Image Denoising Project

By Sneha Maheshwari(22323037)

- **Zero-Reference Deep Curve Estimation** or **Zero-DCE** formulates low-light image enhancement as the task of estimating an image-specific tonal curve with a deep neural network. In this example, we train a lightweight deep network, **DCE-Net**, to estimate pixel-wise and high-order tonal curves for dynamic range adjustment of a given image.

- Zero-DCE takes a low-light image as input and produces high-order tonal curves as its output. These curves are then used for pixel-wise adjustment on the dynamic range of the input to obtain an enhanced image. The curve estimation process is done in such a way that it maintains the range of the enhanced image and preserves the contrast of neighboring pixels. This curve estimation is inspired by curves adjustment used in photo editing software such as Adobe Photoshop where users can adjust points throughout an image's tonal range.

- Zero-DCE is appealing because of its relaxed assumptions with regard to reference images: it does not require any input/output image pairs during training. This is achieved through a set of carefully formulated non-reference loss functions, which implicitly measure the enhancement quality and guide the training of the network.

Downloading LOLDataset

The LoL Dataset, utilized for enhancing low-light images, comprises 485 images for training and 15 for testing. Each pair within the dataset contains a low-light input image alongside its corresponding well-exposed reference image

```
# Download and unzip dataset
def download_and_extract(url, extract_to='.'):
    response = requests.get(url)
    with zipfile.ZipFile(io.BytesIO(response.content)) as zip_ref:
        zip_ref.extractall(extract_to)

download_and_extract('https://huggingface.co/datasets/geekyrakshit/LoL-Dataset/resolve/main/lol_dataset.zip')
```

We employ 300 low-light images from the training set of the LoL Dataset for training purposes, reserving the remaining 185 low-light images for validation. The images are resized to dimensions of 256 x 256 for both training and validation.

The Zero DCE Framework

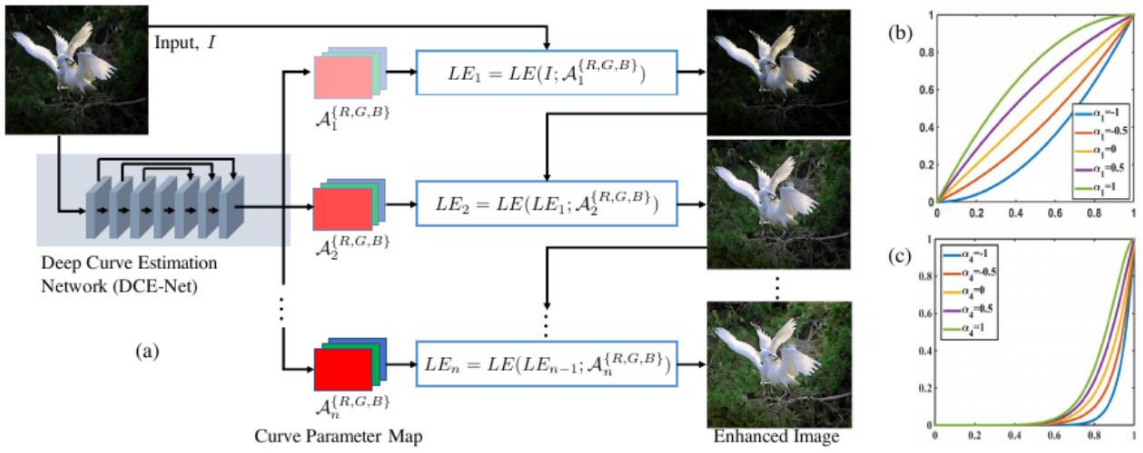
The objective of the Zero-DCE Framework is to utilize the DCE-Net to predict a series of optimal light-enhancement curves (LE-curves) based on an input image. Subsequently, the framework iteratively applies these curves to all pixels within the RGB channels of the input, yielding the final enhanced image.

Understanding light-enhancement curves

A light-enhancement curve is a kind of curve that can map a low-light image to its enhanced version automatically, where the self-adaptive curve parameters are solely dependent on the input image. When designing such a curve, three objectives should be taken into account:

- Each pixel value of the enhanced image should be in the normalized range [0,1], in order to avoid information loss induced by overflow truncation.
- It should be monotonous, to preserve the contrast between neighboring pixels.
- The shape of this curve should be as simple as possible, and the curve should be differentiable to allow backpropagation.

