LOW LIGHT IMAGE ENHANCEMENT USING U-NET ARCHITECTURE

BY: ARPIT KUMAR PANDEY

ABSTRACT

Development and evaluation of a U-Net-based deep learning model for enhancing low-light images. Utilizing a custom dataset, the approach involves pre-processing through gamma correction, followed by training multiple U-Net models. The training process is optimized using a combination of techniques such as mixed precision training and gradient accumulation, with a particular focus on maintaining model stability through gradient clipping. The evaluation metric primarily used is the Peak Signal-to-Noise Ratio (PSNR), which measures the quality of the enhanced images against ground truth high-light images.

INTRODUCTION

This project focuses on implementing and evaluating a U-Net architecture for low-light image enhancement. U-Net, originally developed for biomedical image segmentation, has proven effective in various image-to-image tasks, including enhancing low-light images. The model aims to improve the visibility and quality of images captured under low-light conditions by enhancing their brightness and contrast while minimizing noise. The performance of the network is evaluated using the Peak Signal-to-Noise Ratio (PSNR) metric, a standard measure for assessing the quality of image reconstruction.

ARCHITECTURE SPECIFICATIONS:

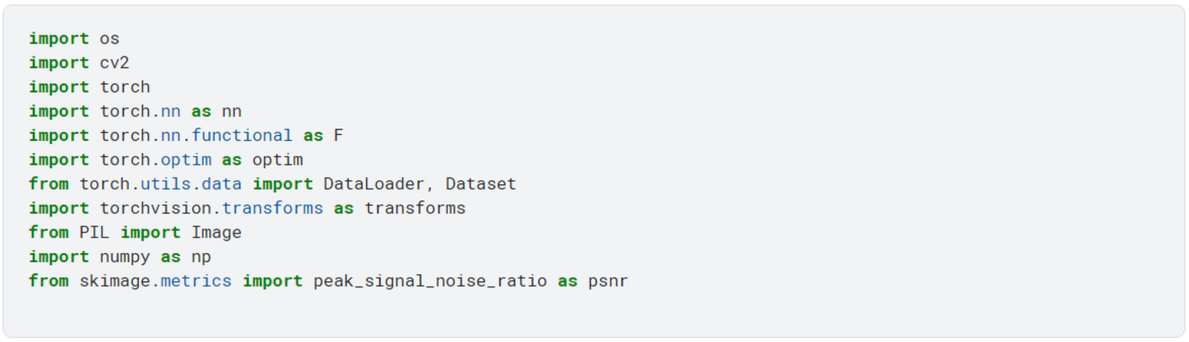
- **Model:** U-Net for image enhancement
- **Learning Rate:** 0.0001
- **Batch Size:** 4
- **Accumulation Steps:** 1
- **Epochs:** 35

The PSNR value achieved by the model across 35 epochs is approximately **21.6182**. The project is based on the methodology described in the paper "Low-Light Image Enhancement Using U-Net Architecture" (link to the paper: [https://arxiv.org/pdf/1805.01934]).

PROJECT DETAILS

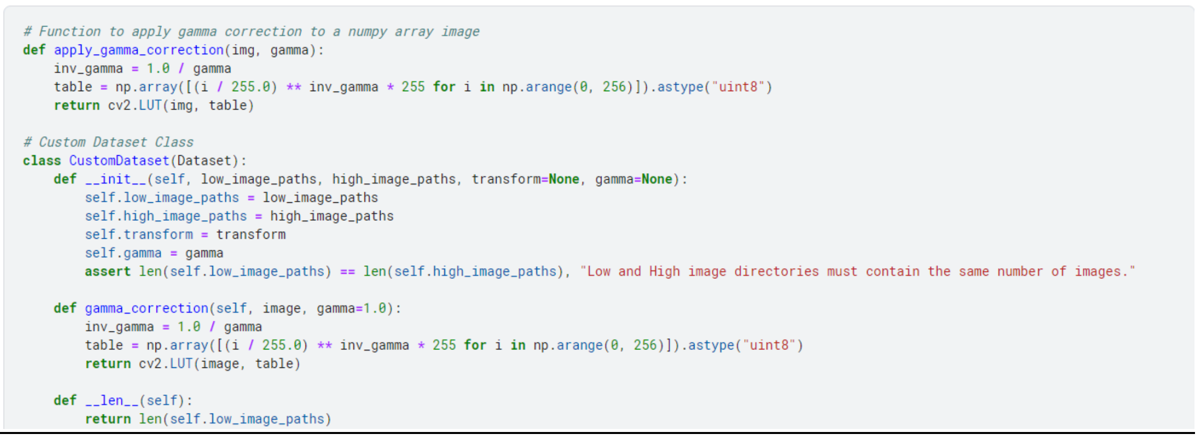
The project is implemented in Python using popular deep learning frameworks such as PyTorch. Below are the components of the code, along with explanations of their purpose and functionality.

1. IMPORTING LIBRARIES

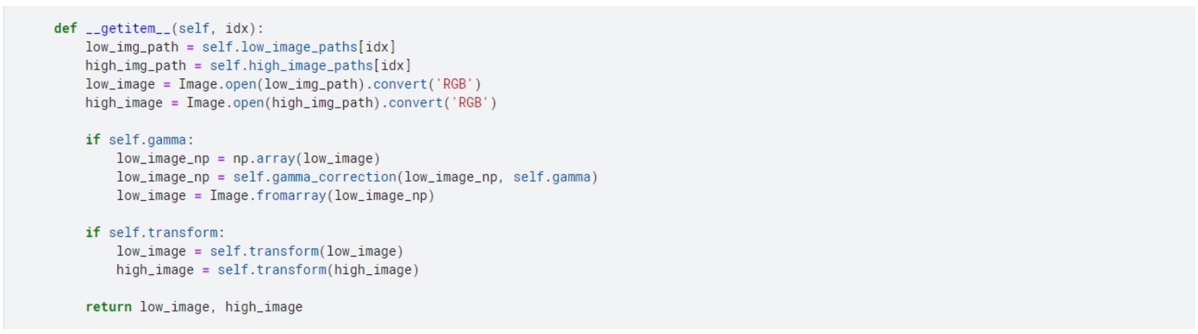


```python
import os
import cv2
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
import torchvision.transforms as transforms
from PIL import Image
import numpy as np
from skimage.metrics import peak_signal_noise_ratio as psnr
```

