Low-light Image Enhancement

Introduction

The project "Low-light Image Enhancement" implements the Zero-Reference Deep Curve Estimation (Zero-DCE) model. Zero-DCE is designed to enhance low-light images by estimating the optimal curve for adjusting image illumination without the need for reference images.

Zero-DCE Architecture

Zero-DCE is a lightweight convolutional neural network (CNN) designed to enhance low-light images through an end-to-end trainable deep neural network. It uses a deep curve estimation approach, where a set of best-fitting curves are predicted to adjust pixel intensities for improved illumination.

Key Specifications

- **Model Type**: Convolutional Neural Network (CNN)
- **Layers**: The model consists of multiple convolutional layers followed by non-linear activations to learn the mapping from low-light to enhanced images.
- **Loss Functions**: A combination of spatial consistency loss, exposure control loss, color constancy loss, and illumination smoothness loss is used.
- **Framework**: Implemented in Keras with a TensorFlow backend.
- **PSNR Value**: The Peak Signal-to-Noise Ratio (PSNR) achieved by the model varies depending on the dataset but is competitive with state-of-the-art methods

Reference Paper

The project is based on the paper "Zero-Reference Deep Curve Estimation for Low-Light Image Enhancement" by Chunle Guo. The paper can be accesse here.

Project Details

This Project contains the implementation of the Zero-DCE model using Keras. The notebook includes sections for data loading, model building, training, and evaluation. Below, I provide an overview of the code snippets, their purposes, and relevant visualizations.

Importing Libraries

The first step involves importing essential libraries such as TensorFlow/Keras for building and training the neural network, NumPy for numerical operations, and Matplotlib for visualizing results.

Data Loading

Images are loaded and pre-processed to ensure they are in the correct format and size for the model. Preprocessing typically includes resizing images to a standard size, normalizing pixel values, and converting them to the appropriate data type for input into the neural network.

There are total 486 Low Light as well as High Light Images in our dataset. Out of which starting 350 images are used to Train the model, next 100 images are used to validate the model and rest images are used for Testing the model. We've used 64 as the Batch Size.

```
Train Dataset: <_BatchDataset element_spec=TensorSpec(shape=(64, 256, 256, 3), dtype=tf.float32, name=None)>
Validation Dataset: <_BatchDataset element_spec=TensorSpec(shape=(64, 256, 256, 3), dtype=tf.float32, name=None)>
```

Model Building - Deep Curve Estimation (DCE) Network

The Zero-DCE model is constructed using several convolutional layers. These layers are designed to capture different levels of features in the images. The initial layers typically capture low-level features such as edges and textures, while the deeper layers capture more complex patterns and structures.

The convolutional layers are followed by non-linear activation functions, which introduce non-linearity into the model and enable it to learn more complex mappings from input images to enhanced images. The final layer of the model outputs the enhanced image, which is then compared to the ground truth image during training.

Color Constancy Loss

It can be calculated as the Square Root of the Sum of Squared difference between mean RGB values = sqrt(d_rg^2 + d_rb^2 + d_gb^2

where, d_rg, d_rb, and d_gb are squared differences between mean RGB values

Exposure Loss

It can be calculated as the mean squared difference between pooled_mean and target_mean. And the Average pooling is performed on mean_intensity to compute the overall mean with a kernel size (ksize) of 16x16 and strides (strides) of 16.

Illumination Smoothness Loss

This loss measures how smoothly illumination changes across adjacent pixels in the image tensor, promoting spatial consistency in illumination. The formula to calculate it is:

$$\text{illumination\_smoothness\_loss} = \frac{2 \times (h\_tv/\text{count\_h} + w\_tv/\text{count\_w})}{\text{batch\_size}}.$$

where, h_tv and w_tv are the sum of squared differences between horizontally and vetically adjacent pixels across all channels in each image of the batch respectively and count_h and count_w are the total number of horizontal and vertical pairwise differences respectively.

