How Will You Prepare Training Data?

I will use the CIFAR-10 dataset with 60,000 RGB images (32x32) across 10 classes, ideal for colorization due to its diversity. I will resize images to 64x64 using bilinear interpolation, convert them to grayscale for input, and keep the original RGB as ground truth. Pixel values will be normalized to [0,1] with ToTensor(), matching the model's Sigmoid output. To increase data diversity and prevent overfitting, I'll apply random horizontal flips and rotations, avoiding color-based augmentations to preserve ground truth consistency. The data will be split into 80% training (48,000), 10% validation (6,000), and 10% testing (6,000). I'll use PyTorch's DataLoader with batch size 16, shuffle=True for training, and num_workers=2 for efficient loading.

What Would Be the Size of the Training Data?

The CIFAR-10 dataset has 60,000 images, with 80% (48,000) used for training. Each image is resized to 64x64 pixels. Inputs are grayscale (1x64x64 = 4,096 pixels), and ground truth images are RGB (3x64x64 = 12,288 pixels). Storage for grayscale inputs is about 0.79 GB, and RGB ground truth requires around 2.36 GB, totaling approximately 3.15 GB—manageable for local storage and batch loading. For a batch size of 16, inputs use about 0.26 MB and outputs about 0.79 MB, totaling ~1 MB per batch, which fits well within an 8 GB GPU alongside the model (~170K parameters, ~3 GB).

How Will You Guarantee the Richness of the Data?

I will leverage CIFAR-10's diversity with 10 classes offering varied colors, textures, and objects for colorization. To enrich the data, I'll apply augmentations while avoiding color-based ones to keep ground truth consistent. For better generalization, I'll use regularization techniques like batch normalization, dropout (p=0.3), and weight decay (0.0001). Setting shuffle=True in the DataLoader ensures random batch sampling each epoch, exposing the model to varied examples for robust learning.

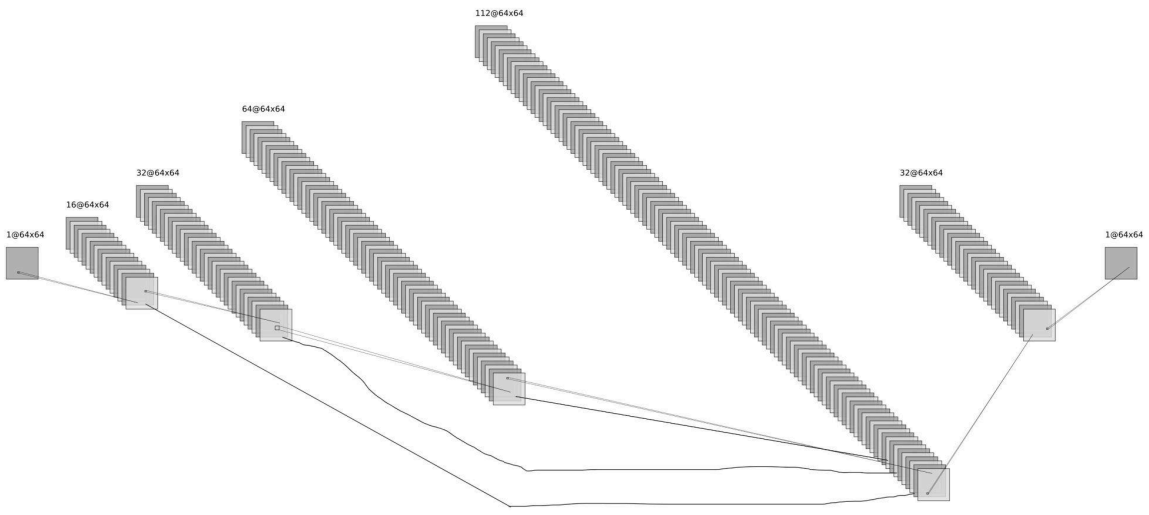
How Will You Collect the Data?