2. IMAGE PRE-PROCESSING



```python
# Function to apply gamma correction to a numpy array image
def apply_gamma_correction(img, gamma):
    inv_gamma = 1.0 / gamma
    table = np.array([(i / 255.0) ** inv_gamma * 255 for i in np.arange(0, 256)]).astype("uint8")
    return cv2.LUT(img, table)

# Custom Dataset Class
class CustomDataset(Dataset):
    def __init__(self, low_image_paths, high_image_paths, transform=None, gamma=None):
        self.low_image_paths = low_image_paths
        self.high_image_paths = high_image_paths
        self.transform = transform
        self.gamma = gamma
        assert len(self.low_image_paths) == len(self.high_image_paths), "Low and High image directories must contain the same number of images."

    def gamma_correction(self, image, gamma=1.0):
        inv_gamma = 1.0 / gamma
        table = np.array([(i / 255.0) ** inv_gamma * 255 for i in np.arange(0, 256)]).astype("uint8")
        return cv2.LUT(image, table)

    def __len__(self):
        return len(self.low_image_paths)
```



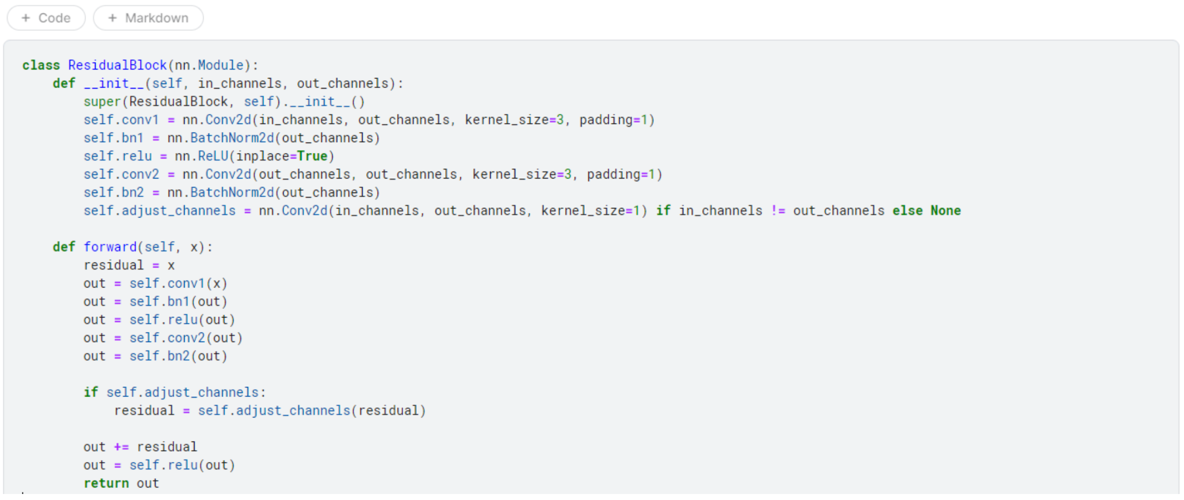
```python
    def __getitem__(self, idx):
        low_img_path = self.low_image_paths[idx]
        high_img_path = self.high_image_paths[idx]
        low_image = Image.open(low_img_path).convert('RGB')
        high_image = Image.open(high_img_path).convert('RGB')

        if self.gamma:
            low_image_np = np.array(low_image)
            low_image_np = self.gamma_correction(low_image_np, self.gamma)
            low_image = Image.fromarray(low_image_np)

        if self.transform:
            low_image = self.transform(low_image)
            high_image = self.transform(high_image)

        return low_image, high_image
```

- The class custom dataset takes the paths of low and high light images and applies gamma correction on low light images to increase their brightness.

- Applying gamma correction ensures that all images have consistent brightness levels, which can help the model learn better representations.
- **Image Conversion**: Converts images to the RGB format to ensure compatibility with the model.
- **Transformation Application**: Applies any specified transformations (e.g., resizing, normalization) to both low-light and high-light images, preparing them for training.

- **Get Item Method**: __getitem__(self, idx): Loads images at the given index, applies gamma correction if specified, and then applies any given transformations.

3. ***RESIDUAL-BLOCK CLASS***



```
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.adjust_channels = nn.Conv2d(in_channels, out_channels, kernel_size=1) if in_channels != out_channels else None

    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)

        if self.adjust_channels:
            residual = self.adjust_channels(residual)

        out += residual
        out = self.relu(out)
        return out
```

• The ResidualBlock class defines a neural network block with two convolutional layers, batch normalization, ReLU activation, and an identity skip connection for residual learning.

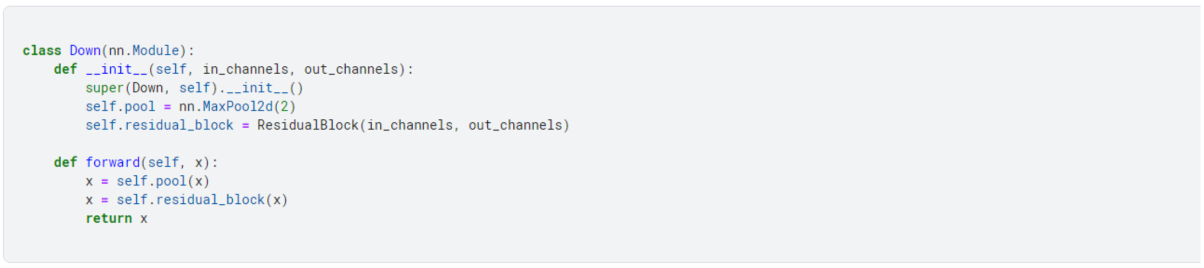
• **Enhanced Gradient Flow**: Residual connections help mitigate vanishing gradient issues, allowing for deeper and more effective U-Net architectures.

