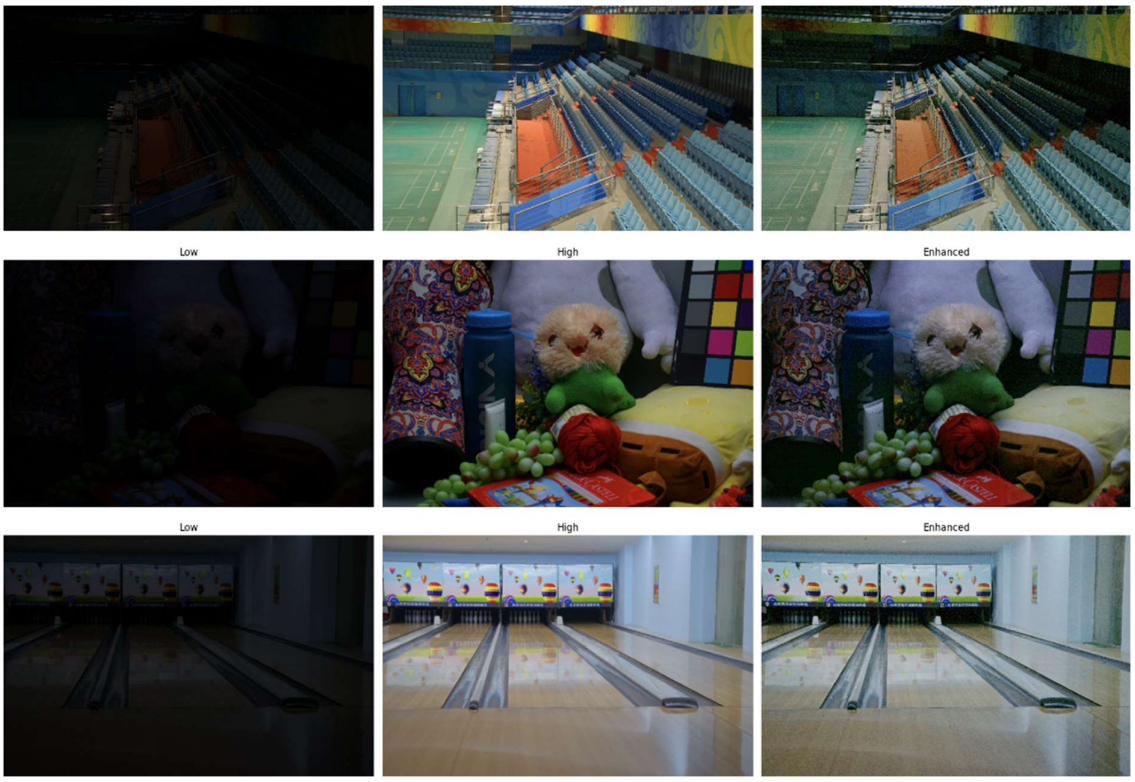
Project Report

Extreme Low Light Image Denoising



Name: Atharva Sonare

Enrollment no: 22116022

Dept & Year: ECE III

June 2024

Introduction

The aim of the project was to denoise images taken in extreme low light conditions. We often find ourselves with these low light images due to technical and environmental constraints. This results in loss of information. While there are many techniques to denoise such images, most of them are computationally expensive. Therefore, we can use Deep neural networks to perform the task.
In this project we mainly explored models DCE Net and UNet and implemented low light image denoising using various loss functions. Eventually taking our modified implementation of DCE Net (Partial Reference DCE) as our final model due to better output quality.

Zero DCE

About the Paper: A lightweight deep network, DCE-Net is used to estimate pixel-wise and high-order curves for dynamic range adjustment of a given image. The curve estimation is specially designed, considering pixel value range, monotonicity, and differentiability. Zero DCE does not require any paired data, this is done by set of non-reference loss functions.

Learning to see in the dark

The paper "Learning to See in the Dark" addresses the challenge of low-light imaging by introducing a data-driven approach using deep neural networks. The authors present a new dataset of raw short-exposure low-light images with corresponding long-exposure reference images to train and evaluate their fully convolutional network.