Spatial Consistency Loss

It returns the sum of squared differences (d_left + d_right + d_up + d_down) and measures how well the enhanced image preserves the spatial arrangement of illumination from the original image,

where d_original_left, d_original_right, d_original_up, d_original_down are the spatial differences between adjacent pixels in original_pool using convolution operations, and d_enhanced_left, d_enhanced_right, d_enhanced_up, d_enhanced_down are the spatial differences between adjacent pixels in enhanced_pool.

Zero-DCE Model

- **Compilation (compile method)**:
  - Configures the optimizer as Adam with a specified learning_rate.
  - Initializes spatial_constancy_loss as an instance of SpatialConsistencyLoss with reduction="none".
- **Image Enhancement (get_enhanced_image method):**
  - Enhances the input data using the outputs (output) from the DCE network.
  - Splits output into eight components (r1 to r8).
  - Iteratively enhances data using these components based on a formula involving element-wise operations with the components.

- **Training Step (train_step method)**:

  - Computes gradients of the total loss with respect to trainable weights of dce_model using tf.GradientTape.
  - Updates weights of dce_model using the optimizer based on computed gradients.
  - Returns losses computed during the step.

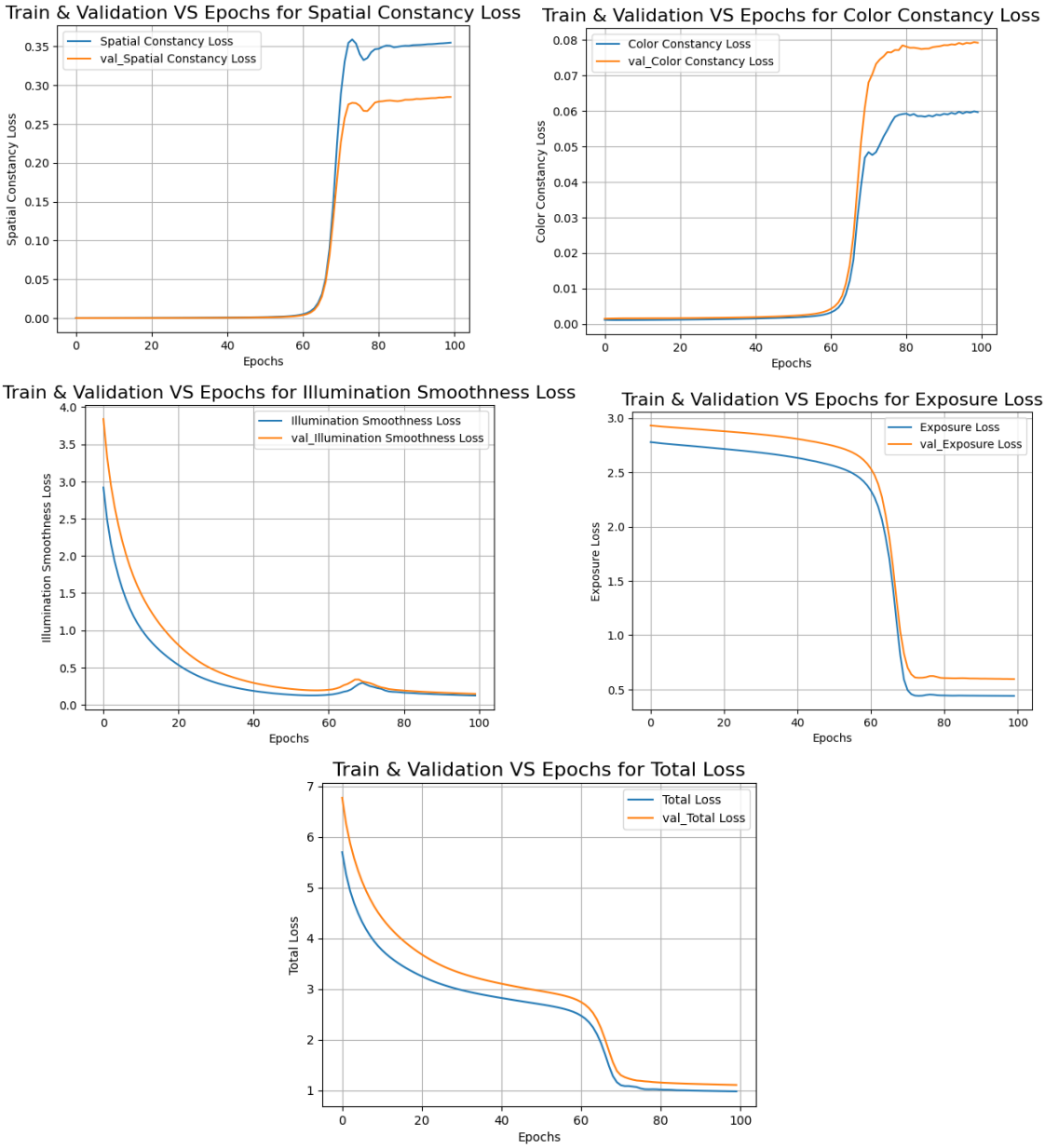
- **Testing (test_step method)**:

  - Performs forward pass through dce_model to get outputs and computes losses using compute_losses.
  - Returns losses computed during testing.

Training

Training the model involves compiling it with an optimizer and a loss function, and then fitting it to the training data. The optimizer updates the model's weights based on the gradients of the loss function with respect to the weights. The loss function measures the difference between the predicted and ground truth images, guiding the optimizer in minimizing this difference. The training process includes monitoring the loss values over epochs. In our model, we've used 100 epochs.

Diagrams and Graphs for different Losses:



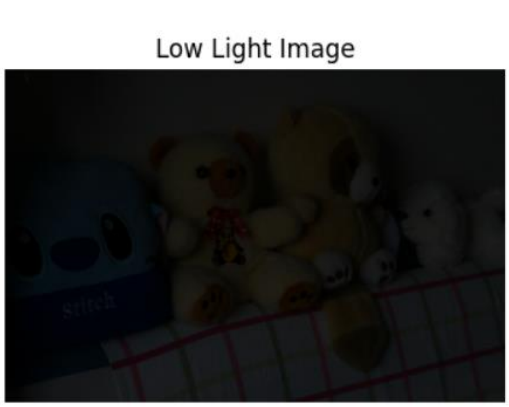
Evaluation and Enhancement

After training, the model is used to enhance low-light images. The evaluation process involves comparing the enhanced images to the original low-light images and calculating metrics such as PSNR (Peak Signal-to-Noise Ratio) to quantify the improvement in image quality.

We have tested last 36 images and displayed the corresponding predicted and high light images along with their PSNR value. **The average PSNR value in my results is around 28 dB.**

```
<ipython-input-36-0a51f8d382fc>:2: RuntimeWarning: More than 20 figures have been opened.
  fig,axs = plt.subplots(1,3,figsize=(15,5))
Average PSNR: 28.106694650798595
```

In Testing, the trained model produces outputs images like:



Summary

In this project, we implemented the Zero-DCE model to enhance low-light images. The model utilizes a deep learning approach to estimate enhancement curves, significantly improving the illumination of images without needing reference images. The Zero-DCE model estimates light enhancement curves (LE-curves) to improve image quality iteratively

Methods to Further Improve the Project

1. **Data Augmentation**: Enhancing the training dataset with more variations of low-light conditions can improve the model's robustness and performance
2. **Hyperparameter Tuning**: Techniques such as grid search or random search can be used to systematically explore different hyperparameter settings.
3. **Advanced Loss Functions**: Incorporating more sophisticated loss functions or a combination of losses tailored for low-light enhancement can further improve image quality.
4. **Real-time Processing**: Optimizing the model for real-time image enhancement, potentially using techniques like model quantization or pruning, can make the model more practical for real-world applications
5. **Cross-Validation**: Implementing cross-validation ensures the model's generalizability and helps avoid overfitting, leading to more reliable performance across different datasets.

------- By

Sachin Kumar
22115133
EE, 2Y