• **Improved Feature Learning**: The use of convolutional layers with residuals enables the model to learn more complex features, enhancing its capability in tasks such as image segmentation.

• **Stability and Speed**: Batch normalization improves training stability and convergence speed, which is beneficial for the overall performance of the U-Net model.

4.

DOWNSAMPLING

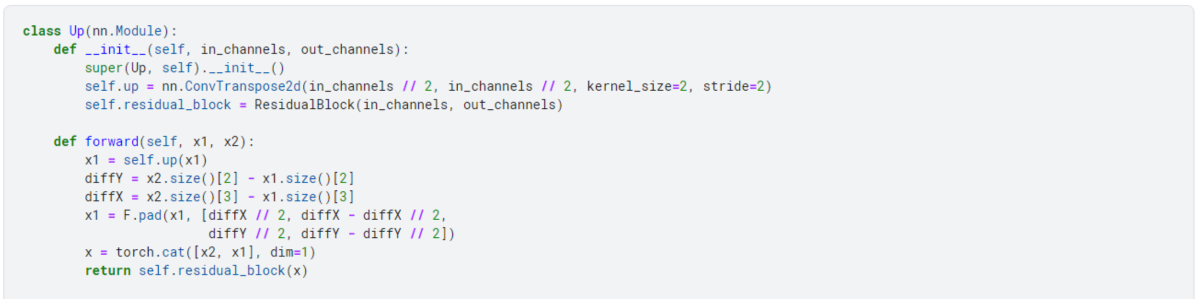


```
class Down(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(Down, self).__init__()
        self.pool = nn.MaxPool2d(2)
        self.residual_block = ResidualBlock(in_channels, out_channels)

    def forward(self, x):
        x = self.pool(x)
        x = self.residual_block(x)
        return x
```

The Down class defines a downsampling module with a max pooling layer followed by a residual block, reducing spatial dimensions while increasing feature depth. This structure allows the U-Net model to capture hierarchical features at multiple scales efficiently.

5.

UPSAMPLING

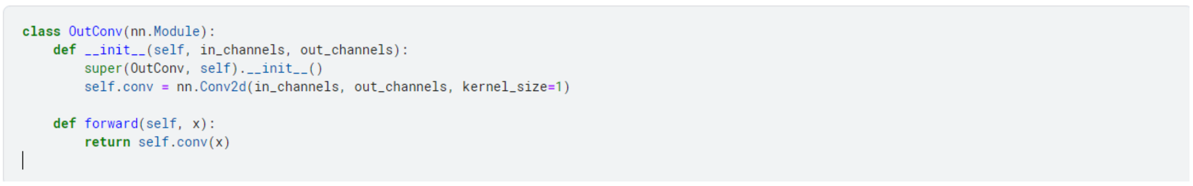


```python
class Up(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(Up, self).__init__()
        self.up = nn.ConvTranspose2d(in_channels // 2, in_channels // 2, kernel_size=2, stride=2)
        self.residual_block = ResidualBlock(in_channels, out_channels)

    def forward(self, x1, x2):
        x1 = self.up(x1)
        diffY = x2.size()[2] - x1.size()[2]
        diffX = x2.size()[3] - x1.size()[3]
        x1 = F.pad(x1, [diffX // 2, diffX - diffX // 2,
                        diffY // 2, diffY - diffY // 2])
        x = torch.cat([x2, x1], dim=1)
        return self.residual_block(x)
```

The Up class defines an upsampling module using a transposed convolution to increase spatial dimensions, followed by a residual block for feature refinement. It aligns and concatenates the upsampled feature map with the corresponding feature map from the encoder path, ensuring detailed feature recovery. This setup helps maintain spatial information while reconstructing the image at higher resolutions.

6.

OUTERMOST LAYER

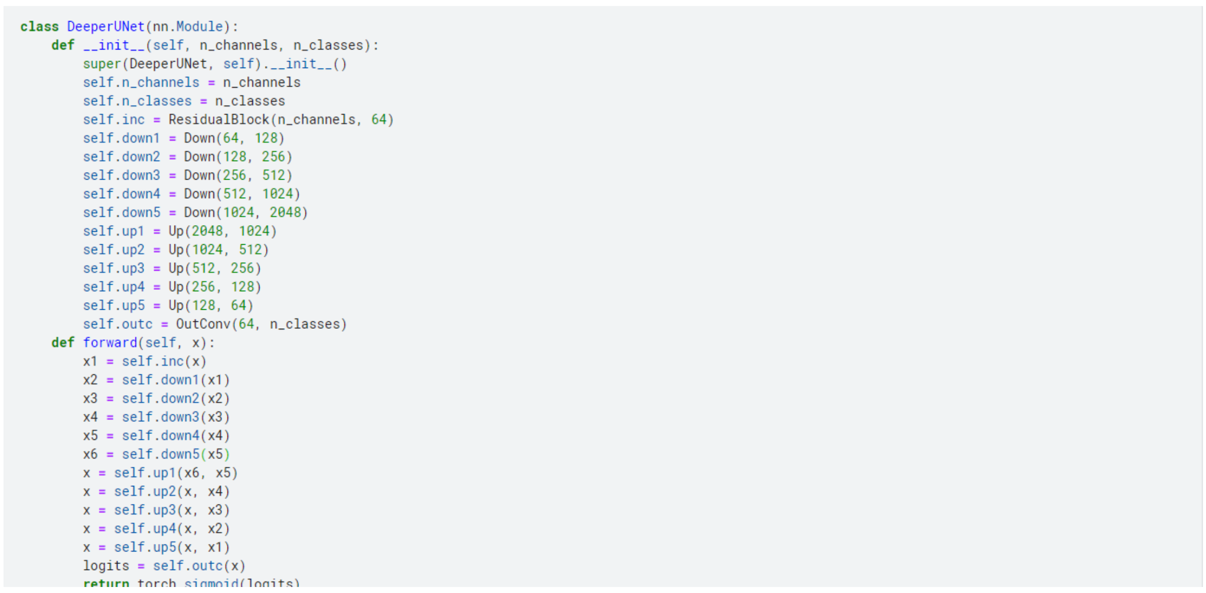


```python
class OutConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(OutConv, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=1)

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

The `Out-Conv` class defines a single convolutional layer with a kernel size of 1, which serves to map the final feature maps to the desired number of output channels. This is typically used to produce the final output of the network, such as a multi-channel image.

7.

U-NET CLASS