Partial Reference Deep Curve Estimation

Since we have access to a dataset that includes both low-light input images and their corresponding high-quality reference images, we can leverage this information to improve Zero-DCE denoising model. Reference loss functions such as Mean Squared Error (MSE) and Mean Absolute Error (MAE) are particularly useful in this scenario.

By incorporating these reference loss functions into our training process, our model learned to produce denoised images that closely resemble the high-quality references. This approach will often lead to higher PSNR scores compared to models trained solely on low-light input images without reference supervision.

After generating denoised images using our model, we applied post-processing techniques that further refined the results without sacrificing visual quality.

Optimization of loss weights was crucial for achieving optimal performance. We used Optuna, a hyperparameter optimization framework that automated the search for optimal hyperparameters, such as the weights assigned to different loss functions.

Dataset Used: LOL Dataset

The LOL dataset is composed of 500 low-light and normal-light image pairs and is divided into 485 training pairs and 15 testing pairs. The low-light images contain noise produced during the photo capture process. Most of the images are indoor scenes. All the images have a resolution of 400×600.

Pipeline (Training)

Loading and Preprocessing -----Defining Network function ----Defining Losses ----Defining Custom Model class---Training the Model----Plotting Learning curves----Applying---Post processing and finally calculating the PSNR values

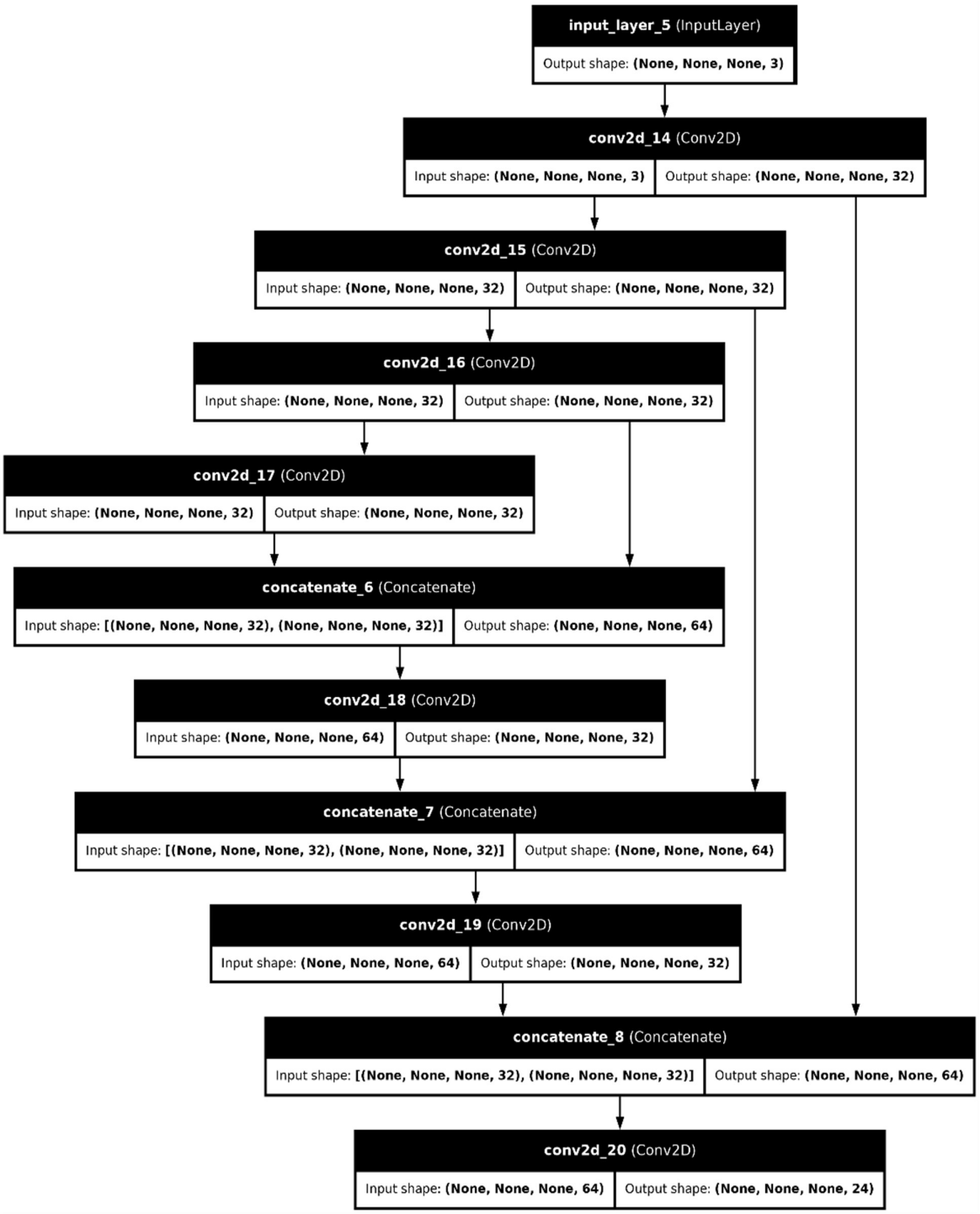
UNET Architecture for the implementation

```
self.encoder1 = block(in_channels, 64)
self.encoder2 = block(64, 128)
self.encoder3 = block(128, 256)
self.pool = nn.MaxPool2d(2)
self.middle = block(256, 512)
self.upconv3 = nn.Conv2d(512, 256, kernel_size=1)
self.decoder3 = block(512, 256)
self.upconv2 = nn.Conv2d(256, 128, kernel_size=1)
self.decoder2 = block(256, 128)
self.upconv1 = nn.Conv2d(128, 64, kernel_size=1)
self.decoder1 = block(128, 64)
self.out_conv = nn.Conv2d(64, out_channels, kernel_size=1)
