Low-Light Image Enhancement using Zero-Reference Deep Curve Estimations

CS7810 Image Enhancement Project

Hrigved Suryawanshi, Haard Shah, Matthew Ray

09/29/2024

Abstract

This project aimed to enhance low-light images using Zero-Reference Deep Curve Estimation (Zero-DCE), as inspired by the work of Chunle Guo et al. Zero-DCE leverages a deep neural network (DCE-Net) to estimate pixel-wise, high-order curves for dynamic range adjustment, enabling enhanced image illumination without the need for paired or unpaired reference images. The process employs non-reference loss functions such as spatial consistency loss, exposure control loss, color constancy loss, and total variation loss, which help improve the visual quality of low-light images by balancing exposure and preserving details.

For this implementation, the team initially tested the model using provided data to validate its efficacy, followed by experiments with an external dataset comprising low-light images and video. The results showed significant enhancement in brightness and detail retention, with improvements over the original implementation in avoiding oversaturation and color distortion. The model was also successfully adapted to enhance low-light video, though some initial frames remained dark. This project demonstrated the potential of Zero-DCE in enhancing low-light media for practical applications in image and video quality improvement.

Introduction and prior work

Inspired by several works on convolutional neural networks, enhancing low-light images, and zero-curve estimation, the team wanted to implement a solution to enhance low-light images. Many images and videos are taken in sub-optimal lighting, leading to poor quality due to reduced contrast, increased noise, and loss of detail in both shadow and bright areas. These issues make it difficult to discern visual information, impacting both aesthetics and practical use in real-world applications.

The main point of reference for this project is "Zero-Reference Deep Curve Estimation for Low-Light Image Enhancement" - by Chunle Guo, Et. Al. Zero-Reference Deep Curve Estimation (Zero-DCE) enhances low-light images by training a lightweight deep network (DCE-Net) to estimate pixel-wise, high-order curves for dynamic range adjustment without requiring paired or unpaired reference images. Zero-reference estimation does not require paired or unpaired data in the training process like existing CNN and GAN-based methods. This is made possible through non-reference loss functions, including spatial consistency loss, exposure control loss, color constancy loss, and illumination smoothness loss, all of which take into consideration multiple factors of light enhancement.

Primary code from Zero-Reference Deep Curve Estimation for Low-Light Image Enhancement repo was leveraged for this project. The model was trained and tested on the supplied data to validate the model. An exterior image dataset was then trained and tested on the model, as well as a few videos to determine the success of the model leveraging outside data.

For additional learning and exploration, resources were referenced exploring convolutional neural networks and low-light image enhancement.

- "A Newbie's Introduction to Convolutional Neural Networks" - Chi-Feng Wang, 2018
- "A Basic Introduction to Separable Convolutions" - Chi-Feng Wang, 2018
- "Breaking Through the Darkness: How to Enhance Low-Light Images with Deep Learning Techniques" - Benjamin Cham, 2023

Method

Our initial execution of the zero-reference deep curve estimation model leveraged the provided code for and supplied data. This was done to test the default implementation of the model. The training data was run through 60 epochs to develop the non-reference loss functions before the test data was run through the model.

Comparison between original vs enhanced Image

The output from the default implementation of the model was successful in enhancing the original low-light images. However, there was a loss of clarity and oversaturation to the output images.

The team wanted to see if better results could be derived and at the same time test the impact on an unknown dataset.

For our second implementation of the model, we followed a similar process but procured a dataset composed of 485 low-light images and video from an Apple iPhone 15 Pro. From the results of the initial experiment, we ran the training data through 40 epochs on the second iteration with new data.

To adapt the model to video, each frame of the video was enhanced in real time, frame by frame. Each frame is taken as an array then converted into a tensor and run through the model. In the video outlined in the results section, a slider is included for post-enhancement iterations. The slider is controlled manually and is used to determine the number of times the enhancement process is applied to each frame. For example, 9 indicates that the enhancement process has been applied nine times to each frame. The manual control gives the user the live control for visualization.

Model Architecture:

The provided model architecture is based on a convolutional neural network designed for low-light image enhancement. It uses 7 convolutional layers to extract and process features from the input image. The early layers capture low-level features, and deeper layers progressively refine these features. The model uses `ReLU` activation for non-linearity and employs skip connections by concatenating feature maps from earlier layers with later ones. This helps preserve spatial details while deepening the network.