```python
class DeeperUNet(nn.Module):
    def __init__(self, n_channels, n_classes):
        super(DeeperUNet, self).__init__()
        self.n_channels = n_channels
        self.n_classes = n_classes
        self.inc = ResidualBlock(n_channels, 64)
        self.down1 = Down(64, 128)
        self.down2 = Down(128, 256)
        self.down3 = Down(256, 512)
        self.down4 = Down(512, 1024)
        self.down5 = Down(1024, 2048)
        self.up1 = Up(2048, 1024)
        self.up2 = Up(1024, 512)
        self.up3 = Up(512, 256)
        self.up4 = Up(256, 128)
        self.up5 = Up(128, 64)
        self.outc = OutConv(64, n_classes)
    def forward(self, x):
        x1 = self.inc(x)
        x2 = self.down1(x1)
        x3 = self.down2(x2)
        x4 = self.down3(x3)
        x5 = self.down4(x4)
        x6 = self.down5(x5)
        x = self.up1(x6, x5)
        x = self.up2(x, x4)
        x = self.up3(x, x3)
        x = self.up4(x, x2)
        x = self.up5(x, x1)
        logits = self.outc(x)
        return torch.sigmoid(logits)
```

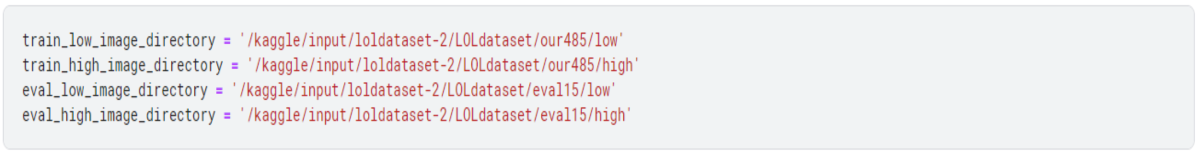
The DeeperUNet class defines a U-Net model with increased depth and more channels in the deeper layers, enhancing its capacity to capture and process complex features. It consists of an initial residual block (inc), five downsampling paths (down1 to down5), followed by five upsampling paths (up1 to up5), and a final convolutional layer (outc) to produce the output. This deeper architecture aims to improve the model's performance on tasks requiring detailed feature extraction and reconstruction.

OVERALL FLOW OF THE MODEL ARCHITECTURE:

- **ResidualBlock Class**: Defines a residual block consisting of two convolutional layers with batch normalization and ReLU activation. It includes a skip connection to preserve information across layers.
- **Down Class**: Implements downsampling using max pooling followed by a residual block. This reduces spatial dimensions while increasing feature depth.
- **Up Class**: Performs upsampling using transposed convolution (deconvolution) to increase spatial dimensions. It also concatenates feature maps from a skip connection with those from the downsampling path before applying a residual block.
- **OutConv Class**: Concludes each path with a 1x1 convolution to map feature channels to the desired output classes (e.g., RGB channels).
- **DeeperUNet Class**: Combines the above components to form a deeper U-Net architecture for super-resolution. It sequentially applies downsampling (Down modules) to capture hierarchical features and upsampling (Up modules) to recover high-resolution details. The final output is obtained through the OutConv module.

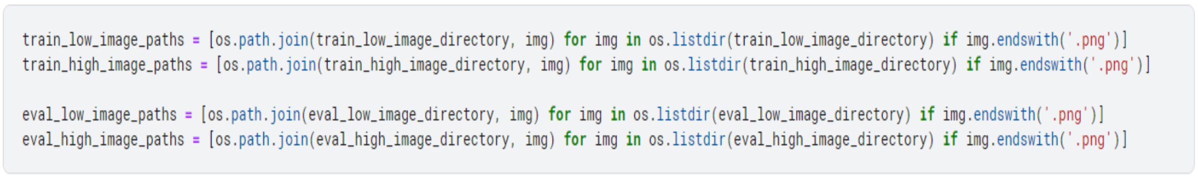
8.

DATASET PATHS



