE)
    return model
    pass


def train_colorization_model(
        model,
        data_folder,
        save_images=False,
        output_folder="training_output",
        finetune=True,
        inference_count= 50
):


    """
    Training + inference pipeline.
    If finetune=False → skip training and run inference only.
    inference_count: number of random images to colorize.
    """

    # ----------------------------------------
    # TRAINING PHASE if finetune=True)
    # ----------------------------------------
    if finetune:
        print(f"Finetune = True , Starting training for {EPOCHS} epochs...")

        train_dataset = ColorizationDataset(data_folder, mode='training', pretrained=False)
        train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)

        model.train()
        optimizer = optim.Adam(model.parameters(), lr=LR)
        criterion = nn.MSELoss()

        if save_images:
            os.makedirs(output_folder, exist_ok=True)

        for epoch in range(EPOCHS):
            running_loss = 0.0

            for batch_idx, (tens_l, tens_ab_gt) in enumerate(
                    tqdm(train_loader, desc=f"Epoch {epoch + 1}/{EPOCHS}")
            ):
                tens_l = tens_l.to(DEVICE)
                tens_ab_gt = tens_ab_gt.to(DEVICE)

                optimizer.zero_grad()
                predicted_ab = model(tens_l)
                loss = criterion(predicted_ab, tens_ab_gt)
```

```
            loss.backward()
            optimizer.step()

            if epoch == 0 and batch_idx == 0:
                print(
                    f"predicted_ab stats: min={predicted_ab.min():.2f}, "
                    f"max={predicted_ab.max():.2f}, mean={predicted_ab.mean():.2f}"
                )
                print(
                    f"tens_ab_gt stats: min={tens_ab_gt.min():.2f}, "
                    f"max={tens_ab_gt.max():.2f}, mean={tens_ab_gt.mean():.2f}"
                )
                print(f"Loss: {loss.item():.4f}")

            running_loss += loss.item()

            if save_images and batch_idx % 500 == 0:
                with torch.no_grad():
                    for i in range(min(2, tens_l.shape[0])):
                        orig_l = tens_l[i].cpu()
                        pred_ab = predicted_ab[i].cpu()
                        colorized_rgb = postprocess_tens(orig_l, pred_ab)
                        save_path = os.path.join(
                            output_folder,
                            f"epoch{epoch + 1}_batch{batch_idx}_img{i}.png"
                        )
                        Image.fromarray((colorized_rgb * 255).astype(np.uint8)).save(save_path)

        avg_loss = running_loss / len(train_loader)
        print(f"Epoch [{epoch + 1}/{EPOCHS}] - Loss: {avg_loss:.4f}")

    print("Training completed.")

    # Save fine tuned weights
    torch.save(model.state_dict(), 'colorization_model_finetuned.pth')
    print("Finetuned model saved as colorization_model_finetuned.pth")

else:
    print("Finetune = False, Skipping training. Running inference only.")

# ----------------------------------------
# INFERENCE PHASE
# ----------------------------------------
print(f"\nStarting inference on: {data_folder}")

# Load full inference dataset
full_dataset = ColorizationDataset(data_folder, mode='inference', pretrained=False)

# Limit inference to N random images
if inference_count is not None:
    print(f"Sampling {inference_count} random images for inference...")
    indices = torch.randperm(len(full_dataset))[:inference_count]
    inference_dataset = torch.utils.data.Subset(full_dataset, indices)
else:
```

```
        inference_dataset = full_dataset

    inference_loader = DataLoader(
        inference_dataset,
        batch_size=4,
        shuffle=False,
        collate_fn=colorization_collate
    )

    final_output = output_folder + "_final"
    os.makedirs(final_output, exist_ok=True)

    model.eval()
    with torch.no_grad():
        for batch_idx, (tens_rs_l, tens_orig_l, img_paths) in enumerate(inference_loader):
            tens_rs_l = tens_rs_l.to(DEVICE)
            predicted_ab = model(tens_rs_l)

            for i in range(len(img_paths)):
                pred_ab = predicted_ab[i].cpu()
                orig_l = tens_orig_l[i]

                colorized_rgb = postprocess_tens(orig_l, pred_ab)

                filename = os.path.basename(img_paths[i])
                save_path = os.path.join(final_output, f"colorized_{filename}")
                Image.fromarray((colorized_rgb * 255).astype(np.uint8)).save(save_path)

            print(f"Batch {batch_idx + 1}/{len(inference_loader)} colorized")

    print(f"Saved {len(inference_dataset)} images → {final_output}")

    return model

# ------------------------------------------------
# MAIN BLOCK
# ------------------------------------------------

if __name__ == "__main__":
    #Configuration
    DATA_FOLDER = "imagenet_50/train"  # Change this to your folder
    OUTPUT_FOLDER = "colorized_output"

    # Create model
    model = define_colorization_model(pretrained=True)

    # Train the model (already includes inference at the end)
    trained_model = train_colorization_model(
        model,
        DATA_FOLDER,
        save_images=True,
        finetune = True,
        output_folder=OUTPUT_FOLDER
    )
```

A.2 GAN Training Script

```
# ================================================================
# COLORIZATION OF GRAYSCALE IMAGES USING DEEP NEURAL NETWORKS
# ENHANCED WITH GENERATIVE ADVERSARIAL NETWORKS (GANs)
# ================================================================

import os
import numpy as np
from PIL import Image
from skimage import color
from tqdm import tqdm
import torch
import torch.nn as nn
import torchvision.models as models
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from datasets import load_dataset
from collections import defaultdict

# ------------------------------------------------
# CONFIG
# ------------------------------------------------
# Resize images to (256x256)
IMAGE_SIZE = 256
EPOCHS = 50  # More epochs for GAN training
LR_G = 0.0002  # Generator learning rate
LR_D = 0.0001  # Discriminator learning rate
BATCH_SIZE = 16
DEVICE = "cuda:0" if torch.cuda.is_available() else "cpu"