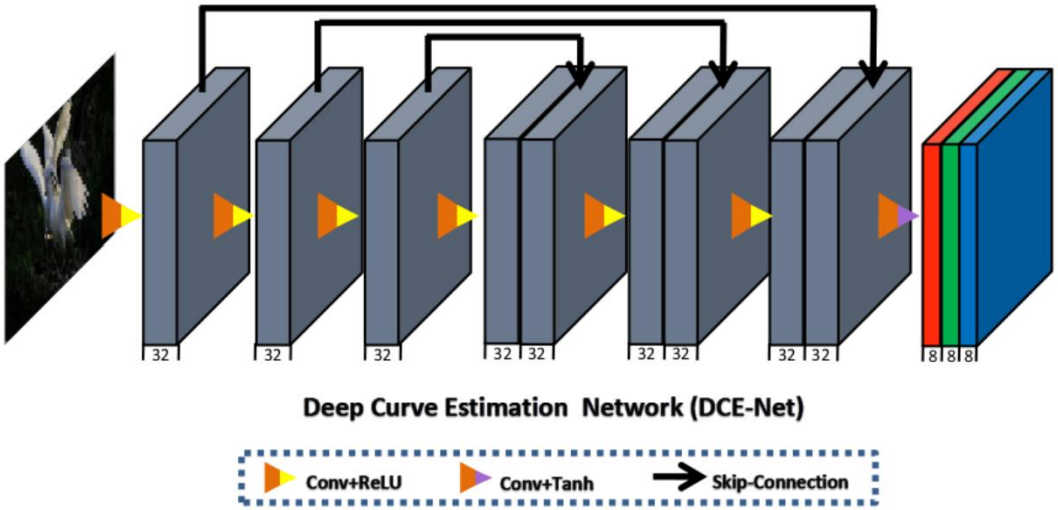
The light-enhancement curve is separately applied to three RGB channels instead of solely on the illumination channel. The three-channel adjustment can better preserve the inherent color and reduce the risk of over-saturation.



DCE-Net

The DCE-Net is a lightweight deep neural network that learns the mapping between an input image and its best-fitting curve parameter maps. The input to the DCE-Net is a

low-light image while the outputs are a set of pixel-wise curve parameter maps for corresponding higher-order curves. It is a plain CNN of seven convolutional layers with symmetrical concatenation. Each layer consists of 32 convolutional kernels of size 3×3 and stride 1 followed by the ReLU activation function. The last convolutional layer is followed by the Tanh activation function, which produces 24 parameter maps for 8 iterations, where each iteration requires three curve parameter maps for the three channels.




```python
# Build the DCE network
def build_dce_net():
    input_img = keras.Input(shape=[None, None, 3])
    conv1 = layers.Conv2D(32, (3, 3), strides=(1, 1), activation="relu", padding="same")(input_img)
    conv2 = layers.Conv2D(32, (3, 3), strides=(1, 1), activation="relu", padding="same")(conv1)
    conv3 = layers.Conv2D(32, (3, 3), strides=(1, 1), activation="relu", padding="same")(conv2)
    conv4 = layers.Conv2D(32, (3, 3), strides=(1, 1), activation="relu", padding="same")(conv3)
    int_con1 = layers.Concatenate(axis=-1)([conv4, conv3])
    conv5 = layers.Conv2D(32, (3, 3), strides=(1, 1), activation="relu", padding="same")(int_con1)
    int_con2 = layers.Concatenate(axis=-1)([conv5, conv2])
    conv6 = layers.Conv2D(32, (3, 3), strides=(1, 1), activation="relu", padding="same")(int_con2)
    int_con3 = layers.Concatenate(axis=-1)([conv6, conv1])
    x_r = layers.Conv2D(24, (3, 3), strides=(1, 1), activation="tanh", padding="same")(int_con3)
    return keras.Model(inputs=input_img, outputs=x_r)
```

Loss functions

Spatial Consistency Loss : The spatial consistency loss encourages spatial coherence of the enhanced image through preserving the difference of neighboring regions between the input image and its enhanced version:

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

where K is the number of local region, and Ω(i) is the four neighboring regions (top, down, left, right) centered at the region i. We denote Y and I as the average intensity value of the local region in the enhanced version and input image, respectively. We empirically set the size of the local region to 4×4. This loss is stable given other region sizes.

```
def spatial_constancy_loss(original_img, enhanced_img, patch_size):
        def gradient(img):
            img = tf.ensure_shape(img, [None, None, None, 3])
            img = tf.image.resize(img, [patch_size, patch_size])
            gradient_y, gradient_x = tf.image.image_gradients(img)
            return gradient_y, gradient_x

        original_gradient_y, original_gradient_x = gradient(original_img)
        enhanced_gradient_y, enhanced_gradient_x = gradient(enhanced_img)

        loss = tf.reduce_mean(tf.square(original_gradient_y - enhanced_gradient_y)) + \
               tf.reduce_mean(tf.square(original_gradient_x - enhanced_gradient_x))
        return loss
```

Exposure Control Loss. To restrain under-/over-exposed regions, we design an exposure control loss to control the exposure level. The exposure control loss measures the distance between the average intensity value of a local region to the well-exposedness level E.

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

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.

```
def exposure_loss(x, mean_val=0.6):
    x = tf.reduce_mean(x, axis=3, keepdims=True)
    mean = tf.nn.avg_pool2d(x, ksize=16, strides=16, padding="VALID")
    return tf.reduce_mean(tf.square(mean - mean_val))
```

Color Constancy Loss. Following Gray-World color constancy hypothesis [2] that color in each sensor channel averages to gray over the entire image, we design a color constancy loss to correct the potential color deviations in the enhanced image and also build the relations among the three adjusted channels. The color constancy loss Lcol can be expressed as:

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

where Jp denotes the average intensity value of p channel in the enhanced image, (p,q) represents a pair of channels.

```
def color_constancy_loss(x):
    mean_rgb = tf.reduce_mean(x, axis=(1, 2), keepdims=True)
    mean_R = tf.reduce_mean(mean_rgb[:, :, :, 0])
    mean_G = tf.reduce_mean(mean_rgb[:, :, :, 1])
    mean_B = tf.reduce_mean(mean_rgb[:, :, :, 2])

    loss_RG = tf.square(mean_R - mean_G)
    loss_RB = tf.square(mean_R - mean_B)
    loss_GB = tf.square(mean_G - mean_B)

    total_loss = loss_RG + loss_RB + loss_GB

    return total_loss
```

Illumination Smoothness Loss: To preserve the monotonicity relations between neighboring pixels, we add an illumination smoothness loss to each curve parameter map A.

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

where N is the number of iteration, ∇x and ∇y represent the horizontal and vertical gradient operations, respectively.

```python
def illumination_smoothness_loss(x):
        sobel_edges = tf.image.sobel_edges(x)
        gradient_x = sobel_edges[:, :, :, 0]
        gradient_y = sobel_edges[:, :, :, 1]
        smoothness_loss = tf.reduce_mean(tf.square(tf.abs(gradient_x) + tf.abs(gradient_y)))
        return smoothness_loss
```

Total Loss. The total loss can be expressed as:

$$L_{total} = L_{spa} + L_{exp} + W_{col}L_{col} + W_{tv_{\mathcal{A}}}L_{tv_{\mathcal{A}}},$$

Deep curve estimation model

Model Architecture:

- The ZeroDCE class extends the keras.Model class and encapsulates the DCE-Net model.
- It initializes the DCE-Net model (self.dce_model) using the build_dce_net() function.

compile: Sets up the optimizer and various custom metrics for tracking during training.