In the final stage, the network outputs eight different feature maps that are combined iteratively to enhance the image through pixel-wise transformations. The use of `tanh` ensures that pixel values remain in a controlled range, making the network suitable for enhancing images in terms of brightness and contrast.

For training, a learning rate of 0.0001 ensures a gradual and stable update to the model weights, and the batch size of 4 allows the model to generalize well without consuming excessive memory.

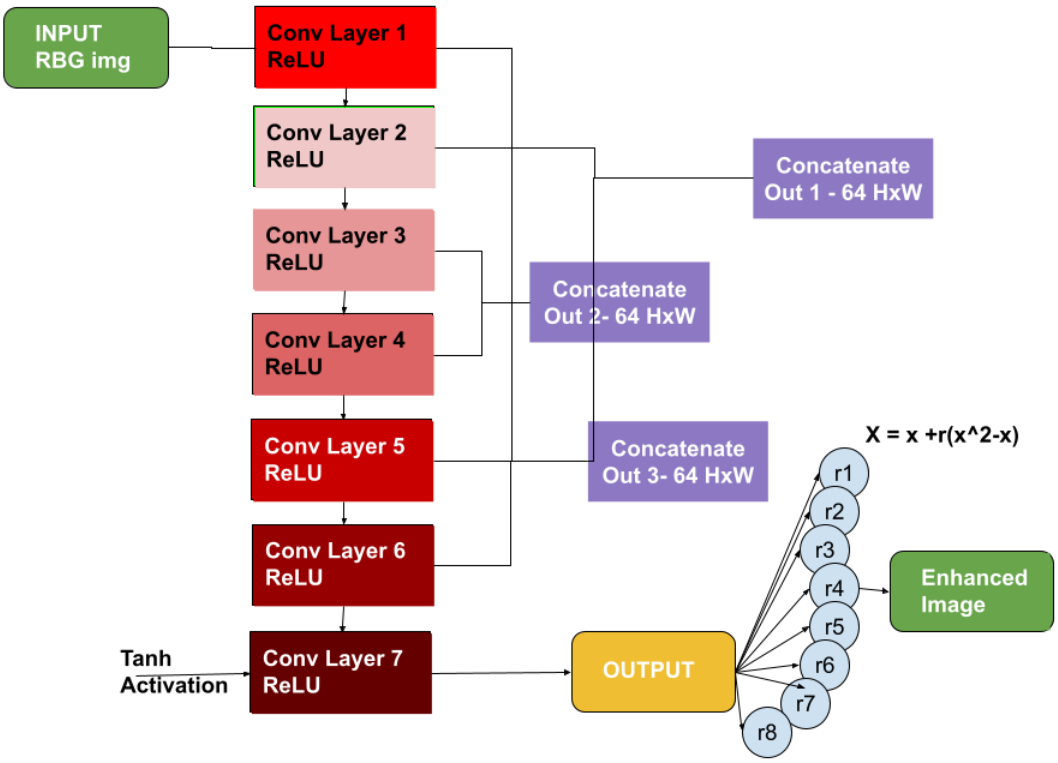


Fig. Model Architecture flowchart

Algorithm Steps

Input Low-light images:

To start, the low-light images are loaded and pre-processed for training. This step initializes the weights of the convolutional layers with a normal distribution to ensure that the network starts training with reasonable initial weights.

```
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        m.weight.data.normal_(0.0, 0.02)
```

Initialize DCE-net:

The DCE-net is initialized and loads the pre-loaded weights into the DCE-net configuration. The data is then loaded into the model from the folder path.

```
def train(config):
    os.environ['CUDA_VISIBLE_DEVICES'] = '0'
    DCE_net = model.enhance_net_nopool().cuda()
    DCE_net.apply(weights_init)
    if config.load_pretrain:
        DCE_net.load_state_dict(torch.load(config.pretrain_dir))
```

Define Custom Loss Functions:

From here, the custom loss functions that will be used during training are implemented and the optimizer is defined with specific learning rates and weight decay.

```
L_color = Myloss.L_color()
L_spa = Myloss.L_spa()
L_exp = Myloss.L_exp(16, 0.6)
L_TV = Myloss.L_TV()
optimizer = torch.optim.Adam(DCE_net.parameters(), lr=config.lr, weight_decay=conf
```

Loss Functions

L_color: Color constancy loss aims to maintain natural color balance and prevent color distortion across the R, G, B channels.