LAMBDA_L1 = 10  # Weight for L1 loss
D_STEPS = 1  # Discriminator steps per generator step
LABEL_SMOOTHING = 0.1  # One-sided label smoothing for real images

import csv
import matplotlib.pyplot as plt

# Create CSV file for logging
def init_loss_log(filename='training_losses.csv'):
    with open(filename, 'w', newline='') as f:
        writer = csv.writer(f)
        writer.writerow(['Epoch', 'G_Loss', 'D_Loss', 'L1_Loss', 'GAN_Loss'])

def log_losses(epoch, g_loss, d_loss, l1_loss, gan_loss, filename='training_losses.csv'):
    with open(filename, 'a', newline='') as f:
        writer = csv.writer(f)
        writer.writerow([epoch, g_loss, d_loss, l1_loss, gan_loss])

# ------------------------------------------------
# DEFINE UTIL FUNCTIONS
# ------------------------------------------------

class FeatureExtractor(nn.Module):
```

```
    def __init__(self):
        super(FeatureExtractor, self).__init__()
        vgg19 = models.vgg19(pretrained=True)
        self.feature_extractor = nn.Sequential(*list(vgg19.features.children())[:18])

    def forward(self, img):
        return self.feature_extractor(img)


# The function to center and normalise the images
class BaseColor(nn.Module):
    def __init__(self):
        super(BaseColor, self).__init__()
        self.l_cent = 50.
        self.l_norm = 100.
        self.ab_norm = 110.

    def normalize_l(self, in_l):
        return (in_l - self.l_cent) / self.l_norm

    def unnormalize_l(self, in_l):
        return in_l * self.l_norm + self.l_cent

    def normalize_ab(self, in_ab):
        return in_ab / self.ab_norm

    def unnormalize_ab(self, in_ab):
        return in_ab * self.ab_norm


normalizer = BaseColor()


def load_img(img_path):
    """
    Load an image from disk as a numpy RGB array.
    If grayscale, convert to 3-channel RGB by tiling.
    """
    out_np = np.asarray(Image.open(img_path))
    if out_np.ndim == 2:
        # if grayscale → replicate channel 3 times
        out_np = np.tile(out_np[:, :, None], 3)
    return out_np


class ECCVGenerator(BaseColor):
    def __init__(self, norm_layer=nn.BatchNorm2d):
        super(ECCVGenerator, self).__init__()

        # Model 1: 1 -> 64, downsample to 128x128
        model1 = [nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1, bias=True), ]
        model1 += [nn.ReLU(True), ]
        model1 += [nn.Conv2d(64, 64, kernel_size=3, stride=2, padding=1, bias=True), ]
        model1 += [nn.ReLU(True), ]
```

```
model1 += [norm_layer(64), ]

# Model 2: 64 -> 128, downsample to 64x64
model2 = [nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1, bias=True), ]
model2 += [nn.ReLU(True), ]
model2 += [nn.Conv2d(128, 128, kernel_size=3, stride=2, padding=1, bias=True), ]
model2 += [nn.ReLU(True), ]
model2 += [norm_layer(128), ]

# Model 3: 128 -> 256, downsample to 32x32
model3 = [nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1, bias=True), ]
model3 += [nn.ReLU(True), ]
model3 += [nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=True), ]
model3 += [nn.ReLU(True), ]
model3 += [nn.Conv2d(256, 256, kernel_size=3, stride=2, padding=1, bias=True), ]
model3 += [nn.ReLU(True), ]
model3 += [norm_layer(256), ]

# Model 4: 256 -> 512, stay at 32x32
model4 = [nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1, bias=True), ]
model4 += [nn.ReLU(True), ]
model4 += [nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1, bias=True), ]
model4 += [nn.ReLU(True), ]
model4 += [nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1, bias=True), ]
model4 += [nn.ReLU(True), ]
model4 += [norm_layer(512), ]

# Model 5: Dilated convolutions
model5 = [nn.Conv2d(512, 512, kernel_size=3, dilation=2, stride=1, padding=2, bias=True), ]
model5 += [nn.ReLU(True), ]
model5 += [nn.Conv2d(512, 512, kernel_size=3, dilation=2, stride=1, padding=2, bias=True), ]
model5 += [nn.ReLU(True), ]
model5 += [nn.Conv2d(512, 512, kernel_size=3, dilation=2, stride=1, padding=2, bias=True), ]
model5 += [nn.ReLU(True), ]
model5 += [norm_layer(512), ]

# Model 6: Dilated convolutions
model6 = [nn.Conv2d(512, 512, kernel_size=3, dilation=2, stride=1, padding=2, bias=True), ]
model6 += [nn.ReLU(True), ]
model6 += [nn.Conv2d(512, 512, kernel_size=3, dilation=2, stride=1, padding=2, bias=True), ]
model6 += [nn.ReLU(True), ]
model6 += [nn.Conv2d(512, 512, kernel_size=3, dilation=2, stride=1, padding=2, bias=True), ]
model6 += [nn.ReLU(True), ]
model6 += [norm_layer(512), ]

# Model 7: Standard convolutions
model7 = [nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1, bias=True), ]
model7 += [nn.ReLU(True), ]
model7 += [nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1, bias=True), ]
model7 += [nn.ReLU(True), ]
model7 += [nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1, bias=True), ]
model7 += [nn.ReLU(True), ]
model7 += [norm_layer(512), ]
```