- Uses Adam optimizer.
- Tracks total loss, illumination smoothness loss, spatial constancy loss, color constancy loss, exposure loss, and PSNR.

```python
class ZeroDCE(keras.Model):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.dce_model = build_dce_net()

    def compile(self, learning_rate, **kwargs):
        super().compile(**kwargs)
        self.optimizer = keras.optimizers.Adam(learning_rate=learning_rate)
        self.total_loss_tracker = keras.metrics.Mean(name="total_loss")
        self.illumination_smoothness_loss_tracker = keras.metrics.Mean(
            name="illumination_smoothness_loss"
        )
        self.spatial_constancy_loss_tracker = keras.metrics.Mean(
            name="spatial_constancy_loss"
        )
        self.color_constancy_loss_tracker = keras.metrics.Mean(
            name="color_constancy_loss"
        )
        self.exposure_loss_tracker = keras.metrics.Mean(name="exposure_loss")
        self.psnr_tracker = keras.metrics.Mean(name="psnr")
```

Custom Metrics:

- The metrics property returns a list of custom metrics for tracking losses during training.

```python
@property
def metrics(self):
    return [
        self.total_loss_tracker,
        self.illumination_smoothness_loss_tracker,
        self.spatial_constancy_loss_tracker,
        self.color_constancy_loss_tracker,
        self.exposure_loss_tracker,
        self.psnr_tracker,
    ]
```

Enhanced Image Generation:

The get_enhanced_image method takes an input image and a set of enhancement curve coefficients (generated by the DCE network) and applies a series of transformations to enhance the image. Each stage of the transformation involves using the coefficients to adjust the image based on a specific formula, iteratively refining the image quality. The process ensures that the enhancement is smooth and progressive, leading to a final image with improved visual quality.

```
def get_enhanced_image(self, data, output):
    r1 = output[:, :, :, :3]
    r2 = output[:, :, :, 3:6]
    r3 = output[:, :, :, 6:9]
    r4 = output[:, :, :, 9:12]
    r5 = output[:, :, :, 12:15]
    r6 = output[:, :, :, 15:18]
    r7 = output[:, :, :, 18:21]
    r8 = output[:, :, :, 21:24]
    x = data + r1 * (tf.square(data) - data)
    x = x + r2 * (tf.square(x) - x)
    x = x + r3 * (tf.square(x) - x)
    enhanced_image = x + r4 * (tf.square(x) - x)
    x = enhanced_image + r5 * (tf.square(enhanced_image) - enhanced_image)
    x = x + r6 * (tf.square(x) - x)
    x = x + r7 * (tf.square(x) - x)
    enhanced_image = x + r8 * (tf.square(x) - x)
    return enhanced_image
```

Forward Pass:

- The call() method performs a forward pass through the DCE-Net model and then applies the obtained enhancement to the input image.

```
def call(self, data):
    dce_net_output = self.dce_model(data)
    return self.get_enhanced_image(data, dce_net_output)
```

Loss Computation:

The compute_losses function calculates the total loss and its individual components for the image enhancement model. The losses ensure that the enhanced images have smooth illumination, maintain spatial and color consistency, and have proper exposure. Each loss is scaled appropriately to contribute to the total loss, guiding the training process to produce high-quality

enhanced images.

```
def compute_losses(self, data, output):
    enhanced_image = self.get_enhanced_image(data, output)
    loss_illumination = 200 * illumination_smoothness_loss(output)
    loss_spatial_constancy = tf.reduce_mean(spatial_constancy_loss(data, enhanced_image, patch_size=32))
    loss_color_constancy = 5 * tf.reduce_mean(color_constancy_loss(enhanced_image))
    loss_exposure = 10 * tf.reduce_mean(exposure_loss(enhanced_image))
    total_loss = (
        loss_illumination
        + loss_spatial_constancy
        + loss_color_constancy
        + loss_exposure
    )

    return {
        "total_loss": total_loss,
        "illumination_smoothness_loss": loss_illumination,
        "spatial_constancy_loss": loss_spatial_constancy,
        "color_constancy_loss": loss_color_constancy,
        "exposure_loss": loss_exposure,
    }
```

Training Step:

The train_step function performs a single iteration of model training:

1. It extracts the input and ground truth images.
2. Computes the enhancement output and corresponding losses using the GradientTape context.
3. Calculates the gradients of the total loss with respect to the model's trainable weights.
4. Updates the model's weights using the optimizer.
5. Updates various loss and performance metrics.
6. Returns the current values of the tracked metrics.

```python
def train_step(self, data):
    low_light_images, high_light_images = data
    with tf.GradientTape() as tape:
        output = self.dce_model(low_light_images)
        losses = self.compute_losses(low_light_images, output)
        enhanced_image = self.get_enhanced_image(low_light_images, output)

    gradients = tape.gradient(
        losses["total_loss"], self.dce_model.trainable_weights
    )
    self.optimizer.apply_gradients(zip(gradients, self.dce_model.trainable_weights))

    self.total_loss_tracker.update_state(losses["total_loss"])
    self.illumination_smoothness_loss_tracker.update_state(
        losses["illumination_smoothness_loss"]
    )
    self.spatial_constancy_loss_tracker.update_state(
        losses["spatial_constancy_loss"]
    )
    self.color_constancy_loss_tracker.update_state(losses["color_constancy_loss"])
    self.exposure_loss_tracker.update_state(losses["exposure_loss"])
    self.psnr_tracker.update_state(self.psnr(high_light_images, enhanced_image))
```

Testing Step:

The test_step function performs a single evaluation step on a batch of test data:

1. Extracts the input and ground truth images.
2. Computes the enhancement output and corresponding losses.
3. Updates the state of various loss and performance metrics, including total loss, illumination smoothness loss, spatial constancy loss, color constancy loss, exposure loss, and PSNR.
4. Returns the current values of the tracked metrics.

```
def test_step(self, data):
    low_light_images, high_light_images = data
    output = self.dce_model(low_light_images)
    losses = self.compute_losses(low_light_images, output)
    enhanced_image = self.get_enhanced_image(low_light_images, output)

    self.total_loss_tracker.update_state(losses["total_loss"])
    self.illumination_smoothness_loss_tracker.update_state(
        losses["illumination_smoothness_loss"]
    )
    self.spatial_constancy_loss_tracker.update_state(
        losses["spatial_constancy_loss"]
    )
    self.color_constancy_loss_tracker.update_state(losses["color_constancy_loss"])
    self.exposure_loss_tracker.update_state(losses["exposure_loss"])
    self.psnr_tracker.update_state(self.psnr(high_light_images, enhanced_image))

    return {metric.name: metric.result() for metric in self.metrics}
```

Model Weight Saving and Loading:

- The save_weights() and load_weights() methods facilitate saving and loading the weights of the DCE-Net model, respectively.

```
def save_weights(self, filepath, overwrite=True, save_format=None, options=None):
    """While saving the weights, we simply save the weights of the DCE-Net"""
    self.dce_model.save_weights(
        filepath,
        overwrite=overwrite,
        save_format=save_format,
        options=options,
    )

def load_weights(self, filepath, by_name=False, skip_mismatch=False, options=None):
    """While loading the weights, we simply load the weights of the DCE-Net"""
    self.dce_model.load_weights(
        filepath=filepath,
        by_name=by_name,
        skip_mismatch=skip_mismatch,
        options=options,
    )
```

Training

The model is trained with a learning rate of 1e-4 and epochs=100

plot_result aids in visualizing the training progress by plotting specified metrics (e.g., loss) over epochs, providing insights into the model's performance and potential overfitting.

```
zero_dce_model = ZeroDCE()
zero_dce_model.compile(learning_rate=1e-4)
history = zero_dce_model.fit(train_dataset, validation_data=val_dataset, epochs=100)


def plot_result(item):
    plt.plot(history.history[item], label=item)
    plt.plot(history.history["val_" + item], label="val_" + item)
    plt.xlabel("Epochs")
    plt.ylabel(item)
    plt.title("Train and Validation {} Over Epochs".format(item), fontsize=14)
    plt.legend()
    plt.grid()
    plt.show()


plot_result("total_loss")
plot_result("illumination_smoothness_loss")
plot_result("spatial_constancy_loss")
plot_result("color_constancy_loss")
plot_result("exposure_loss")
```

plot_results aids in visualizing the results of inference or any other image-related analysis, allowing for side-by-side comparison of images with associated titles.

