Image Denoising Project

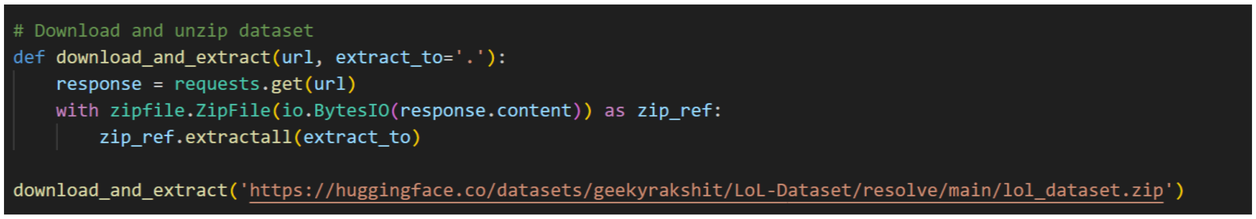
- **Zero-Reference Deep Curve Estimation** or **Zero-DCE** formulates low-light image enhancement as the task of estimating an image-specific tonal curve with a deep neural network. In this example, we train a lightweight deep network, **DCE-Net**, to estimate pixel-wise and high-order tonal curves for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

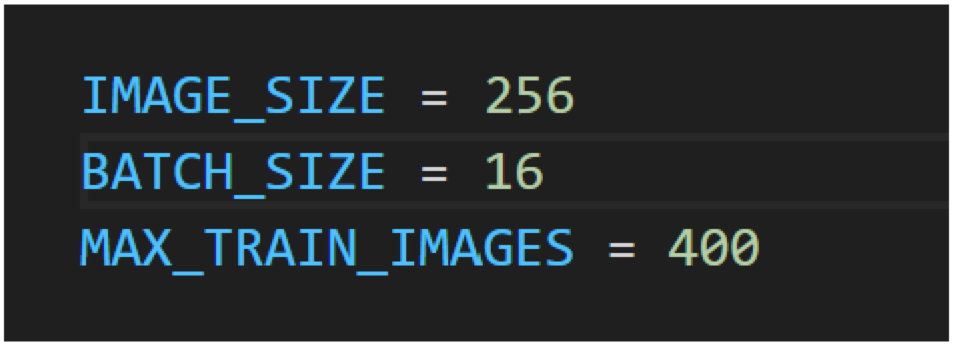
Downloading LOLDataset

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

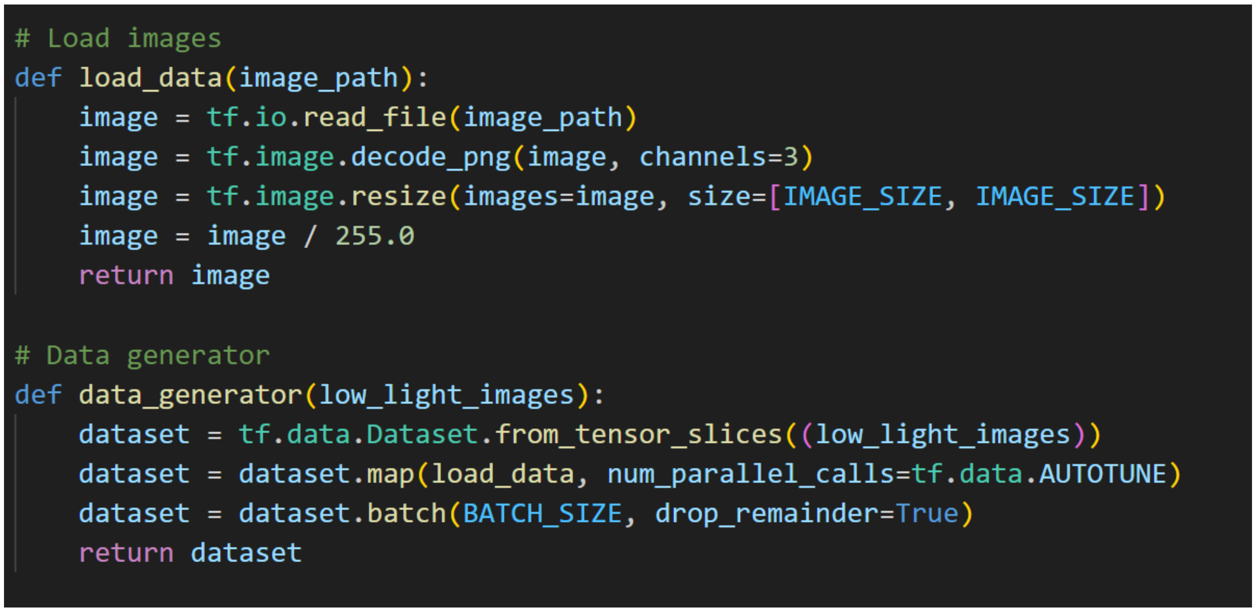


```
# Download and unzip dataset
def download_and_extract(url, extract_to='.'):
    response = requests.get(url)
    with zipfile.ZipFile(io.BytesIO(response.content)) as zip_ref:
        zip_ref.extractall(extract_to)

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