```
        # Model 8: Decoder
        model8 = [nn.ConvTranspose2d(512, 256, kernel_size=4, stride=2, padding=1, bias=True), ]
        model8 += [nn.ReLU(True), ]
        model8 += [nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=True), ]
        model8 += [nn.ReLU(True), ]
        model8 += [nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=True), ]
        model8 += [nn.ReLU(True), ]
        model8 += [nn.Conv2d(256, 313, kernel_size=1, stride=1, padding=0, bias=True), ]

        self.model1 = nn.Sequential(*model1)
        self.model2 = nn.Sequential(*model2)
        self.model3 = nn.Sequential(*model3)
        self.model4 = nn.Sequential(*model4)
        self.model5 = nn.Sequential(*model5)
        self.model6 = nn.Sequential(*model6)
        self.model7 = nn.Sequential(*model7)
        self.model8 = nn.Sequential(*model8)

        self.softmax = nn.Softmax(dim=1)
        self.model_out = nn.Conv2d(313, 2, kernel_size=1, padding=0, dilation=1, stride=1, bias=False)
        self.upsample4 = nn.Upsample(scale_factor=4, mode='bilinear')

    def forward(self, input_l):
        conv1_2 = self.model1(self.normalize_l(input_l))
        conv2_2 = self.model2(conv1_2)
        conv3_3 = self.model3(conv2_2)
        conv4_3 = self.model4(conv3_3)
        conv5_3 = self.model5(conv4_3)
        conv6_3 = self.model6(conv5_3)
        conv7_3 = self.model7(conv6_3)
        conv8_3 = self.model8(conv7_3)
        out_reg = self.model_out(self.softmax(conv8_3))

        return self.unnormalize_ab(self.upsample4(out_reg))


class PatchGANDiscriminator(nn.Module):
    """
    PatchGAN discriminator for 70x70 patches
    Input: L channel (1) + ab channels (2) = 3 channels
    """

    def __init__(self, in_channels=3):
        super(PatchGANDiscriminator, self).__init__()

        def discriminator_block(in_feat, out_feat, normalize=True):
            layers = [nn.Conv2d(in_feat, out_feat, 4, stride=2, padding=1)]
            if normalize:
                layers.append(nn.BatchNorm2d(out_feat))
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            return layers

        self.model = nn.Sequential(
            # No norm in first layer
```

```
            *discriminator_block(in_channels, 64, normalize=False),  # 128x128
            *discriminator_block(64, 128),  # 64x64
            *discriminator_block(128, 256),  # 32x32
            nn.Conv2d(256, 512, 4, stride=1, padding=1, bias=False),  # 31x31
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(512, 1, 4, stride=1, padding=1)  # 30x30
        )

    def forward(self, img_L, img_ab):
        # Concatenate L and ab channels
        img_input = torch.cat([img_L, img_ab], dim=1)
        return self.model(img_input)


def resize_img(img, HW=(256, 256), resample=Image.BILINEAR):
    """
    Resize numpy image to (HW[0], HW[1]).
    """
    return np.asarray(
        Image.fromarray(img).resize((HW[1], HW[0]), resample=resample)
    )


def preprocess_img(img_rgb_orig, HW=(256, 256), resample=Image.BILINEAR, return_ab=False):
    """
    Preprocess image into L_orig (original size) and L_rs (resized).
    Returns:
      tens_orig_l : (1,1,H_orig,W_orig) -> original L channel tensor
      tens_rs_l   : (1,1,HW[0],HW[1]) -> resized L channel tensor
    """
    # Resize original
    img_rgb_rs = resize_img(img_rgb_orig, HW=HW, resample=resample)

    # Convert both original & resized images to LAB
    img_lab_orig = color.rgb2lab(img_rgb_orig)
    img_lab_rs = color.rgb2lab(img_rgb_rs)

    # Extract only L channel
    img_l_orig = img_lab_orig[:, :, 0]
    img_l_rs = img_lab_rs[:, :, 0]

    # Convert to torch tensors with shape (1,1,H,W)
    tens_orig_l = torch.tensor(img_l_orig, dtype=torch.float32)[None, :, :]
    tens_rs_l = torch.tensor(img_l_rs, dtype=torch.float32)[None, :, :]

    #### Normalize L channel (paper recommends dividing by 100) -> I will look into this again(for now, lets have this) ####
    # tens_orig_l = tens_orig_l / 100.0
    # tens_rs_l   = tens_rs_l / 100.0

    # If pretrained is not True then we could center and Normalise the images
    # if pretrained == False:
    #     normalizer = BaseColor()
```

```
    #     tens_orig_l = normalizer.normalize_l(tens_orig_l)
    #     tens_rs_l   = normalizer.normalize_l(tens_rs_l)

    if return_ab:
        img_ab_rs = img_lab_rs[:, :, 1:]
        tens_ab = torch.tensor(img_ab_rs, dtype=torch.float32).permute(2, 0, 1)
        # tens_ab = normalizer.normalize_ab(tens_ab)

        # if not pretrained:
        #     tens_ab = normalizer.normalize_ab(tens_ab)

        return tens_rs_l, tens_ab

    return tens_orig_l, tens_rs_l


def postprocess_tens(tens_orig_l, out_ab, mode='bilinear'):
    """
    Combine predicted ab channels with original L channel and convert back to RGB.
    """

    HW_orig = tens_orig_l.shape[1:]  # (H_orig, W_orig)
    HW_pred = out_ab.shape[1:]  # (H_pred, W_pred)

    # If needed, resize ab to match original size
    if HW_pred != HW_orig:
        out_ab = F.interpolate(out_ab.unsqueeze(0),
                               size=HW_orig, mode='bilinear')[0]

    # if not pretrained:
    #     normalizer = BaseColor()
    #     tens_orig_l = normalizer.unnormalize_l(tens_orig_l)
    # out_ab = normalizer.unnormalize_ab(out_ab)