infer facilitates the application of the trained Zero-DCE model to a single input image, providing an enhanced version as output, which can be further analyzed or compared.

```
def plot_results(images, titles, figure_size=(12, 12)):
    fig = plt.figure(figsize=figure_size)
    for i in range(len(images)):
        fig.add_subplot(1, len(images), i + 1).set_title(titles[i])
        _ = plt.imshow(images[i])
        plt.axis("off")
    plt.show()

def infer(original_image):
    image = keras.utils.img_to_array(original_image)
    image = image.astype("float32") / 255.0
    image = np.expand_dims(image, axis=0)
    output_image = zero_dce_model(image)
    output_image = tf.cast((output_image[0, :, :, :] * 255), dtype=np.uint8)
    output_image = Image.fromarray(output_image.numpy())
    return output_image
```

After training, The model is tested. It takes the images from test/low directory and saves the enhanced images in test/predicted directory.

```
base_directory = os.getcwd()
print("Base Directory:", base_directory)

# Define the paths relative to the base directory
test_directory = os.path.join(base_directory, "test/low")
predicted_directory = os.path.join(base_directory, "test/predicted")
original_directory = os.path.join(base_directory, "test/original")

for val_image_file in os.listdir(test_directory):
    val_image_path = os.path.join(test_directory, val_image_file)
    if os.path.isfile(val_image_path):
        original_image = Image.open(val_image_path)
        enhanced_image = infer(original_image)
        base_name = os.path.basename(val_image_file)
        save_path = os.path.join(predicted_directory, base_name)
        enhanced_image.save(save_path)
        plot_results(
            [original_image, enhanced_image],
            ["Original", "Enhanced"],
            (20, 12),
        )
```

PSNR SCORE: 55.92

Summary of the Image Denoising Project Using Zero-DCE

Overview

The Image Denoising project leverages Zero-Reference Deep Curve Estimation (Zero-DCE) for enhancing low-light images. Zero-DCE employs a deep neural network, DCE-Net, to estimate high-order tonal curves that adjust the dynamic range of images, enhancing their visibility and preserving contrast without requiring reference images.

Key Components

1. **Zero-DCE Framework**:
    - **Objective**: Enhance low-light images by predicting optimal light-enhancement curves (LE-curves) using DCE-Net.
    - **Process**: Apply these curves iteratively to all pixels in the image to produce an enhanced version.
    - **Inspiration**: Similar to curves adjustment in photo editing software like Adobe Photoshop.
2. **DCE-Net**:
    - A lightweight neural network designed to map input images to their best-fitting curve parameters.
    - Comprises seven convolutional layers with symmetrical concatenation.
    - Uses 32 convolutional kernels per layer and ReLU activation functions, with the final layer using Tanh to produce parameter maps for iterative enhancement.
3. **Loss Functions**:
    - **Spatial Consistency Loss**: Maintains spatial coherence by preserving the intensity differences between neighboring regions.
    - **Exposure Control Loss**: Controls the exposure level to avoid under- or over-exposed regions.
    - **Color Constancy Loss**: Ensures color balance based on the Gray-World hypothesis.
    - **Illumination Smoothness Loss**: Maintains smoothness in illumination changes to preserve pixel contrast.
4. **Model Training and Evaluation**:
    - **Training Dataset**: Uses the LoL Dataset, which includes 485 images for training and 15 for testing.
    - **Training Process**: Utilizes a learning rate of 1e-4 and trains for 100 epochs.
    - **Evaluation Metrics**: Includes plotting results and calculating PSNR (Peak Signal-to-Noise Ratio) to assess image quality.

5. **Model Application**:
    - **Inference**: Applies the trained model to enhance low-light images, saving results in the specified directories.
    - **Testing**: Evaluates the model using images from the test set, comparing enhanced images with original low-light images.

Implementation Details

- **Model Architecture**: The ZeroDCE class encapsulates DCE-Net and includes methods for model compilation, loss computation, and training steps.
- **Training Logic**: Defined in the train_step method, it includes forward pass, loss calculation, gradient computation, and optimizer updates.
- **Enhanced Image Generation**: Achieved through the get_enhanced_image method, applying the enhancement curves iteratively.

Conclusion

The project demonstrates an effective approach to low-light image enhancement using Zero-DCE, a framework that estimates image-specific tonal curves without requiring reference images. The use of DCE-Net, combined with carefully formulated loss functions, allows for efficient and high-quality enhancement, making it a valuable tool for image processing tasks.

Link to the research paper: https://arxiv.org/pdf/2001.06826