CIFAR-10 is publicly available dataset, accessible via PyTorch's torchvision.datasets

How Will You Obtain Ground Truth?

I will use the original RGB images from the CIFAR-10 dataset as the ground truth. These images have dimensions 3x32x32 and will be resized to 3x64x64 to match the model's output size.

Show Your Architecture



Explain Your Architecture

Block 1: I extract low-level features such as edges using two Conv2D layers with 3x3 kernels, increasing filters from 1 to 16. This is followed by Batch Normalization and ReLU activation, while maintaining the spatial resolution at 64x64. The output of this block is saved for use in skip connections.

Block 2: I capture mid-level features like textures with two Conv2D layers (3x3 kernels) that increase filters from 16 to 32. BatchNorm, ReLU, and dropout with p=0.3 are applied for regularization. The spatial resolution remains 64x64. The output is also saved for skip connections.

Block 3: I process high-level features such as object parts using two Conv2D layers (3x3 kernels) increasing filters from 32 to 64, followed by BatchNorm, ReLU, and dropout. The resolution stays at 64x64.

Output Block: I concatenate the outputs from Block 1 (16 channels), Block 2 (32 channels), and Block 3 (64 channels) to form a 112-channel feature map. Then, a 1x1 Conv2D layer reduces the channels from 112 to 32, followed by BatchNorm and ReLU. Finally, a 3x3 Conv2D layer maps 32 channels to 3 RGB channels, and a Sigmoid activation scales the output to [0,1].

Loss: I use Mean Squared Error (MSE) to minimize pixel-wise differences between the predicted and ground truth RGB images.

Skip connections concatenate multi-scale features from low to high levels, enriching the color prediction. Batch Normalization stabilizes training. Dropout helps prevent overfitting. ReLU and Sigmoid activations provide non-linearity and ensure bounded output values.

Why the Architecture is Robust

I ensure the robustness of my architecture through several key strategies. To improve generalization, I incorporate Batch Normalization and Dropout (30%) to reduce overfitting, along with Adam optimizer's weight decay (0.0001) for effective weight regularization. I apply data augmentation techniques such as random flips and rotations on the diverse CIFAR-10 dataset, which contains 10 distinct classes, to enhance the model's ability to handle varied inputs. For stable training, Batch Normalization helps stabilize gradients, while Adam's adaptive learning rate facilitates efficient convergence. I also implement early stopping with a patience of 5 epochs to prevent overfitting by halting training when validation loss plateaus. My architecture leverages skip connections that concatenate multi-scale features enabling rich feature representation. Hierarchical blocks progressively deepen feature channels from 16 to 64, allowing the model to capture complex patterns effectively. The model is lightweight, with approximately 170K parameters and a memory footprint around 3.5 GB, making it suitable for training on an 8 GB GPU. I expect the model to achieve a Peak Signal-to-Noise Ratio (PSNR) of approximately 20–25 dB, delivering accurate colorization with well-preserved details thanks to the combination of skip connections and regularization techniques.

Explain the Loss Function You Will Use

I choose MSE because it directly measures pixel-wise differences, which is ideal for image-to-image regression tasks like colorization. It penalizes larger errors more strongly by squaring them, encouraging the model to make accurate color predictions. Also, MSE is simple and works well with backpropagation, making it a natural fit for my training process.

Which Performance Metrics Will You Use and Why

I will use Mean Squared Error (MSE) for training and validation, and Peak Signal-to-Noise Ratio (PSNR) for testing (and optionally validation). I use PSNR because it is a standard metric for image quality in colorization tasks. It provides a logarithmic scale in decibels (dB), which is more interpretable than raw MSE values. A higher PSNR indicates better image quality, making it crucial for evaluating the model's performance on the test set.