    # tens_orig_l: (1, H_orig, W_orig)
    # out_ab:      (2, H_orig, W_orig)

    out_lab = torch.cat((tens_orig_l, out_ab), dim=0)  # (3, H_orig, W_orig)

    out_lab_np = out_lab.cpu().numpy().transpose((1, 2, 0))
    out_rgb = color.lab2rgb(out_lab_np)

    return out_rgb


# -----------------------------------------------
# DEFINE DATASET CLASS
# -----------------------------------------------
class ColorizationDataset(Dataset):
    """
    Dataset for automatic colorization using Hugging Face dataset
    """

    def __init__(self, split='train', mode='training'):
```

```
        """
        Args:
            split: 'train' or 'validation'
            mode: 'training' or 'inference'
        """
        print(f"Loading HuggingFace dataset: split={split}, mode={mode}")
        self.hf_dataset = load_dataset("Elriggs/imagenet-50-subset", split=split)
        self.mode = mode
        print(f"Loaded {len(self.hf_dataset)} images")

    def __len__(self):
        return len(self.hf_dataset)

    def __getitem__(self, idx):
        item = self.hf_dataset[idx]

        # Get PIL image and convert to RGB
        img_pil = item['image'].convert('RGB')
        img_rgb_orig = np.array(img_pil)

        if self.mode == 'training':
            # Training mode - return L and ab
            img_rgb = np.array(img_pil.resize((IMAGE_SIZE, IMAGE_SIZE)))
            img_lab = color.rgb2lab(img_rgb)

            L = img_lab[:, :, 0]
            ab = img_lab[:, :, 1:]

            L_tensor = torch.FloatTensor(L).unsqueeze(0)
            ab_tensor = torch.FloatTensor(ab).permute(2, 0, 1)

            return L_tensor, ab_tensor
        else:
            # Inference mode - return L (resized and original) + label
            tens_orig_l, tens_rs_l = preprocess_img(img_rgb_orig, HW=(IMAGE_SIZE, IMAGE_SIZE),
                                                    return_ab=False)
            label = item['label']
            return tens_rs_l, tens_orig_l, label


def colorization_collate(batch):
    """
    Allows batching resized L (fixed size),
    while keeping original L tensors in a list.
    """
    tens_rs_l_batch = torch.stack([b[0] for b in batch], dim=0)  # batchable
    tens_orig_l_list = [b[1] for b in batch]  # NOT stacked
    img_paths = [b[2] for b in batch]  # list of paths

    return tens_rs_l_batch, tens_orig_l_list, img_paths
```

```
def gan_loss(predictions, target_is_real, device):
    """
    GAN loss with label smoothing
    """
    if target_is_real:
        # Real labels with smoothing: 0.9 instead of 1.0
        target = torch.ones_like(predictions) * 0.9
    else:
        # Fake labels: 0.0
        target = torch.zeros_like(predictions)

    loss = F.binary_cross_entropy_with_logits(predictions, target)
    return loss


def calculate_losses(generator, discriminator, feature_extractor, criterion_content, real_L, real_ab, device):
    """
    Calculate all losses for one batch
    Returns: g_loss, d_loss, l1_loss
    """
    batch_size = real_L.size(0)

    # Generate fake ab channels
    fake_ab = generator(real_L)

    # ===== DISCRIMINATOR LOSS =====
    # Real images
    d_real = discriminator(real_L, real_ab)
    d_loss_real = gan_loss(d_real, True, device)

    # Fake images (detach to not train generator)
    d_fake = discriminator(real_L, fake_ab.detach())
    d_loss_fake = gan_loss(d_fake, False, device)

    # Total discriminator loss
    d_loss = (d_loss_real + d_loss_fake) * 0.5

    # ===== GENERATOR LOSS =====
    # Adversarial loss (try to fool discriminator)
    d_fake_for_g = discriminator(real_L, fake_ab)
    g_loss_gan = gan_loss(d_fake_for_g, True, device)

    # L1 loss (pixel-wise accuracy)
    l1_loss = F.l1_loss(fake_ab, real_ab)

    # Perceptual loss
    fake_lab = torch.cat([real_L, fake_ab], dim=1)  # (B, 3, H, W)
    real_lab = torch.cat([real_L, real_ab], dim=1)
    gen_features = feature_extractor(fake_lab)
    real_features = feature_extractor(real_lab)
    loss_content = criterion_content(gen_features, real_features.detach())

    # Combined generator loss
    g_loss = g_loss_gan + LAMBDA_L1 * l1_loss + loss_content
```

```
    return g_loss, d_loss, l1_loss, g_loss_gan


def save_sample_images(generator, data_loader, epoch, save_dir="gan_samples"):
    """
    Save ONE sample image per epoch to monitor progress
    """
    os.makedirs(save_dir, exist_ok=True)

    generator.eval()
    with torch.no_grad():
        # Get one batch
        for tens_l, tens_ab in data_loader:
            tens_l = tens_l.to(DEVICE)
            tens_ab = tens_ab.to(DEVICE)

            # Generate fake colors
            fake_ab = generator(tens_l)

            # Save ONLY first image
            l_channel = tens_l[0].cpu()
            real_ab = tens_ab[0].cpu()
            pred_ab = fake_ab[0].cpu()

            # Create side-by-side comparison
            gray_rgb = postprocess_tens(l_channel, torch.zeros_like(real_ab))
            real_rgb = postprocess_tens(l_channel, real_ab)
            fake_rgb = postprocess_tens(l_channel, pred_ab)

            # Concatenate horizontally
            comparison = np.concatenate([gray_rgb, real_rgb, fake_rgb], axis=1)

            # Save
            save_path = os.path.join(save_dir, f"epoch_{epoch}.png")
            Image.fromarray((comparison * 255).astype(np.uint8)).save(save_path)

            break  # Only process one batch

    generator.train()
    print(f"Saved sample image: {save_path}")


def get_stratified_indices(hf_dataset, images_per_class=10):
    """
    Get indices for N images per class

    Args:
        hf_dataset: HuggingFace dataset object
        images_per_class: number of images to sample per class

    Returns:
        list of indices
    """
```

```
    print(f"Selecting {images_per_class} images per class...")