```
IMAGE_SIZE = 256
BATCH_SIZE = 16
MAX_TRAIN_IMAGES = 400
```



```
# Load images
def load_data(image_path):
    image = tf.io.read_file(image_path)
    image = tf.image.decode_png(image, channels=3)
    image = tf.image.resize(images=image, size=[IMAGE_SIZE, IMAGE_SIZE])
    image = image / 255.0
    return image

# Data generator
def data_generator(low_light_images):
    dataset = tf.data.Dataset.from_tensor_slices((low_light_images))
    dataset = dataset.map(load_data, num_parallel_calls=tf.data.AUTOTUNE)
    dataset = dataset.batch(BATCH_SIZE, drop_remainder=True)
    return dataset
```

The Zero DCE Framework

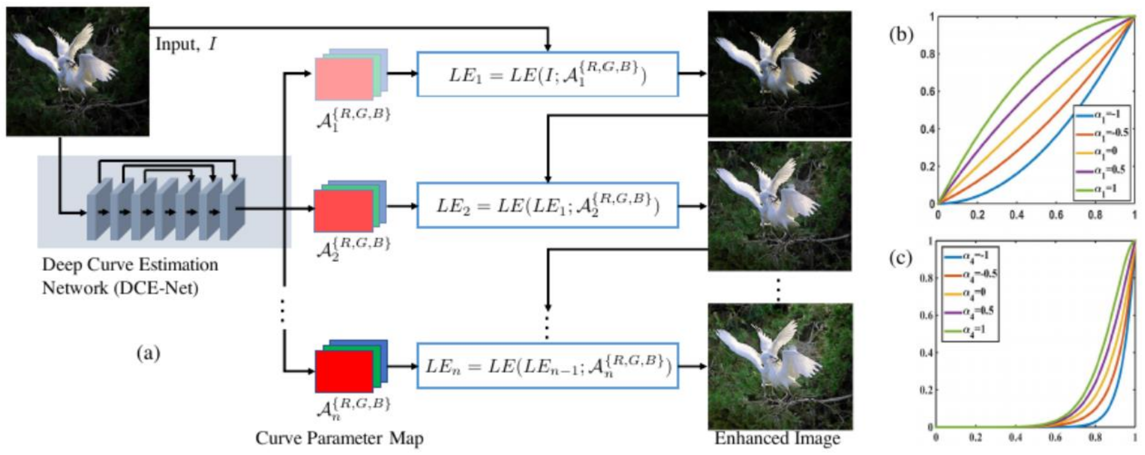
The objective of the Zero-DCE Framework is to utilize the DCE-Net to predict a series of optimal light-enhancement curves (LE-curves) based on an input image. Subsequently, the framework iteratively applies these curves to all pixels within the RGB channels of the input, yielding the final enhanced image.