Give Details on Training Parameters and Choices

I will use Adam because it adapts learning rates for each parameter, making it robust and effective for training CNNs. I will start with 0.001, a standard and reliable choice for Adam that balances convergence speed and stability, especially for image tasks. I will set weight decay to 0.0001 to regularize the model and help prevent overfitting. I will use a batch size of 16, which balances GPU memory usage (about 1 MB per batch) and gradient stability. This size fits well within an 8 GB GPU alongside the model (~170K parameters, ~3.5 GB total memory). I plan to train for 30 epochs with early stopping. This provides enough iterations for learning while preventing overfitting. I will use a patience of 5 epochs, meaning training will stop if the validation loss does not improve for 5 consecutive epochs. This helps ensure good generalization and saves computational resources.

Estimate the Training Time with the Training and Validation Dataset

- Training: 48,000 images, 3,000 batches (48,000 / 16). Validation: 6,000 images, 375 batches (6,000 / 16).
- Parameters: ~170K. Image size: 3x64x64 (~12,288 pixels).
- Architecture: Feedforward CNN with 3 blocks, concatenation, ~10 Conv2D layers (including 1x1), BatchNorm, Dropout.
- FLOPs per image:
    - Forward pass: ~1-2 GFLOPs (based on similar CNNs for 64x64 images).
    - Backpropagation: ~2x forward pass FLOPs
    - Total per image: ~3-6 GFLOPs (forward + backward).
- Batch FLOPs: 16 images * 3-6 GFLOPs ≈ 48-96 GFLOPs.
- GPU: 8 GB Nvidia (e.g., RTX 3060, ~10 TFLOPS FP32). Efficiency: ~50% (due to I/O, PyTorch overhead, synchronization).
- Training batch: 48-96 GFLOPs / (10 TFLOPS * 0.5) ≈ 9.6-19.2 ms.
    - Includes forward pass, loss computation, backpropagation (gradients for ~170K parameters), and optimizer step
- Validation batch: ~4.8-9.6 ms (forward pass only, no backprop).
- Per epoch:
    - Training: 3,000 batches * 19.2 ms ≈ 57.6 seconds. Validation: 375 batches * 9.6 ms ≈ 3.6 seconds.
    - Total: ~61.2 seconds per epoch.
- Assume early stopping at ~15-20 epochs Total: 61.2 seconds/epoch * 20 epochs ≈ 1,224 seconds ≈ **20 minutes**.

What Will Be the Allocated GPU Memory Once Model Is Trained (for a Single Inference)

- Memory for parameters: 170,000×4 bytes=680,000 bytes≈0.68 MB
- BatchNorm Parameters:2×10 layers×(16+32+64+112+32) channels×4 bytes≈0.01 MB
- Total model size (parameters + BatchNorm): ≈0.7 MB
- Input:1×64×64×4 bytes = 16 KB
- Block 1 (16 channels): 16×64×64×4=0.26 MB
- Block 2 (32 channels): 32×64×64×4=0.52 MB
- Block 3 (64 channels): 64×64×64×4=1.05 MB
- Concatenation (112 channels): 112×64×64×4=1.83 MB
- Output Block (32 channels): 32×64×64×4=0.52 MB
- Output RGB (3 channels): 3×64×64×4=0.05 MB
- **Total Activations:** 0.26+0.52+1.05+1.83+0.52+0.05≈4.2 MB
- **Total Estimated GPU Memory Usage:** 0.7 MB (model)+4.2 MB (activations)+2 MB (intermediate)+200 MB (overhead)≈207 MB

Estimated Inference Time for a 512x512 Grayscale Image

(512/64)^2 = 8^2 = 64 times more pixels than 64x64.

- 512x512: 64 * 1-2 GFLOPs ≈ 64-128 GFLOPs.
- 64-128 GFLOPs / (10 TFLOPS * 0.5) = 12.8-25.6 ms.

Potential Limitations: I acknowledge that my model's reliance on low-resolution CIFAR-10 data, shallow architecture, and potential ambiguity in predicting colors from grayscale inputs may limit its ability to generalize to high-resolution images and complex scenes.