    # Group indices by class label
    class_indices = defaultdict(list)
    for idx in range(len(hf_dataset)):
        label = hf_dataset[idx]['label']
        class_indices[label].append(idx)

    # Sample images_per_class from each class
    selected_indices = []
    for label in sorted(class_indices.keys()):
        indices = class_indices[label]
        # Take first N images from each class
        sampled = indices[:images_per_class]
        selected_indices.extend(sampled)
        print(f"Class {label}: selected {len(sampled)} images")

    print(f"Total selected: {len(selected_indices)} images")
    return selected_indices


def run_inference(generator, split='validation', output_folder="gan_inference", num_images=10):
    """
    Run inference on num_images per class from HuggingFace dataset

    Args:
        generator: trained generator model
        split: 'train' or 'validation'
        output_folder: where to save results
        num_images: images per class (10 means 500 total for 50 classes)
    """
    print(f"INFERENCE: {num_images} images per class from {split} split")

    # Load dataset
    full_dataset = ColorizationDataset(split=split, mode='inference')

    # Get stratified indices
    selected_indices = get_stratified_indices(full_dataset.hf_dataset, images_per_class=num_images)

    # Create subset with selected indices
    inference_dataset = torch.utils.data.Subset(full_dataset, selected_indices)

    # Custom collate function for inference
    def inference_collate(batch):
        tens_rs_l = torch.stack([b[0] for b in batch], dim=0)
        tens_orig_l = [b[1] for b in batch]
        labels = [b[2] for b in batch]
        return tens_rs_l, tens_orig_l, labels

    inference_loader = DataLoader(
        inference_dataset,
        batch_size=4,
        shuffle=False,
        collate_fn=inference_collate
```

```
    )

    os.makedirs(output_folder, exist_ok=True)

    generator.eval()
    img_count = 0

    with torch.no_grad():
        for batch_idx, (tens_rs_l, tens_orig_l_list, labels) in enumerate(inference_loader):
            tens_rs_l = tens_rs_l.to(DEVICE)
            predicted_ab = generator(tens_rs_l)

            for i in range(len(labels)):
                pred_ab = predicted_ab[i].cpu()
                orig_l = tens_orig_l_list[i]
                label = labels[i]

                colorized_rgb = postprocess_tens(orig_l, pred_ab)

                # Save with class label in filename
                save_path = os.path.join(output_folder, f"class{label:03d}_img{img_count:04d}.png")
                Image.fromarray((colorized_rgb * 255).astype(np.uint8)).save(save_path)
                img_count += 1

    print(f" Saved {img_count} images to {output_folder}/")


def train_gan(generator, data_folder, pretrained_path=None, save_interval=5):
    """
    Train GAN with pretrained ECCV generator
    """
    # Load pretrained weights if provided
    # print("Loading Zhang pretrained model (original)...")
    # import torch.utils.model_zoo as model_zoo
    # generator.load_state_dict(
    #     model_zoo.load_url(
    #         'https://colorizers.s3.us-east-2.amazonaws.com/colorization_release_v2-9b330a0b.pth',
    #         map_location=DEVICE,
    #         check_hash=True
    #     )
    # )
    init_loss_log()

    if pretrained_path and os.path.exists(pretrained_path):
        print(f"Loading pretrained model from: {pretrained_path}")
        generator.load_state_dict(torch.load(pretrained_path, map_location=DEVICE))
        print(f"Loaded pretrained weights from {pretrained_path}")
    else:
        print("No pretrained weights loaded! Training from scratch.")

    # Initialize discriminator
    discriminator = PatchGANDiscriminator().to(DEVICE)
    feature_extractor = FeatureExtractor().to(DEVICE)
    feature_extractor.eval()
```

```
# Optimizers
optimizer_G = optim.Adam(generator.parameters(), lr=LR_G, betas=(0.5, 0.999))
optimizer_D = optim.Adam(discriminator.parameters(), lr=LR_D, betas=(0.5, 0.999))
criterion_content = nn.L1Loss()
# Dataset
train_dataset = ColorizationDataset(data_folder, mode='training')
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE,
                          shuffle=True, num_workers=4, pin_memory=True)

# Fixed validation sample (same image every epoch)
val_dataset = ColorizationDataset(data_folder, mode='training')
val_subset = torch.utils.data.Subset(val_dataset, [0])  # Just first image
val_loader = DataLoader(val_subset, batch_size=1, shuffle=False)

best_g_loss = float('inf')
best_epoch = 0

# Training loop
for epoch in range(EPOCHS):
    generator.train()
    discriminator.train()

    epoch_g_loss = 0
    epoch_d_loss = 0
    epoch_l1_loss = 0
    epoch_gan_loss = 0

    pbar = tqdm(train_loader, desc=f"Epoch {epoch + 1}/{EPOCHS}")

    for batch_idx, (tens_l, tens_ab) in enumerate(pbar):
        tens_l = tens_l.to(DEVICE)
        tens_ab = tens_ab.to(DEVICE)

        # Calculate losses
        g_loss, d_loss, l1_loss, g_gan_loss = calculate_losses(
            generator, discriminator, feature_extractor, criterion_content, tens_l, tens_ab, DEVICE
        )

        # ===== UPDATE DISCRIMINATOR =====
        optimizer_D.zero_grad()
        d_loss.backward()
        torch.nn.utils.clip_grad_norm_(discriminator.parameters(), 1.0)
        optimizer_D.step()

        # ===== UPDATE GENERATOR =====
        if batch_idx % D_STEPS == 0:
            optimizer_G.zero_grad()
            g_loss, _, l1_loss, g_gan_loss = calculate_losses(
                generator, discriminator, feature_extractor, criterion_content, tens_l, tens_ab, DEVICE
            )
            g_loss.backward()
            torch.nn.utils.clip_grad_norm_(generator.parameters(), 1.0)
            optimizer_G.step()
```