```
train_low_image_directory = '/kaggle/input/loldataset-2/LOLdataset/our485/low'
train_high_image_directory = '/kaggle/input/loldataset-2/LOLdataset/our485/high'
eval_low_image_directory = '/kaggle/input/loldataset-2/LOLdataset/eval15/low'
eval_high_image_directory = '/kaggle/input/loldataset-2/LOLdataset/eval15/high'
```

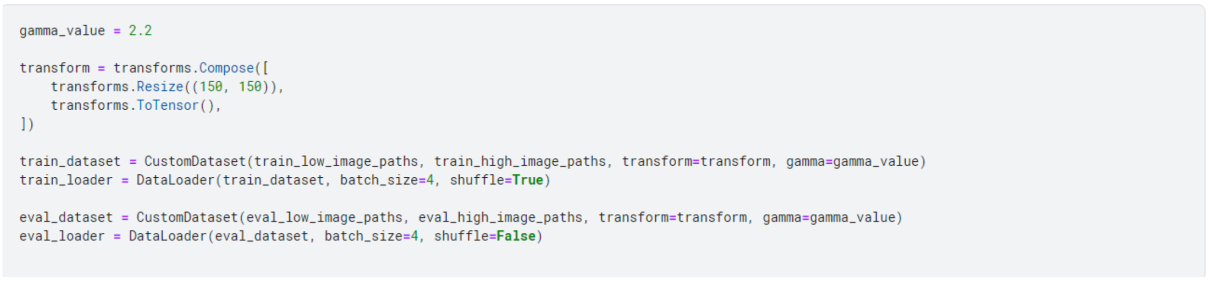
IMAGE PATHS LISTING



```
train_low_image_paths = [os.path.join(train_low_image_directory, img) for img in os.listdir(train_low_image_directory) if img.endswith('.png')]
train_high_image_paths = [os.path.join(train_high_image_directory, img) for img in os.listdir(train_high_image_directory) if img.endswith('.png')]

eval_low_image_paths = [os.path.join(eval_low_image_directory, img) for img in os.listdir(eval_low_image_directory) if img.endswith('.png')]
eval_high_image_paths = [os.path.join(eval_high_image_directory, img) for img in os.listdir(eval_high_image_directory) if img.endswith('.png')]
```

9.

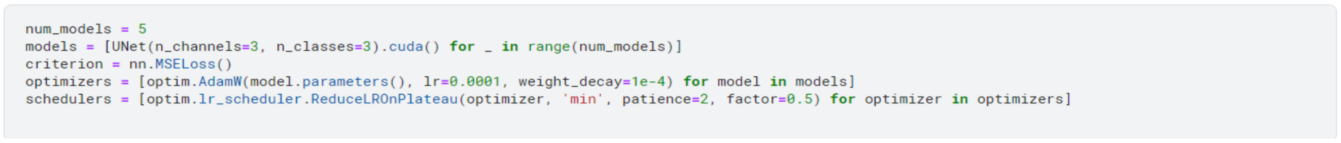
LOADING THE DATA



```
gamma_value = 2.2

transform = transforms.Compose([
    transforms.Resize((150, 150)),
    transforms.ToTensor(),
])

train_dataset = CustomDataset(train_low_image_paths, train_high_image_paths, transform=transform, gamma=gamma_value)
train_loader = DataLoader(train_dataset, batch_size=4, shuffle=True)

eval_dataset = CustomDataset(eval_low_image_paths, eval_high_image_paths, transform=transform, gamma=gamma_value)
eval_loader = DataLoader(eval_dataset, batch_size=4, shuffle=False)
```

10.

SCHEDULING AND OPTIMIZATION



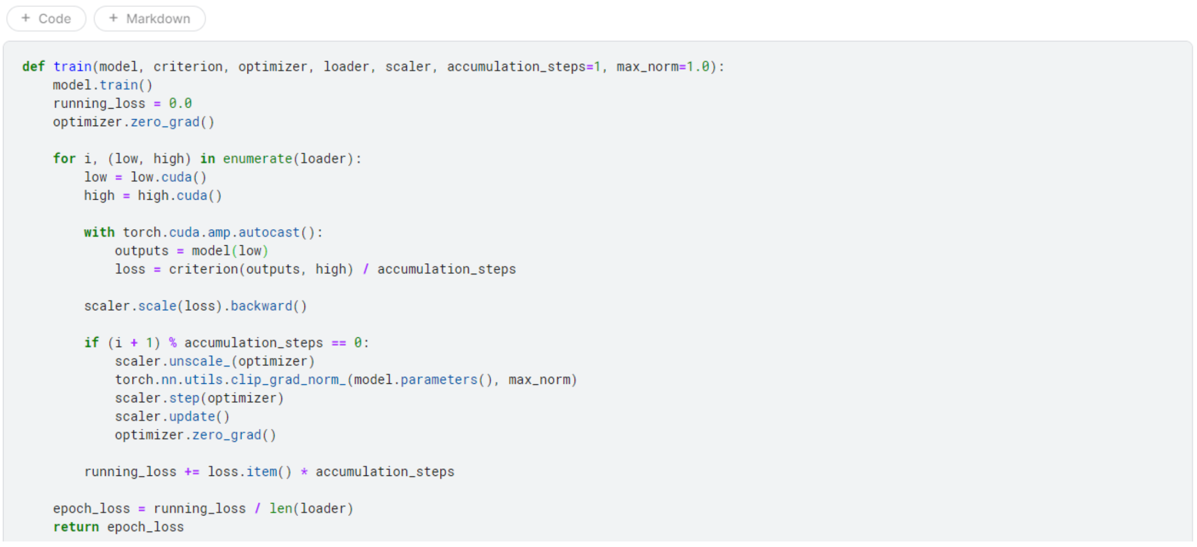
```
num_models = 5
models = [UNet(n_channels=3, n_classes=3).cuda() for _ in range(num_models)]
criterion = nn.MSELoss()
optimizers = [optim.AdamW(model.parameters(), lr=0.0001, weight_decay=1e-4) for model in models]
schedulers = [optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=2, factor=0.5) for optimizer in optimizers]
```

I have applied ensemble method for training multiple models and compare the best model among them so that I get the best model and most better psnr .

Also I have used Adam optimizer along with ReduceLROnPlateau for proper scheduling of the learning rate .

11.

TRAINING FUNCTION