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:

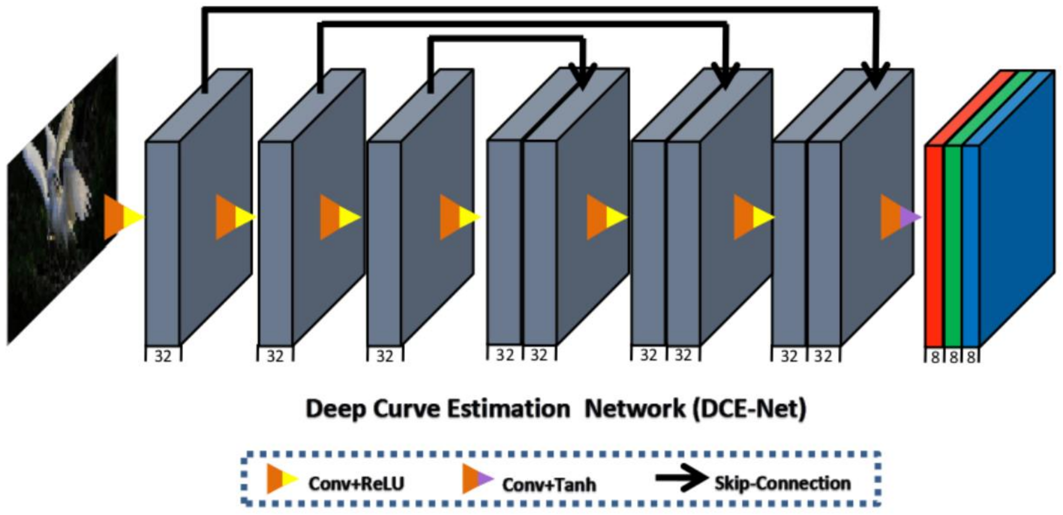
- Each pixel value of the enhanced image should be in the normalized range [0,1], in order to avoid information loss induced by overflow truncation.
- It should be monotonous, to preserve the contrast between neighboring pixels.
- The shape of this curve should be as simple as possible, and the curve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

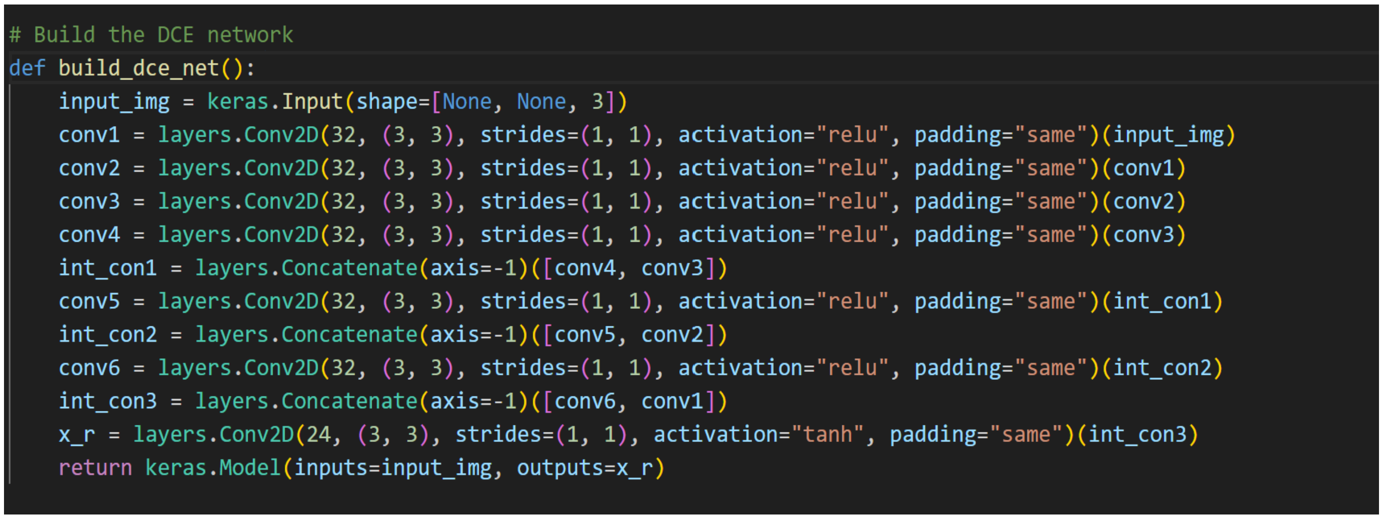
The light-enhancement curve is separately applied to three RGB channels instead of solely on the illumination channel. The three-channel adjustment can better preserve the inherent color and reduce the risk of over-saturation.


Input, I
$LE_1 = LE(I; \mathcal{A}_1^{\{R,G,B\}})$