```
            # Track losses
            epoch_g_loss += g_loss.item()
            epoch_d_loss += d_loss.item()
            epoch_l1_loss += l1_loss.item()
            epoch_gan_loss += g_gan_loss.item()

            # Update progress bar
            pbar.set_postfix({
                'G_loss': f'{g_loss.item():.3f}',
                'D_loss': f'{d_loss.item():.3f}',
                'L1': f'{l1_loss.item():.3f}',
                'GAN': f'{g_gan_loss.item():.3f}'
            })

        n_batches = len(train_loader)
        avg_g_loss = epoch_g_loss / n_batches
        avg_d_loss = epoch_d_loss / n_batches
        avg_l1_loss = epoch_l1_loss / n_batches
        avg_gan_loss = epoch_gan_loss / n_batches

        # Epoch summary
        n_batches = len(train_loader)
        print(f"\n[Epoch {epoch + 1}] "
              f"G_loss: {epoch_g_loss / n_batches:.4f} | "
              f"D_loss: {epoch_d_loss / n_batches:.4f} | "
              f"L1_loss: {epoch_l1_loss / n_batches:.4f} | "
              f"GAN_loss: {epoch_gan_loss / n_batches:.4f}")
        log_losses(epoch + 1, avg_g_loss, avg_d_loss, avg_l1_loss, avg_gan_loss)


        # Save ONE sample image per epoch
        save_sample_images(generator, val_loader, epoch + 1)

        # ===== SAVE BEST MODEL =====
        if avg_g_loss < best_g_loss:
            best_g_loss = avg_g_loss
            best_epoch = epoch + 1
            torch.save(generator.state_dict(), 'best_gan_generator.pth')
            torch.save(discriminator.state_dict(), 'best_gan_discriminator.pth')
            print(f"✓ NEW BEST MODEL! Saved at epoch {best_epoch} with G_loss: {best_g_loss:.2f}")

        # Save checkpoints
        if (epoch + 1) % save_interval == 0:
            torch.save(generator.state_dict(),
                       f'gan_generator_epoch_{epoch + 1}.pth')
            torch.save(discriminator.state_dict(),
                       f'gan_discriminator_epoch_{epoch + 1}.pth')
            print(f"Saved checkpoint at epoch {epoch + 1}")

    return generator, discriminator


def plot_losses(csv_file='training_losses.csv', save_path='loss_plot.png'):
```

```
    """Plot and save training losses"""
    import pandas as pd

    # Read CSV
    df = pd.read_csv(csv_file)

    # Create plot
    plt.figure(figsize=(12, 8))

    plt.subplot(2, 2, 1)
    plt.plot(df['Epoch'], df['G_Loss'], 'b-', label='Generator Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Generator Loss')
    plt.grid(True)

    plt.subplot(2, 2, 2)
    plt.plot(df['Epoch'], df['D_Loss'], 'r-', label='Discriminator Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Discriminator Loss')
    plt.grid(True)

    plt.subplot(2, 2, 3)
    plt.plot(df['Epoch'], df['L1_Loss'], 'g-', label='L1 Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('L1 Loss')
    plt.grid(True)

    plt.subplot(2, 2, 4)
    plt.plot(df['Epoch'], df['GAN_Loss'], 'm-', label='GAN Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('GAN Loss')
    plt.grid(True)

    plt.tight_layout()
    plt.savefig(save_path, dpi=300, bbox_inches='tight')
    plt.close()
    print(f" Loss plot saved to {save_path}")



# ------------------------------------------------
# MAIN BLOCK
# ------------------------------------------------

if __name__ == "__main__":
    generator = ECCVGenerator()
    generator.to(DEVICE)

    print("STARTING GAN TRAINING")
    trained_gen, trained_disc = train_gan(
```

```
    generator=generator,
    data_folder="train",
    pretrained_path="colorization_model.pth",
    save_interval=5
)

# Load BEST model for inference
print("\nLoading BEST model for inference")
best_generator = ECCVGenerator()
best_generator.load_state_dict(torch.load('best_gan_generator.pth', map_location=DEVICE))
best_generator.to(DEVICE)

# Run inference with BEST model - 10 images per class
print("RUNNING INFERENCE WITH BEST FINETUNED MODEL")
run_inference(
    generator=best_generator,
    split='validation',
    output_folder="finetuned_gan_output",
    num_images=10  # <-- 10 per class = 500 total
)

print("COMPARISON: Loading Zhang's pretrained model...")

pretrained_gen = ECCVGenerator()
import torch.utils.model_zoo as model_zoo

pretrained_gen.load_state_dict(
    model_zoo.load_url(
        'https://colorizers.s3.us-east-2.amazonaws.com/colorization_release_v2-9b330a0b.pth',
        map_location=DEVICE,
        check_hash=True
    )
)
pretrained_gen.to(DEVICE)

print("\nRunning inference with PRETRAINED model on validation...")
run_inference(
    generator=pretrained_gen,
    split='validation',
    output_folder="pretrained_zhang_output",
    num_images=10
)

print("DONE! Check these folders:")
print("  - finetuned_gan_output/      (Your finetuned GAN)")
print("  - pretrained_zhang_output/   (Zhang's original)")
print("  Each has 10 images per class (500 total)")
# Plot losses from training
plot_losses('training_losses.csv', 'training_losses.png')
```

A.3 Model Evaluation Script

```
import os
import numpy as np
from PIL import Image
```

```
from tqdm import tqdm
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
import torch.utils.model_zoo as model_zoo
import pandas as pd
from skimage.metrics import structural_similarity as ssim