```
def train(model, criterion, optimizer, loader, scaler, accumulation_steps=1, max_norm=1.0):
    model.train()
    running_loss = 0.0
    optimizer.zero_grad()

    for i, (low, high) in enumerate(loader):
        low = low.cuda()
        high = high.cuda()

        with torch.cuda.amp.autocast():
            outputs = model(low)
            loss = criterion(outputs, high) / accumulation_steps

        scaler.scale(loss).backward()

        if (i + 1) % accumulation_steps == 0:
            scaler.unscale_(optimizer)
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm)
            scaler.step(optimizer)
            scaler.update()
            optimizer.zero_grad()

        running_loss += loss.item() * accumulation_steps

    epoch_loss = running_loss / len(loader)
    return epoch_loss
```

• **Initialization**: Sets the model to training mode (model.train()), initializes the optimizer gradients (optimizer.zero_grad()), and initializes running_loss to accumulate the loss over batches.

• **Main Loop**: Iterates through batches in the loader. Each batch consists of low and high resolution images, which are transferred to the GPU (low.cuda() and high.cuda()).

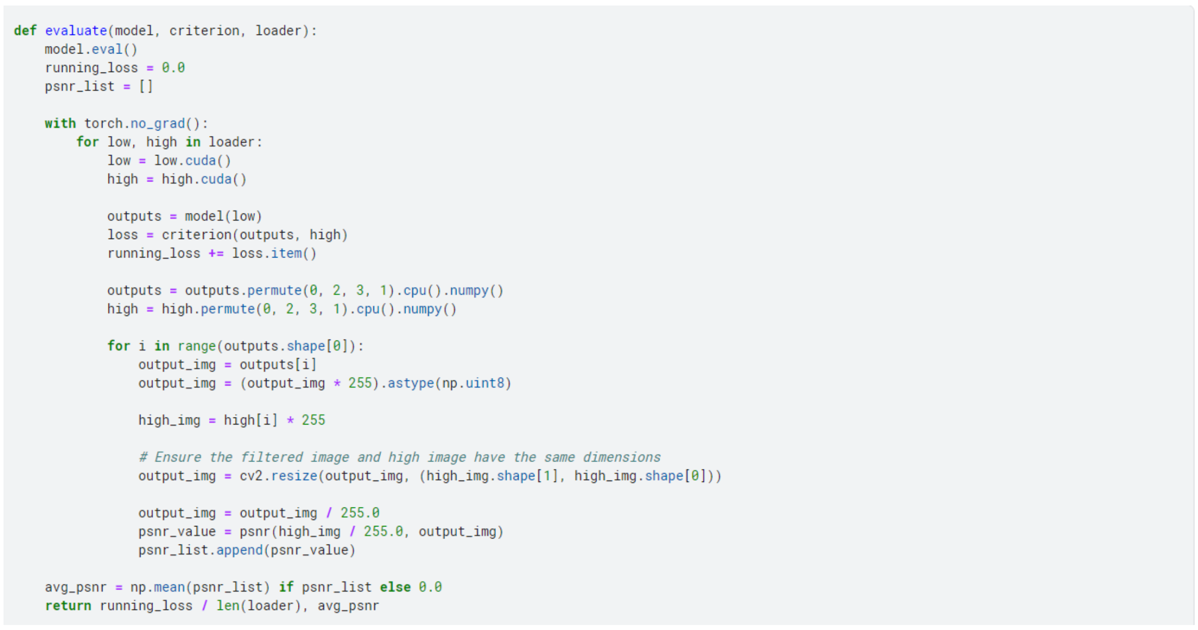
• **Training Steps**:

- **Autocast**: Uses automatic mixed precision training with torch.cuda.amp.autocast() to accelerate computations while minimizing numerical overflow.
- **Forward Pass**: Computes model predictions (outputs) for low resolution inputs and calculates the loss against high resolution targets.
- **Backward Pass**: Scales the loss and performs backpropagation (scaler.scale(loss).backward()) to compute gradients.
- **Gradient Accumulation**: Updates the optimizer every accumulation_steps batches. This involves unscaling the optimizer, clipping gradients to prevent explosion, stepping the optimizer, and updating the scaler.
- **Loss Calculation**: Accumulates the scaled loss (loss.item() * accumulation_steps) for reporting and averaging across the dataset.

• **Epoch Loss**: Calculates the average loss per batch (epoch_loss = running_loss / len(loader)) and returns it as the training loss for the epoch.

12.

EVALUATION FUNCTION



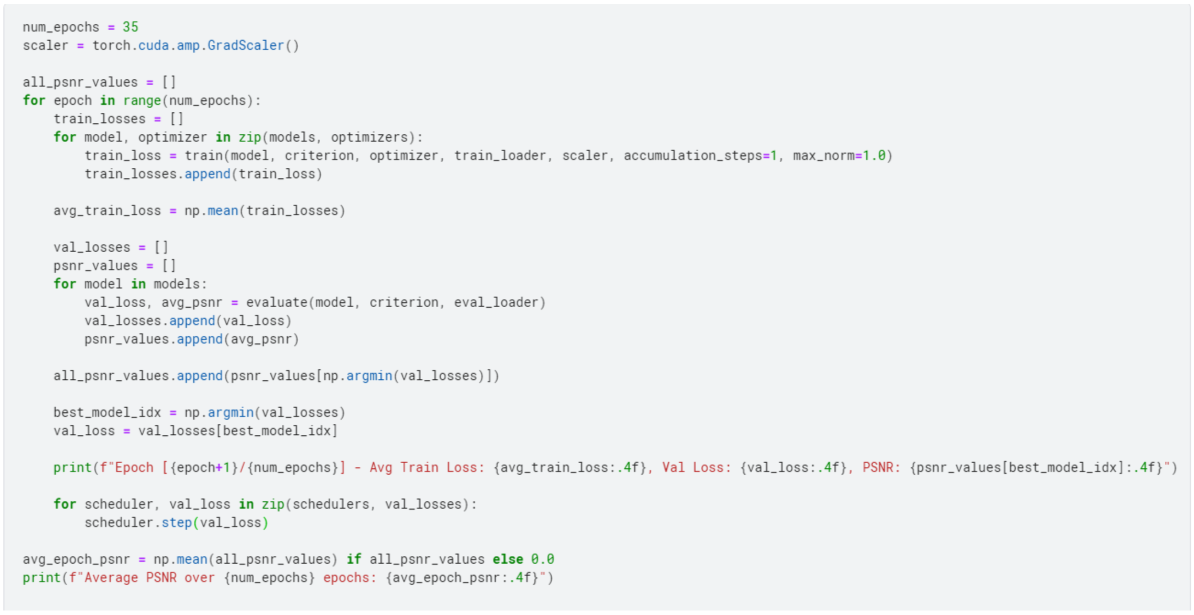
```python
def evaluate(model, criterion, loader):
    model.eval()
    running_loss = 0.0
    psnr_list = []