$\mathcal{A}_1^{\{R,G,B\}}$
$LE_2 = LE(LE_1; \mathcal{A}_2^{\{R,G,B\}})$
$\mathcal{A}_2^{\{R,G,B\}}$
$LE_n = LE(LE_{n-1}; \mathcal{A}_n^{\{R,G,B\}})$
$\mathcal{A}_n^{\{R,G,B\}}$
Deep Curve Estimation Network (DCE-Net)
(a)
Curve Parameter Map
Enhanced Image
(b)
$\alpha_1=-1$
$\alpha_1=-0.5$
$\alpha_1=0$
$\alpha_1=0.5$
$\alpha_1=1$
(c)
$\alpha_4=-1$
$\alpha_4=-0.5$
$\alpha_4=0$
$\alpha_4=0.5$
$\alpha_4=1$

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.





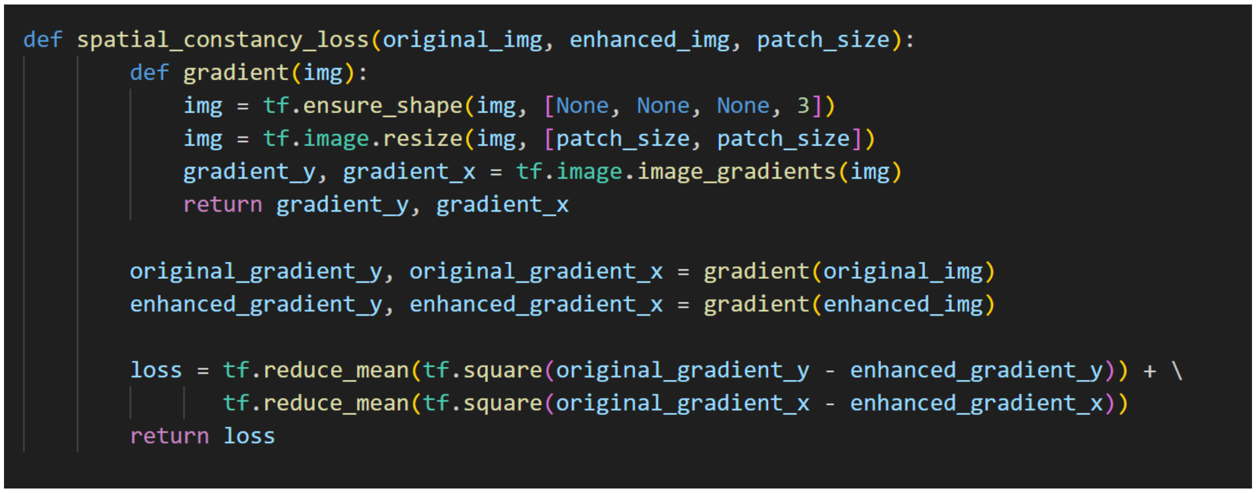
```
# Build the DCE network
def build_dce_net():
    input_img = keras.Input(shape=[None, None, 3])
    conv1 = layers.Conv2D(32, (3, 3), strides=(1, 1), activation="relu", padding="same")(input_img)
    conv2 = layers.Conv2D(32, (3, 3), strides=(1, 1), activation="relu", padding="same")(conv1)
    conv3 = 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:

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

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.

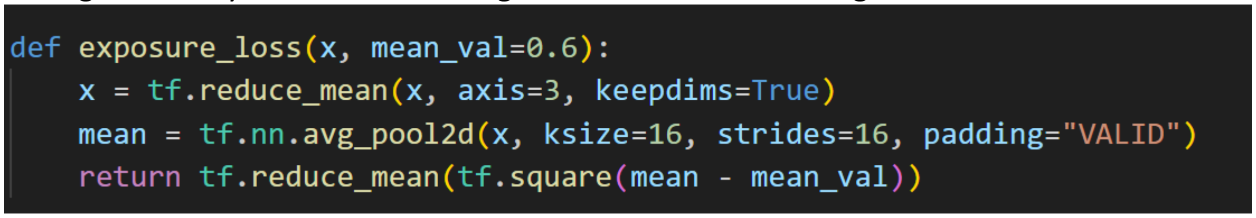


```
def spatial_constancy_loss(original_img, enhanced_img, patch_size):
        def gradient(img):
            img = tf.ensure_shape(img, [None, None, None, 3])
            img = tf.image.resize(img, [patch_size, patch_size])
            gradient_y, gradient_x = tf.image.image_gradients(img)
       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

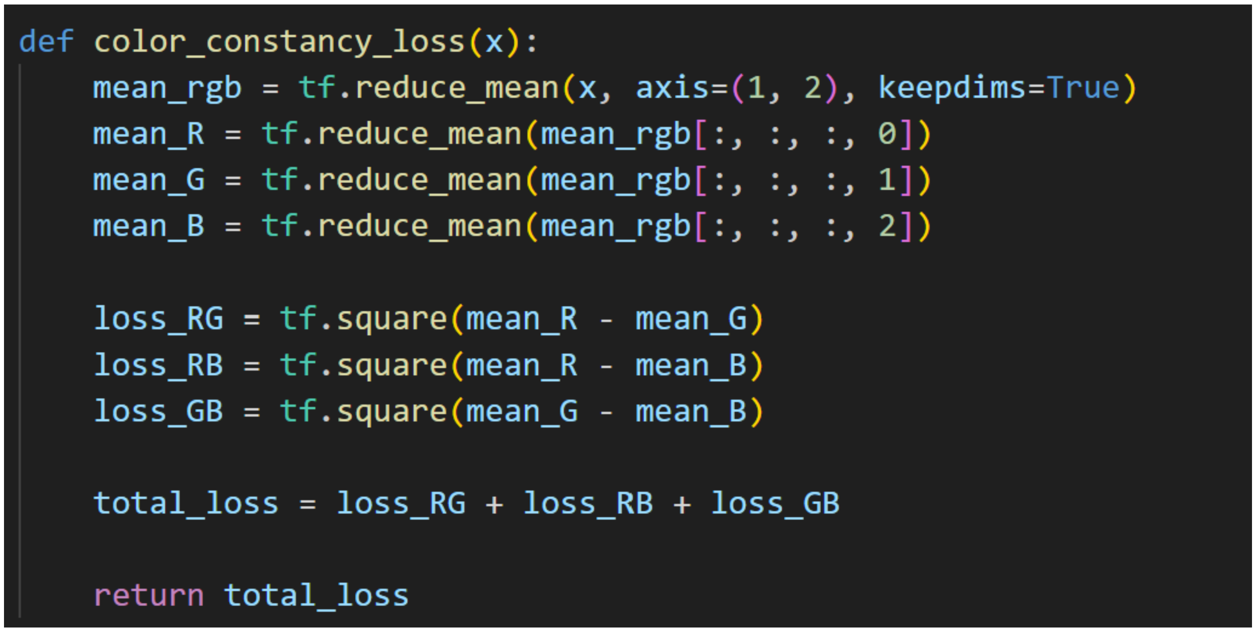


```
def exposure_loss(x, mean_val=0.6):
    x = tf.reduce_mean(x, axis=3, keepdims=True)
    mean = tf.nn.avg_pool2d(x, ksize=16, strides=16, padding="VALID")
    return tf.reduce_mean(tf.square(mean - mean_val))
```

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:

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

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

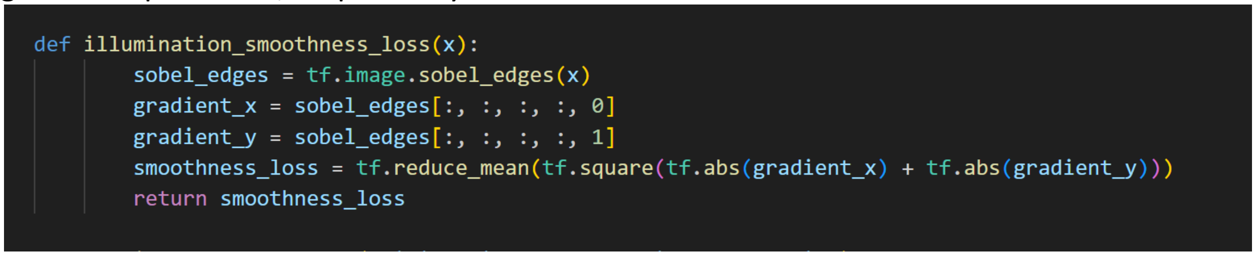


```python
def color_constancy_loss(x):
    mean_rgb = tf.reduce_mean(x, axis=(1, 2), keepdims=True)
    mean_R = tf.reduce_mean(mean_rgb[:, :, :, 0])
    mean_G = tf.reduce_mean(mean_rgb[:, :, :, 1])
    mean_B = tf.reduce_mean(mean_rgb[:, :, :, 2])