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

- Mean color value is computed for each channel across the spatial dimensions (height and width)
- Mean values are then split into separate tensors for R, G, B
- Squared differences are calculated for the mean values of each pair of color channels.
- The differences are combined to compute the final color loss.

```python
class L_color(nn.Module):
    def __init__(self):
        super(L_color, self).__init__()

    def forward(self, x):
        b, c, h, w = x.shape
        mean_rgb = torch.mean(x, [2, 3], keepdim=True)  # Compute mean over spati
        mr, mg, mb = torch.split(mean_rgb, 1, dim=1)    # Split mean values into
        Drg = torch.pow(mr - mg, 2)
        Drb = torch.pow(mr - mb, 2)
        Dgb = torch.pow(mb - mg, 2)
        k = torch.pow(torch.pow(Drg, 2) + torch.pow(Drb, 2) + torch.pow(Dgb, 2),
        return k
```

L_spa: Spatial consistency loss ensures the enhanced image preserves the spatial structure and details by preserving the difference of neighboring regions between the input image and its enhanced version.

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

- Initialization defines the convolutional kernels to compute gradients in four directions and sets these as non-trainable
- An average pooling layer with a kernel size of 4 is created. This will downsample the input and is used to reduce the spatial dimensions and focus on local mean values.
- Both the original and enhanced images are converted to grayscale and gradients for both are calculated using convolutional kernels.
- Squared differences of the gradients between the original and enhanced images are calculated.
- The differences are combined to compute the final loss.

```python
class L_spa(nn.Module):
    def __init__(self):
        super(L_spa, self).__init__()
        # Define convolution kernels for gradient computation
        kernel_left = torch.FloatTensor([[0, 0, 0], [-1, 1, 0], [0, 0, 0]]).cuda(
        kernel_right = torch.FloatTensor([[0, 0, 0], [0, 1, -1], [0, 0, 0]]).cuda
        kernel_up = torch.FloatTensor([[0, -1, 0], [0, 1, 0], [0, 0, 0]]).cuda().
        kernel_down = torch.FloatTensor([[0, 0, 0], [0, 1, 0], [0, -1, 0]]).cuda(
        # Register the kernels as parameters (not trainable)
        self.weight_left = nn.Parameter(data=kernel_left, requires_grad=False)
        self.weight_right = nn.Parameter(data=kernel_right, requires_grad=False)
        self.weight_up = nn.Parameter(data=kernel_up, requires_grad=False)
        self.weight_down = nn.Parameter(data=kernel_down, requires_grad=False)
        self.pool = nn.AvgPool2d(4)

    def forward(self, org, enhance):
        b, c, h, w = org.shape
        # Convert images to grayscale by averaging over channels
        org_mean = torch.mean(org, 1, keepdim=True)
        enhance_mean = torch.mean(enhance, 1, keepdim=True)
        # Downsample images
        org_pool = self.pool(org_mean)
        enhance_pool = self.pool(enhance_mean)
        # Compute gradient differences
        D_org_left = F.conv2d(org_pool, self.weight_left, padding=1)
        D_enhance_left = F.conv2d(enhance_pool, self.weight_left, padding=1)
        # Similarly for right, up, and down
        D_left = torch.pow(D_org_left - D_enhance_left, 2)
        # Sum up the differences
        E = D_left + D_right + D_up + D_down
        return E
```

L_exp: Exposure control loss adjusts the image to desired exposure levels by ensuring local regions have brightness to the optimal value. The distance between the average intensity value of a local region is measured against the well-exposedness level E

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

- The enhanced image is converted to grayscale and the local mean brightness is calculated using average pooling.
- The mean squared error (MSE) between the local mean brightness and target mean value are calculated.

```python
class L_exp(nn.Module):
    def __init__(self, patch_size, mean_val):
        super(L_exp, self).__init__()
        self.pool = nn.AvgPool2d(patch_size)
        self.mean_val = mean_val

    def forward(self, x):
        b, c, h, w = x.shape
        x = torch.mean(x, 1, keepdim=True)  # Convert to grayscale
        mean = self.pool(x)  # Compute local means
        # Compute the mean squared error between local means and target mean value
        d = torch.mean(torch.pow(mean - torch.FloatTensor([self.mean_val]).cuda(),
        return d
```

L_tv: Total variation loss enforces smoothness in the estimated illumination adjustment curves by penalizing abrupt changes between neighboring pixels, reducing artifacts and high-frequency noise in the enhanced image..