    with torch.no_grad():
        for low, high in loader:
            low = low.cuda()
            high = high.cuda()

            outputs = model(low)
            loss = criterion(outputs, high)
            running_loss += loss.item()

            outputs = outputs.permute(0, 2, 3, 1).cpu().numpy()
            high = high.permute(0, 2, 3, 1).cpu().numpy()

            for i in range(outputs.shape[0]):
                output_img = outputs[i]
                output_img = (output_img * 255).astype(np.uint8)

                high_img = high[i] * 255

                # Ensure the filtered image and high image have the same dimensions
                output_img = cv2.resize(output_img, (high_img.shape[1], high_img.shape[0]))

                output_img = output_img / 255.0
                psnr_value = psnr(high_img / 255.0, output_img)
                psnr_list.append(psnr_value)

    avg_psnr = np.mean(psnr_list) if psnr_list else 0.0
    return running_loss / len(loader), avg_psnr
```

• **Model Evaluation**: Sets the model to evaluation mode (model.eval()) to disable dropout and batch normalization.

• **Loss Calculation**: Iterates through batches in the loader, computes model predictions (outputs) for low resolution inputs, and calculates the loss against high resolution targets using the specified criterion.

• **Performance Metrics**: Computes the average loss (running_loss / len(loader)) over the entire dataset. Additionally, it calculates the Peak Signal-to-Noise Ratio (PSNR) for each batch, comparing the model's output to the ground truth (high resolution images). PSNR values are averaged (avg_psnr) across all batches to quantify image quality.

13. TRAINING LOOP



```
num_epochs = 35
scaler = torch.cuda.amp.GradScaler()

all_psnr_values = []
for epoch in range(num_epochs):
    train_losses = []
    for model, optimizer in zip(models, optimizers):
        train_loss = train(model, criterion, optimizer, train_loader, scaler, accumulation_steps=1, max_norm=1.0)
        train_losses.append(train_loss)

    avg_train_loss = np.mean(train_losses)

    val_losses = []
    psnr_values = []
    for model in models:
        val_loss, avg_psnr = evaluate(model, criterion, eval_loader)
        val_losses.append(val_loss)
        psnr_values.append(avg_psnr)

    all_psnr_values.append(psnr_values[np.argmin(val_losses)])

    best_model_idx = np.argmin(val_losses)
    val_loss = val_losses[best_model_idx]

    print(f"Epoch [{epoch+1}/{num_epochs}] - Avg Train Loss: {avg_train_loss:.4f}, Val Loss: {val_loss:.4f}, PSNR: {psnr_values[best_model_idx]:.4f}")

    for scheduler, val_loss in zip(schedulers, val_losses):
        scheduler.step(val_loss)