    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
```

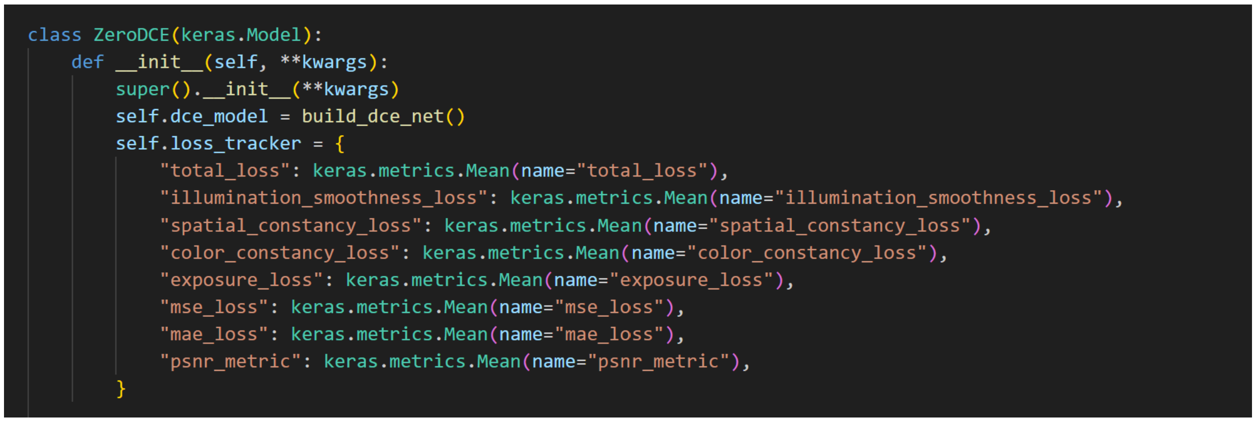
Total Loss. The total loss can be expressed as:

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

Deep curve estimation model

Model Architecture:

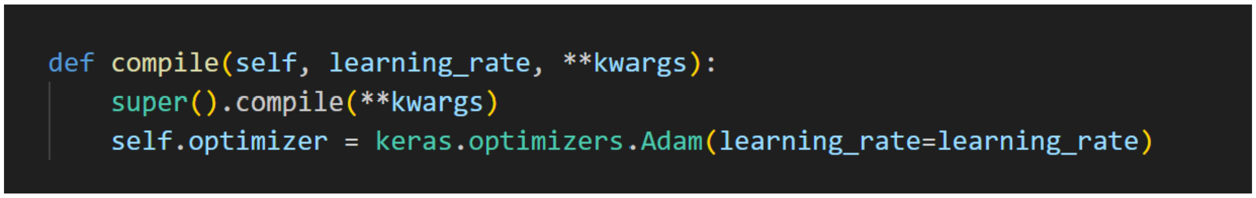
- The ZeroDCE class extends the keras.Model class and encapsulates the DCE-Net model.
- It initializes the DCE-Net model (self.dce_model) using the build_dce_net() function.
- It defines a set of loss trackers for monitoring various losses during training.



```
class ZeroDCE(keras.Model):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.dce_model = build_dce_net()
        self.loss_tracker = {
            "total_loss": keras.metrics.Mean(name="total_loss"),
            "illumination_smoothness_loss": keras.metrics.Mean(name="illumination_smoothness_loss"),
            "spatial_constancy_loss": keras.metrics.Mean(name="spatial_constancy_loss"),
            "color_constancy_loss": keras.metrics.Mean(name="color_constancy_loss"),
            "exposure_loss": keras.metrics.Mean(name="exposure_loss"),
            "mse_loss": keras.metrics.Mean(name="mse_loss"),
            "mae_loss": keras.metrics.Mean(name="mae_loss"),
            "psnr_metric": keras.metrics.Mean(name="psnr_metric"),
        }
```

Model Compilation:

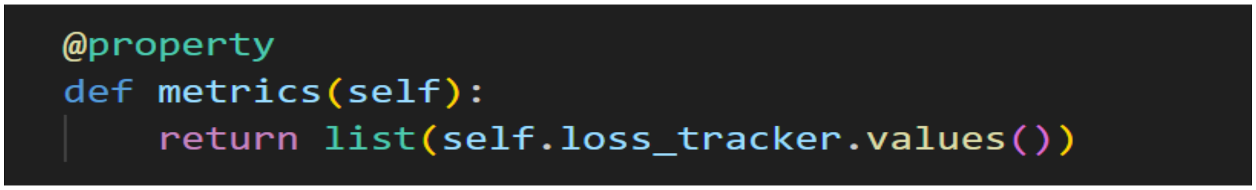
- The compile() method configures the model for training by specifying the optimizer (Adam optimizer with a given learning rate).