```

```
def block(in_channels, out_channels):
    return nn.Sequential(
        nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1),
        nn.ReLU(inplace=True)
    )
```

```
    def forward(self, x):
        # Apply amplification factor
        x = x * self.amplification_factor
        enc1 = self.encoder1(x)
        enc2 = self.encoder2(self.pool(enc1))
        enc3 = self.encoder3(self.pool(enc2))
        middle = self.middle(self.pool(enc3))
        dec3 = F.interpolate(middle, scale_factor=2, mode='bilinear', align_corners=True)
        dec3 = self.center_crop(enc3, dec3)
        dec3 = torch.cat((dec3, enc3), dim=1)
        dec3 = self.decoder3(dec3)
        dec2 = F.interpolate(dec3, scale_factor=2, mode='bilinear', align_corners=True)
        dec2 = self.center_crop(enc2, dec2)
        dec2 = torch.cat((dec2, enc2), dim=1)
        dec2 = self.decoder2(dec2)
        dec1 = F.interpolate(dec2, scale_factor=2, mode='bilinear', align_corners=True)
        dec1 = self.center_crop(enc1, dec1)
        dec1 = torch.cat((dec1, enc1), dim=1)
        dec1 = self.decoder1(dec1)
        y = self.out_conv(dec1)
#         y = y * amplification_factor
        return y
    def center_crop(self, enc, dec):
        _, _, H, W = enc.size()
        _, _, h, w = dec.size()
        x1 = (W - w) // 2
        y1 = (H - h) // 2
        return enc[:, :, y1:y1+h, x1:x1+w]
```

DCE Architecture for the implementation



Preprocessing

Partial DCE

- Read: `tf.io.read_file(image_path)` reads the image file.
- Decode: `tf.image.decode_png(image, channels=3)` decodes it to a 3-channel tensor.
- Resize: `tf.image.resize(image, [IMAGE_SIZE, IMAGE_SIZE])` resizes it to 256x256.
- Normalize: `image / 255.0` scales pixel values to [0, 1].

The generator function maps the low light images to corresponding high light images

UNet

- Load image filenames from low-light (input) and high-light (target) directories.
- For each image pair, load and convert images to RGB.
- Apply optional transformations (e.g., resizing, normalization) to both images.
- Return paired low-light and high-light images ready for model training.

All of this preprocessing is achieved via LowLightDataset Class in code.