avg_epoch_psnr = np.mean(all_psnr_values) if all_psnr_values else 0.0
print(f"Average PSNR over {num_epochs} epochs: {avg_epoch_psnr:.4f}")
```

• **Training Loop**: For each epoch, it iterates through each model and corresponding optimizer in models and optimizers, respectively. It calls the train function to optimize the models using the train_loader, calculating and storing the average training loss (avg_train_loss).

• **Validation and Evaluation**: After training, it evaluates each model (evaluate function) on the validation dataset (eval_loader). It computes validation loss (val_loss) and evaluates the average Peak Signal-to-Noise Ratio (PSNR) of the model's predictions compared to ground truth high-resolution images. The PSNR values are collected in psnr_values.

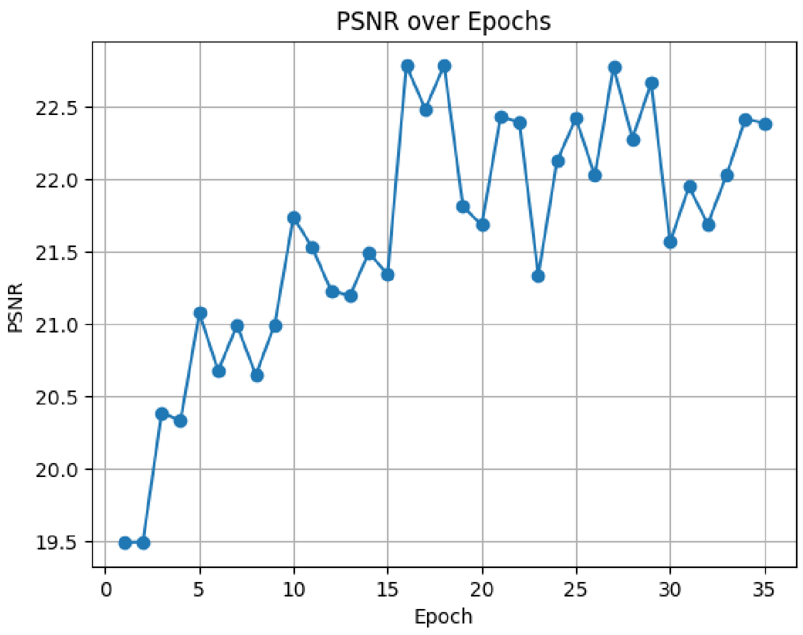
• **Learning Rate Scheduling**: Each optimizer's learning rate scheduler (schedulers) adjusts the learning rate based on the validation loss (val_loss).

• **Metrics Tracking**: Throughout each epoch, it prints the average training loss, validation loss, and PSNR of the best performing model. At the end of training, it calculates the average PSNR across all epochs (avg_epoch_psnr) and prints this metric.

14.

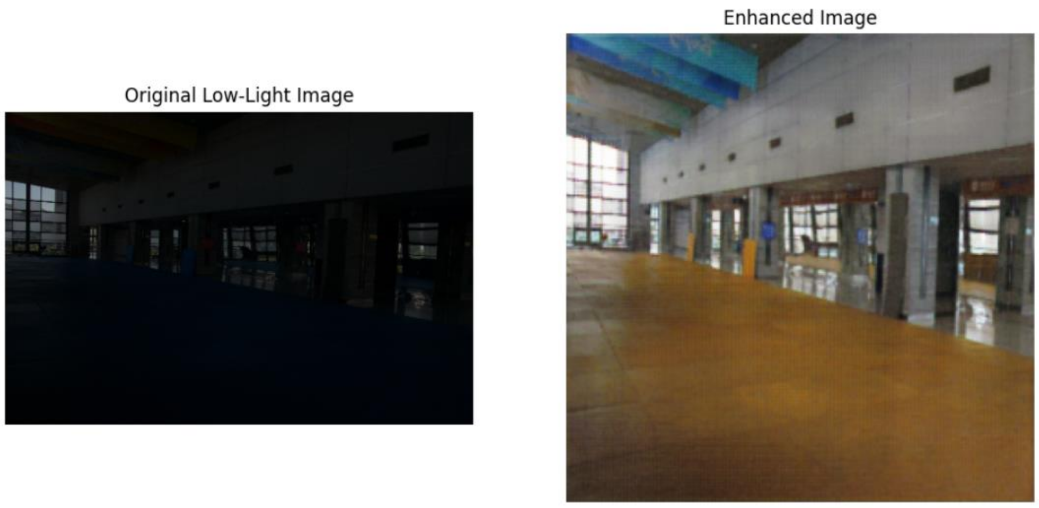
GRAPH PLOT

```
import matplotlib.pyplot as plt

# Plot PSNR values over epochs
plt.plot(range(1, num_epochs+1), all_psnr_values, marker='o')
plt.xlabel('Epoch')
plt.ylabel('PSNR')
plt.title('PSNR over Epochs')
plt.grid(True)
plt.show()
```



15.
ENHANCING TEST IMAGE

```
import matplotlib.pyplot as plt
def apply_gamma_correction(img_np, gamma):
    inv_gamma = 1.0 / gamma
    table = (torch.arange(0, 256, dtype=torch.float32) / 255.0) ** inv_gamma * 255
    table = table.clamp(0, 255).byte().cpu().numpy()
    img_np = cv2.LUT(img_np, table)
    return img_np


def enhance_image(model, image_path, transform, gamma=None):
    model.eval()
    image = Image.open(image_path).convert('RGB')

    if gamma:
        image_np = np.array(image)
        image_np = apply_gamma_correction(image_np, gamma)
        image = Image.fromarray(image_np)

    image = transform(image).unsqueeze(0).cuda()

    with torch.no_grad():
        enhanced_image = model(image).squeeze(0).cpu()
    return transforms.ToPILImage()(enhanced_image)
# Path to a sample low-light image
sample_image_path = '/kaggle/input/loldataset-2/LOLdataset/eval15/low/780.png'
# Enhance the image using the best model
best_model = models[best_model_idx]  # Assuming best_model_idx is determined from the training process
enhanced_image = enhance_image(best_model, sample_image_path, transform, gamma=gamma_value)
# Display the original and enhanced images
original_image = Image.open(sample_image_path).convert('RGB')
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.title("Original Low-Light Image")
plt.imshow(original_image)
plt.axis('off')
plt.subplot(1, 2, 2)
plt.title("Enhanced Image")
plt.imshow(enhanced_image)
plt.axis('off')
plt.show()
```



16.HYPER PARAMETERS TUNING

I run a separate code for finding the best hyperparameters and after finding I applied in my model . Here is the notebook from which I found best combination of hyperparameters.

Best Hyper-parameters obtained are:

Btach Size-4

Learning rate:0.0005

Accumulation-steps:1

low-lie (10).ipynb

PROBLEMS FACED:

1.GPU and computational power limits issues.

2.Finding the correct model architecture wich should be used

3.Cuda out of memory: I faced this error for a long time (for 2-3 days) I was not able to remove the error for many days.

4.Size mis-match: Since I have used U-net model architecture so dimension error occurred because of downsampling and upsampling.

5.win_size error: for calculating SSIM I got this error and treid to debug but can't remove it.

ENCOUNTERING THE CHALLENGES:

1.First, I studied on Retinex-Former theory and went to implement that but faced memory problems. So applied gradient accumulation, decreasing of batch size, gradient clipping, decreasing the size of image and many more to remove memory issues. After 3 days it removed but I got very less psnr (17-18). So, I changed the model architecture and switched to U-net model architecture.

2.For size-mismatch revised the whole code accordingly.

3.I can’t debug win_size error so I removed SSIM loss from evaluation metrics.

REFERECES:

1. U-net model : https://arxiv.org/pdf/1805.01934

2. Google and ChatGPT: For removing errors and debugging tips.

3. Dataset: LOLv1 Dataset in From The document provided by you.

4.Mostly I have prepared the model using google rather than implementing a research paper.