# Import components from the finetuned GAN training script
from dl_rough_final_gan import (
    ECCVGenerator,
    FeatureExtractor,
    ColorizationDataset,
    postprocess_tens,
    BATCH_SIZE
)

DEVICE = "cuda:0" if torch.cuda.is_available() else "cpu"
MAX_DISTANCE = 150.0
N_BINS = 151


def evaluate_single_model(
    generator: nn.Module,
    eval_loader: DataLoader,
    feature_extractor: nn.Module,
):
    """
    Run evaluation for a *single* generator instance.
    Returns a dict of scalar metrics.
    """

    l1_loss_fn = nn.L1Loss()
    mse_loss_fn = nn.MSELoss()
    perceptual_loss_fn = nn.L1Loss()

    total_l1 = 0.0
    total_mse = 0.0
    total_perc = 0.0
    n_batches = 0

    total_psnr = 0.0
    total_psnr_count = 0

    total_ssim = 0.0
    total_ssim_count = 0

    error_hist = torch.zeros(N_BINS, dtype=torch.float64)
    total_pixels = 0

    generator.eval()
    feature_extractor.eval()

    with torch.no_grad():
```

```
for batch_idx, (tens_l, tens_ab_gt) in enumerate(tqdm(eval_loader)):
    tens_l = tens_l.to(DEVICE)
    tens_ab_gt = tens_ab_gt.to(DEVICE)

    # Forward pass
    pred_ab = generator(tens_l)

    # ------------------------------------------------
    # Losses in ab space
    # ------------------------------------------------
    l1_loss_val = l1_loss_fn(pred_ab, tens_ab_gt).item()
    mse_loss_val = mse_loss_fn(pred_ab, tens_ab_gt).item()

    fake_lab = torch.cat([tens_l, pred_ab], dim=1)
    real_lab = torch.cat([tens_l, tens_ab_gt], dim=1)
    perc_loss_val = perceptual_loss_fn(
        feature_extractor(fake_lab),
        feature_extractor(real_lab)
    ).item()

    total_l1 += l1_loss_val
    total_mse += mse_loss_val
    total_perc += perc_loss_val

    # ------------------------------------------------
    # PSNR & SSIM in RGB space (values in [0, 1])
    # ------------------------------------------------
    for i in range(tens_l.size(0)):
        gray_cpu = tens_l[i].cpu()
        real_ab_cpu = tens_ab_gt[i].cpu()
        fake_ab_cpu = pred_ab[i].cpu()

        real_rgb = postprocess_tens(gray_cpu, real_ab_cpu)
        fake_rgb = postprocess_tens(gray_cpu, fake_ab_cpu)

        # PSNR with MAX=1.0
        mse_rgb = np.mean((fake_rgb - real_rgb) ** 2)
        if mse_rgb > 0:
            psnr_val = 10.0 * np.log10(1.0 / mse_rgb)
        else:
            psnr_val = float("inf")

        total_psnr += psnr_val
        total_psnr_count += 1

        # SSIM (skimage expects HWC)
        ssim_val = ssim(
            real_rgb,
            fake_rgb,
            data_range=1.0,
            channel_axis=-1,
        )
        total_ssim += ssim_val
        total_ssim_count += 1
```

```
            n_batches += 1

            # ----------------------------------------------------
            # AUC histogram in ab space
            # ----------------------------------------------------
            diff = pred_ab - tens_ab_gt             # (B,2,H,W)
            err = torch.sqrt(torch.sum(diff ** 2, dim=1))  # (B,H,W)
            err_flat = err.reshape(-1).cpu()
            err_flat = torch.clamp(err_flat, 0.0, MAX_DISTANCE)

            batch_hist = torch.histc(
                err_flat,
                bins=N_BINS,
                min=0.0,
                max=MAX_DISTANCE,
            )

            error_hist += batch_hist
            total_pixels += err_flat.numel()

    # ------------------------------------------------------------
    # Aggregate metrics
    # ------------------------------------------------------------
    avg_l1 = total_l1 / n_batches
    avg_mse = total_mse / n_batches
    avg_perc = total_perc / n_batches

    avg_psnr = total_psnr / total_psnr_count if total_psnr_count > 0 else float("nan")
    avg_ssim = total_ssim / total_ssim_count if total_ssim_count > 0 else float("nan")

    if total_pixels > 0:
        pdf = error_hist / float(total_pixels)
        cmf = torch.cumsum(pdf, dim=0)

        thresholds = torch.linspace(0.0, MAX_DISTANCE, steps=N_BINS)
        auc_raw = torch.trapz(cmf, thresholds).item()
        auc_norm = auc_raw / MAX_DISTANCE
    else:
        auc_norm = float("nan")

    return {
        "avg_l1": avg_l1,
        "avg_mse": avg_mse,
        "avg_perceptual": avg_perc,
        "auc_raw_norm": auc_norm,
        "avg_psnr": avg_psnr,
        "avg_ssim": avg_ssim,
        "num_batches": n_batches,
    }


def evaluate_both_models(
    gan_weights_path="best_gan_generator.pth",
```

```
    split="validation",
    batch_size=None,
    max_batches=None,
):
    """
    Evaluate BOTH:
      1) Fine-tuned GAN generator
      2) Pretrained ECCV-16 generator

    on the same Hugging Face split (e.g., 'train' or 'validation').
    """

    if batch_size is None:
        batch_size = BATCH_SIZE

    # ----------------------------------------------------------
    # Dataset / dataloader (Hugging Face via ColorizationDataset)
    eval_dataset = ColorizationDataset(split=split, mode="training")

    if max_batches is not None:
        max_samples = max_batches * batch_size
        eval_dataset = torch.utils.data.Subset(
            eval_dataset,
            range(min(len(eval_dataset), max_samples))
        )

    eval_loader = DataLoader(
        eval_dataset,
        batch_size=batch_size,
        shuffle=False,
        num_workers=4,
        pin_memory=True,
    )

    print(f"Dataset size for evaluation (split='{split}'): {len(eval_dataset)}")