Training

Partial DCE

- We created a custom class that we can used directly to train the model calling the fit method.
- We store the training data in a history object to plot the learning curves right after.
- To check for overfitting, we initially train over only 400 images keeping the other 85 images for validation afterwards train over 485 and test over 15.
- Batch size = 16, Epochs = 60 (curves show that larger number of epochs do not help), learning rate = 1e-4, check with reducing learning rate, did not have any impact.
- Optimizer = Adam
- Loss functions are discussed afterwards, the weights (reference losses) were tuned using Optuna
- GPU: Kaggle T4 X 2

UNet

- Batch size = 4
- Optimizer = Adam
- Learning Rate = 1e-3(for 10) 1e-4(for next 20) 1e-5(for next 20) Total Epoch = 50
- Loss functions are a combination of MSE, which is called Color Loss here, Perceptual Loss (using VGG 19), L1 Loss
- Weights and Amplification factor found using Optuna

- GPU: Kaggle P100

Evaluation

Partial DCE

Train (400)

```
Mean PSNR: 17.35
Median PSNR: 17.55
Min PSNR: 6.00
Max PSNR: 28.17
```

Validation (85)

```
Mean PSNR: 17.50
Median PSNR: 17.66
Min PSNR: 6.00
Max PSNR: 26.30
```

Train (485)

```
Mean PSNR: 17.39
Median PSNR: 17.50
Min PSNR: 6.36
Max PSNR: 28.18
```

Test (15 images)

```
Mean PSNR: 19.50
Median PSNR: 20.68
Min PSNR: 11.04
Max PSNR: 25.99
```

Performed on different parts of the given dataset, the images had good visual quality, but on closes inspection, uniformly distributed noise could be observed. Paper had a PSNR of about 16.57.

UNet

Train (485)

```
Mean PSNR: 17.310
Median PSNR: 17.4
Min PSNR:11.66713
Max PSNR:24.14971
```

Test (15)

```
Mean PSNR: 19.1347
Median PSNR: 19.11
Min PSNR:12.264968
Max PSNR:24.778503
```

The images are good but lack in quality and PSNR when compared to Partial DCE model.

Loss Functions

Non-Reference loss functions (Partial DCE)

1. Color Constancy Loss

$$L_{col} = \sum_{\forall(p,q)\in\varepsilon} (J^p - J^q)^2, \varepsilon = \{(R,G),(R,B),(G,B)\}, \tag{6}$$

It corrects potential color deviation and builds relations among the three adjusted channels. where J^p denotes the average intensity value of p channel in the enhanced image, (p,q) represents a pair of channels

2. Exposure Control Loss

$$L_{exp} = \frac{1}{M}\sum_{k=1}^{M} |Y_k - E|, \tag{5}$$

It restrains under and overexposed regions, setting E = 0.6 as gray level, where M represents the number of nonoverlapping local regions of size 16×16, Y is the average intensity value of a
local region in the enhanced image

3. Illumination Smoothness Loss

$$L_{tv_{\mathcal{A}}} = \frac{1}{N}\sum_{n=1}^{N}\sum_{c\in\xi}(|\nabla_x \mathcal{A}_n^c| + \nabla_y \mathcal{A}_n^c|)^2, \xi = \{R,G,B\}, \tag{7}$$

Preserves monotonicity relations between neighboring pixels. N is the number of iterations, ∇x and ∇y represents the horizontal and vertical gradient operations, respectively

4.Spatial Constancy Loss

$$L_{spa} = \frac{1}{K}\sum_{i=1}^{K}\sum_{j\in\Omega(i)} (|(Y_i - Y_j)| - |(I_i - I_j)|)^2, \tag{4}$$

Encourages coherence of enhanced image by preserving difference of neighboring regions between input image and enhanced version.

Reference loss functions (Common in both implementations)

1. Supervised Loss

$$\text{Supervised Loss}(y_{\text{true}}, y_{\text{pred}}) = \frac{1}{n}\sum_{i=1}^{n}(y_{\text{true},i} - y_{\text{pred},i})^2 + 0.1 \times \frac{1}{n}\sum_{i=1}^{n}|y_{\text{true},i} - y_{\text{pred},i}|$$

   It's a combination of MSE and MAE loss, it helps to encourage the model to have a high PSNR, there are other benefits to MAE inclusion too such as robustness, sparsity etc.

2. SSIM loss

$$\text{SSIM Loss}(y_{\text{true}}, y_{\text{pred}}) = 1 - \frac{1}{n}\sum_{i=1}^{n}\text{SSIM}(y_{\text{true},i}, y_{\text{pred},i})$$

   By focusing on the similarity in structure, SSIM is better at preserving edges and fine details, which are crucial for high-quality image reconstruction. Hence, we create a loss function for that too.