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

- Sets weighting factor.
- Differences between neighboring pixels in both vertical and horizontal directions are calculated.
- The differences are summed to calculate the total variation.
- The weighting factor is applied to normalize the number of elements and scales.

```python
class L_TV(nn.Module):
    def __init__(self, TVLoss_weight=1):
        super(L_TV, self).__init__()
        self.TVLoss_weight = TVLoss_weight

    def forward(self, x):
        batch_size = x.size()[0]
        h_x = x.size()[2]
        w_x = x.size()[3]
        count_h = (h_x - 1) * w_x
        count_w = h_x * (w_x - 1)
        # Compute differences between neighboring pixels
        h_tv = torch.pow((x[:, :, 1:, :] - x[:, :, :h_x - 1, :]), 2).sum()
        w_tv = torch.pow((x[:, :, :, 1:] - x[:, :, :, :w_x - 1]), 2).sum()
        # Compute total variation loss
        return self.TVLoss_weight * 2 * (h_tv / count_h + w_tv / count_w) / batch_
```

Train the Network:

A training loop is then implemented to iterate epochs and training data and performs a forward pass through the DCE-net to get an intermediate enhanced image, a final enhanced image, and the estimated adjustment curves.

```python
for epoch in range(config.num_epochs):
    for iteration, img_lowlight in enumerate(train_loader):
        img_lowlight = img_lowlight.cuda()
        enhanced_image_1, enhanced_image, A = DCE_net(img_lowlight)
```

Individual loss components are then calculated and combined to create a total loss to be used in backpropagation. Each loss component addresses a specific aspect of image quality. By combining them, the total loss guides the network to produce enhanced images that are well-exposed, color-balanced, and structurally consistent with the original images.

```python
Loss_TV = 200 * L_TV(A)
loss_spa = torch.mean(L_spa(enhanced_image, img_lowlight))
loss_col = 5 * torch.mean(L_color(enhanced_image))
loss_exp = 10 * torch.mean(L_exp(enhanced_image))
loss = Loss_TV + loss_spa + loss_col + loss_exp
```

Backpropagation and Optimization:

Using the total loss, the network parameters are updated using backpropagation and an optimization algorithm. This step adjusts the network weights to minimize the total loss, thereby improving the network's performance over time.

Backpropagation

- All gradients are cleared of optimized tensors to prevent accumulation from previous iterations.
- The gradient of total loss is computed with respect to network parameters.
- To prevent destabilizing from exploding gradients, gradients are configured to a maximum norm.

Optimization

- The optimizer updates the network parameters using the gradients computed during backpropagation.

```
optimizer.zero_grad()
loss.backward()
torch.nn.utils.clip_grad_norm(DCE_net.parameters(), config.grad_clip_norm)
optimizer.step()
```

Enhance Images via Recursive Adjustments:

Adjustment curves are output by the DCE-net. These adjustments are applied recursively to the input image to progressively enhance its illumination, leading to more stable and refined image enhancement.

Test the Network:

After training, the network is tested on new low-light images to evaluate its performance. The testing process involves loading the trained model, processing the test images, and obtaining enhanced results.

```
if __name__ == '__main__':
    with torch.no_grad():
        filePath = 'data/test_data/'
        file_list = os.listdir(filePath)
        for file_name in file_list:
            test_list = glob.glob(filePath + file_name + "/*")
            for image in test_list:
                print(image)
                lowlight(image)
```

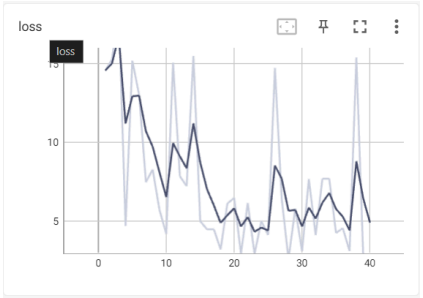
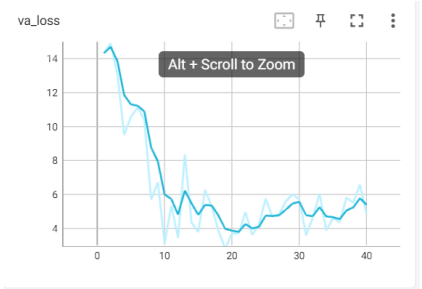
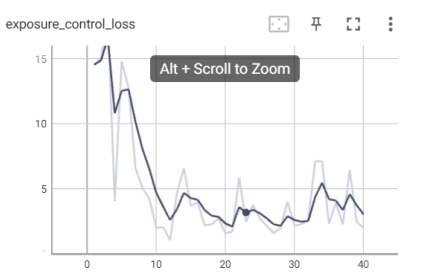
Output Enhanced Images:

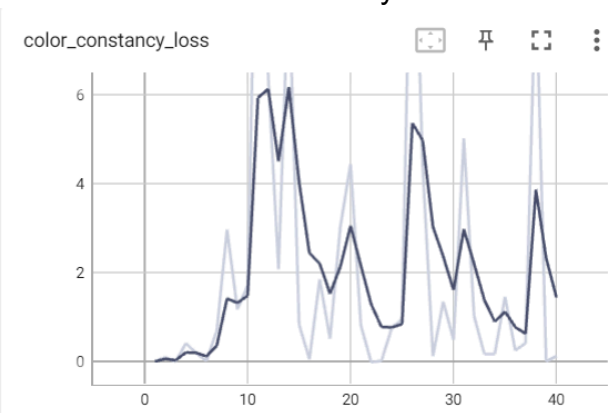
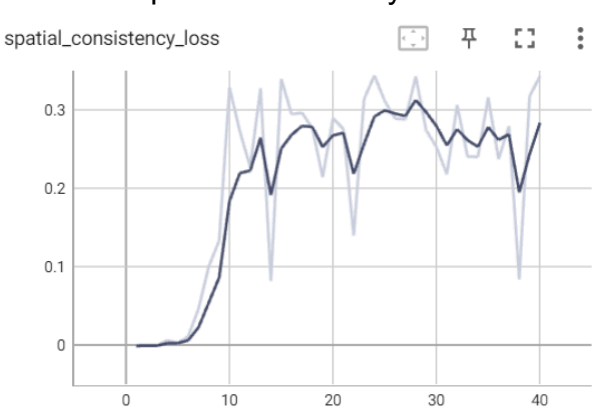
The final step involves outputting the enhanced images generated by the DCE-Net. These images should exhibit improved illumination, preserved details, and natural color balance.

Results

Looking at our loss below, general training loss decreases significantly during early epochs, showing that the model is learning effectively on training data. There is some fluctuation as the mode learns, indicating that there may be some instability. However, the downward trend in the general training loss model indicates the model is minimizing error as it learns.

The validation and exposure control loss following a similar decrease as the general training loss without exactly fitting the training data indicate that the model is learning from general features and minimizing under/overexposure of the output image.

General Training Loss	Variation Loss	Exposure Control Loss
		

Color constancy loss	Spatial consistency loss
	

Output

Visually, we can see that the enhanced image is brighter and highlights finer details in the image than the original image which looks to have been taken at dusk in sub-optimal lighting. When compared to the original set of images provided in our initial test, we can see that adjusting the parameters in this second implementation with an outside dataset produced a clearer, brighter image that did not suffer from oversaturation and color dilution like the first implementation.

Fig 1. Original Image captured at sunset

Fig 2. Image Enhanced

The Image on the Left here is the Enhanced Image vs the right original dull Image

Some more examples on the test images where we tried to see how the model performs and the results as you can observe are very evident.

Video Implementation
[Screen Recording 2024-09-29 170026.mp4]

Adapting the model to a video taken in low light proves to be successful for enhancing brightness and keeping clarity within the main subject. Using the manual adjustment, to visualize the iterations of the enhancement process to each frame, seven iterations looks to be the best balance between enhancing brightness while still keeping detail and clarity.

Reflection and acknowledgements

In reimplementing the Zero-DCE (Zero-Reference Deep Curve Estimation) model, We gained a deep understanding of how image enhancement works through learned non-linear transformations. The model architecture relies on several convolutional layers without pooling, followed by an iterative enhancement process that progressively refines the image. Each step involves applying learned adjustment curves (denoted by `r` maps) to correct brightness and contrast, a method that is lightweight but effective.

Through the process, we explored the significance of spatial consistency loss, exposure control, and color preservation loss in ensuring that the enhanced images retain natural tones without artifacts like noise or oversaturation. The spatial consistency loss, in particular, was designed to maintain the structure and content across regions of an image during enhancement, preventing distortions. We discovered that increasing spatial consistency loss during training could indicate issues with model generalization or overfitting to specific patterns, which helped me better understand the importance of carefully balancing loss components.