    # ----------------------------------------------------------
    # Shared feature extractor
    # ----------------------------------------------------------
    feature_extractor = FeatureExtractor().to(DEVICE)

    results = {}

    # ----------------------------------------------------------
    # Fine-tuned GAN generator
    # ----------------------------------------------------------
    print(f"\n=== Evaluating Fine-tuned GAN generator ({gan_weights_path}) ===")
    gan_gen = ECCVGenerator().to(DEVICE)
    gan_gen.load_state_dict(torch.load(gan_weights_path, map_location=DEVICE))

    gan_results = evaluate_single_model(gan_gen, eval_loader, feature_extractor)
    results["finetuned_gan"] = gan_results

    print("===== FINETUNED GAN RESULTS =====")
```

```
    print(f"Avg L1 Loss:         {gan_results['avg_l1']:.4f}")
    print(f"Avg MSE Loss:        {gan_results['avg_mse']:.4f}")
    print(f"Avg Perceptual Loss: {gan_results['avg_perceptual']:.4f}")
    print(f"Raw AUC (0..{int(MAX_DISTANCE)}): {gan_results['auc_raw_norm']:.4f}")
    print(f"Avg PSNR:            {gan_results['avg_psnr']:.4f}")
    print(f"Avg SSIM:            {gan_results['avg_ssim']:.4f}")
    print(f"Total batches:       {gan_results['num_batches']}")
    print("====================================\n")

    # ----------------------------------------------------
    # Pretrained ECCV-16 generator (Zhang et al.)
    # ----------------------------------------------------
    print("=== Evaluating Pretrained ECCV-16 generator (Zhang et al., 2016) ===")
    pre_gen = ECCVGenerator().to(DEVICE)
    pre_gen.load_state_dict(
        model_zoo.load_url(
            "https://colorizers.s3.us-east-2.amazonaws.com/colorization_release_v2-9b330a0b.pth",
            map_location="cpu",
            check_hash=True,
        )
    )
    pre_gen.to(DEVICE)

    pre_results = evaluate_single_model(pre_gen, eval_loader, feature_extractor)
    results["pretrained_eccv"] = pre_results

    print("===== PRETRAINED ECCV-16 RESULTS =====")
    print(f"Avg L1 Loss:         {pre_results['avg_l1']:.4f}")
    print(f"Avg MSE Loss:        {pre_results['avg_mse']:.4f}")
    print(f"Avg Perceptual Loss: {pre_results['avg_perceptual']:.4f}")
    print(f"Raw AUC (0..{int(MAX_DISTANCE)}): {pre_results['auc_raw_norm']:.4f}")
    print(f"Avg PSNR:            {pre_results['avg_psnr']:.4f}")
    print(f"Avg SSIM:            {pre_results['avg_ssim']:.4f}")
    print(f"Total batches:       {pre_results['num_batches']}")
    print("========================================\n")

    return results


if __name__ == "__main__":
    results = evaluate_both_models(
        gan_weights_path="best_gan_generator.pth",
        split="validation",
        batch_size=16,
        max_batches=None,
    )
```

A.4 Streatlit App Demo

```
import streamlit as st
import torch
```

```
import numpy as np
from PIL import Image
from dl_rough_final_gan import ECCVGenerator, preprocess_img, postprocess_tens
import torch.utils.model_zoo as model_zoo

DEVICE = "cuda:0" if torch.cuda.is_available() else "cpu"
PRETRAINED_URL = "https://colorizers.s3.us-east-2.amazonaws.com/colorization_release_v2-9b330a0b.pth"

@st.cache_resource
def load_models():
    # pretrained model
    pretrained_gen = ECCVGenerator().to(DEVICE)
    state_pre = model_zoo.load_url(
        PRETRAINED_URL,
        map_location=DEVICE,
        check_hash=True
    )
    pretrained_gen.load_state_dict(state_pre)
    pretrained_gen.eval()

    # fine-tuned GAN generator
    gan_gen = ECCVGenerator().to(DEVICE)
    state_gan = torch.load("best_gan_generator.pth", map_location=DEVICE)
    gan_gen.load_state_dict(state_gan)
    gan_gen.eval()

    return pretrained_gen, gan_gen

pretrained_generator, gan_generator = load_models()

st.set_page_config(layout="wide")
st.title("Image Colorizer")

uploaded_file = st.file_uploader("Upload an image", type=["jpg", "jpeg", "png"])

if uploaded_file is not None:
    # Load original image
    img = Image.open(uploaded_file).convert("RGB")
    img_np = np.array(img)

    tens_orig_l, tens_rs_l = preprocess_img(img_np, HW=(256, 256), return_ab=False)
    tens_rs_l = tens_rs_l.unsqueeze(0).to(DEVICE)

    # inference
    with torch.no_grad():
        # pretrained model prediction
        pred_ab_pre = pretrained_generator(tens_rs_l)[0].cpu()

        # GAN fine-tuned model prediction
        pred_ab_gan = gan_generator(tens_rs_l)[0].cpu()

    # postprocess
    colorized_pre = postprocess_tens(tens_orig_l, pred_ab_pre)
    colorized_pre_img = (colorized_pre * 255).astype(np.uint8)
```

```
colorized_gan = postprocess_tens(tens_orig_l, pred_ab_gan)
colorized_gan_img = (colorized_gan * 255).astype(np.uint8)

# display side by side
col1, col2, col3 = st.columns(3)

with col1:
    st.subheader("Ground Truth")
    st.image(img_np)

with col2:
    st.subheader("Pretrained")
    st.image(colorized_pre_img)

with col3:
    st.subheader("Fine-tuned GAN")
    st.image(colorized_gan_img)
```