3. Perceptual Loss

$$\mathcal{L}_{\text{perceptual}}(y_{\text{true}}, y_{\text{pred}}) = \text{MSE}(\text{VGG}_{\text{block5\_conv2}}(y_{\text{true}}), \text{VGG}_{\text{block5\_conv2}}(y_{\text{pred}}))$$

   Perceptual loss function with VGG features provides a powerful approach by leveraging learned representations that are sensitive to perceptual quality rather than just pixel-level accuracy.

Total Loss:

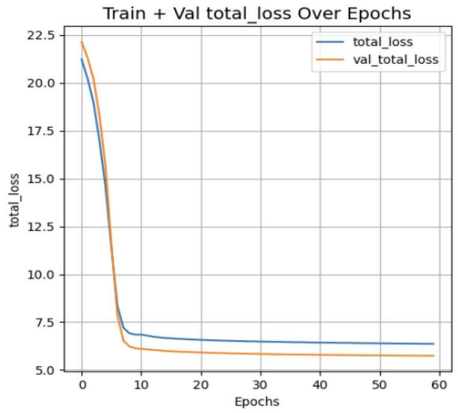
Its summation of Products of weights and losses

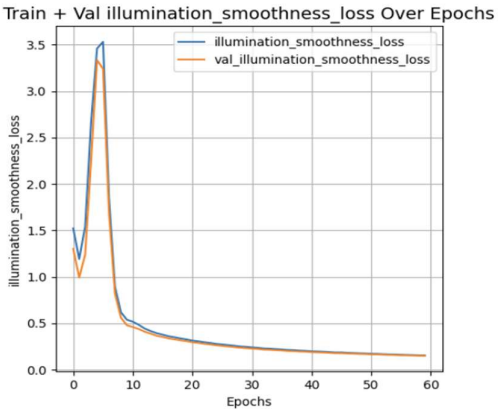
Post processing for Partial DCE

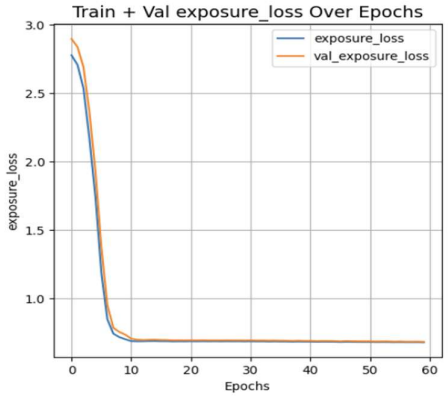
Gaussian blurring and Non-Local Means Denoising

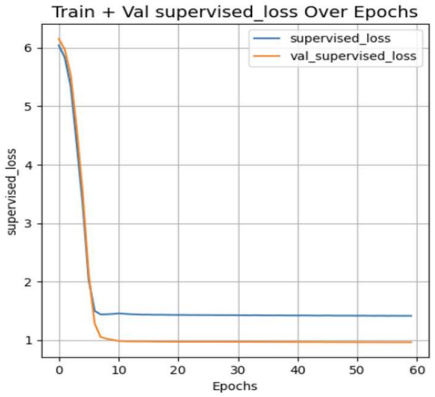
```python
def apply_gaussian_blur(image, kernel_size=(3, 3), sigma=0.6):
    image_np = np.array(image)
    blurred_image_np = cv2.GaussianBlur(image_np, kernel_size, sigma)
    return blurred_image_np

def post_process(image):
    image_np = np.array(image)
    denoised_image = cv2.fastNlMeansDenoisingColored(image_np, None, 7, 7, 5 ,5)
    denoised_image = apply_gaussian_blur(denoised_image)
    return Image.fromarray(denoised_image)
```

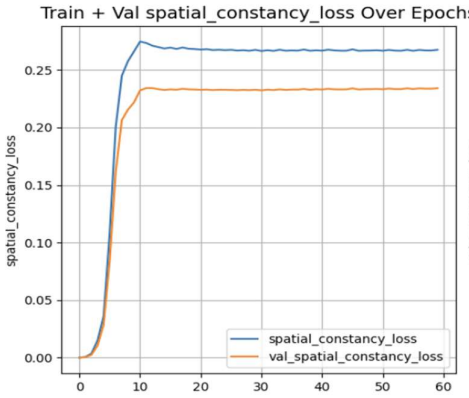
The reason for choosing these methods was that they are computationally cheap and do not tamper with the image quality while helping in getting a better PSNR ratio.

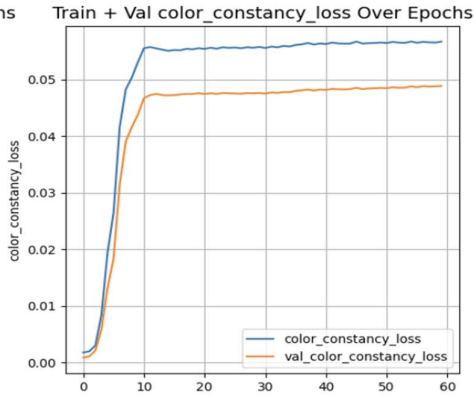




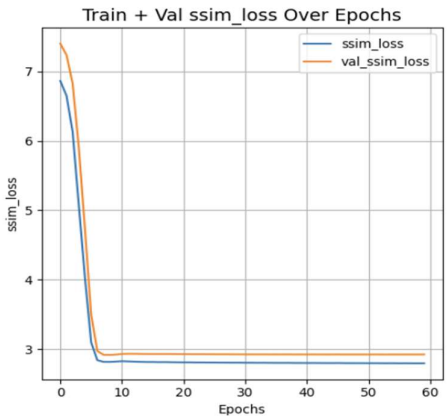


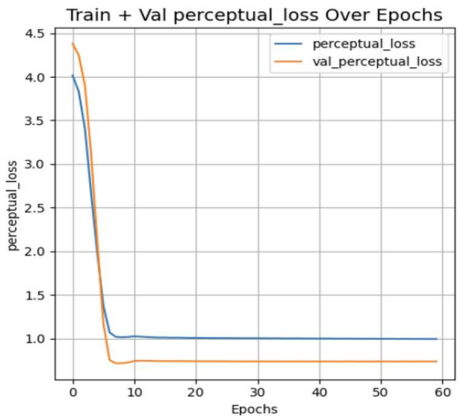






LEARNING CURVES DCE





Model Limitations and Challenges Faced:

Partial DCE and UNet

- Zero-DCE models excel in enhancing low-light images but typically focus on improving brightness and contrast rather than denoising, this issue persists in Partial DCE as well despite the inclusion of reference function, we don't get any extraordinary improvement with respect to PSNR.
- Training requires lots of time, which restrains us from running Optuna for too long, hence we can run less no of trials, which means we could have potentially better hyper parameters
- While UNet gets higher PSNR on complete 500 images compared to DCE on training data, its images are not visually appealing.
- GPU limits on Kaggle and 12 hour running limits in Kaggle presented an obstacle to tuning parameters.

Observations:

- Oddly the training process seemed to behave differently on different GPU's, which is why the model seemed to behave erratically differently on certain GPU's. I was not able to work out the exact reason, but it must be related to TensorFlow and its GPU usage algorithms
- While the output images seem to look good if you look closely, you can still see uniformly distributed noise across the image in certain places, this is also evident with the low increase in PSNR for Partial DCE.

Improvements:

- If we have a large and robust dataset, we might get better results, we could try patching the data, to simulate a larger dataset or try data

augmentation to simulate robust dataset or both together, my model did not show any improvements upon implementing these.

- If we have more resources, we could perform complete hyperparameter tuning even tune the parameters for post processing techniques this could result in significant improvement
- We could also cascade a pretrained denoiser to have more noise free images specially for Partial DCE, I avoided this as we were told not to use pretrained model.

References:

1. Research Paper: Zero-Reference Deep Curve Estimation for Low Light enhancement -- Chunle Guo, Chongyi Li etc.

   Link: https://arxiv.org/pdf/2001.06826

2. Research Paper: Learning to See in the Dark-- Chen Chen, Jia Xu etc.

   Link: https://arxiv.org/pdf/1805.01934