```
def compile(self, learning_rate, **kwargs):
    super().compile(**kwargs)
    self.optimizer = keras.optimizers.Adam(learning_rate=learning_rate)
```

Custom Metrics:

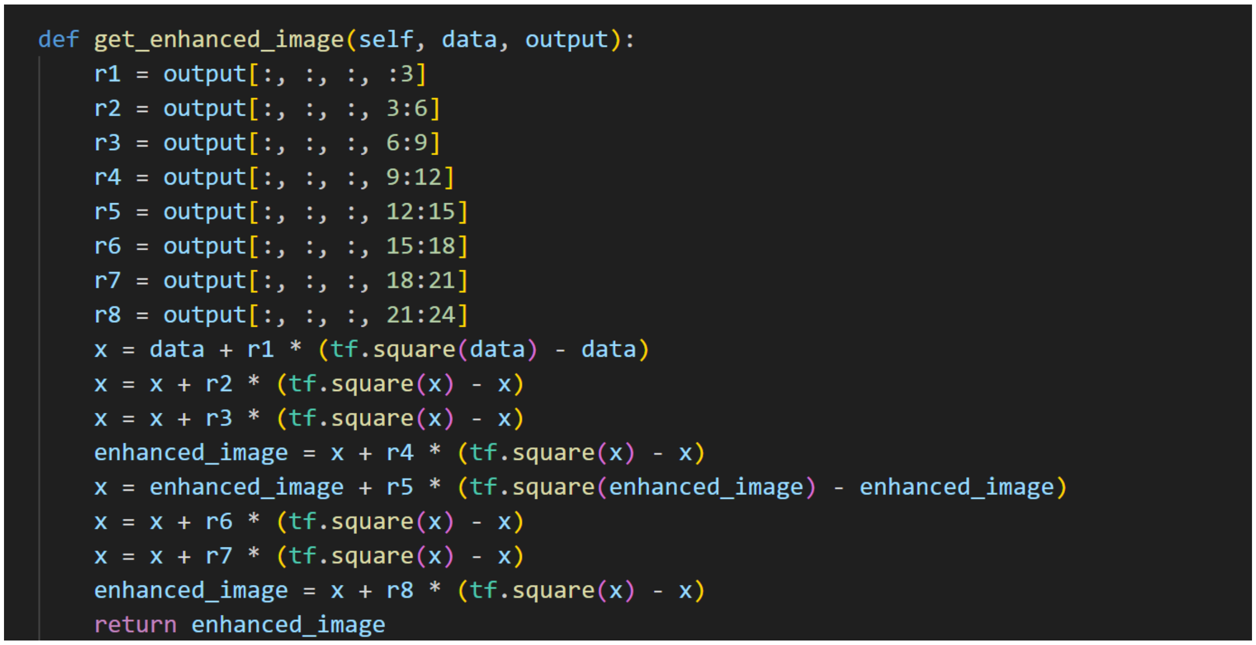
- The metrics property returns a list of custom metrics for tracking losses during training.



```python
@property
def metrics(self):
    return list(self.loss_tracker.values())
```

Enhanced Image Generation:

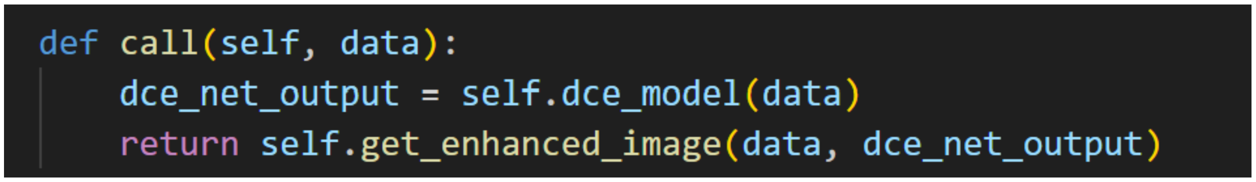
- The get_enhanced_image() method applies the estimated light-enhancement curves iteratively to the input image, resulting in the enhanced image.



```python
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:

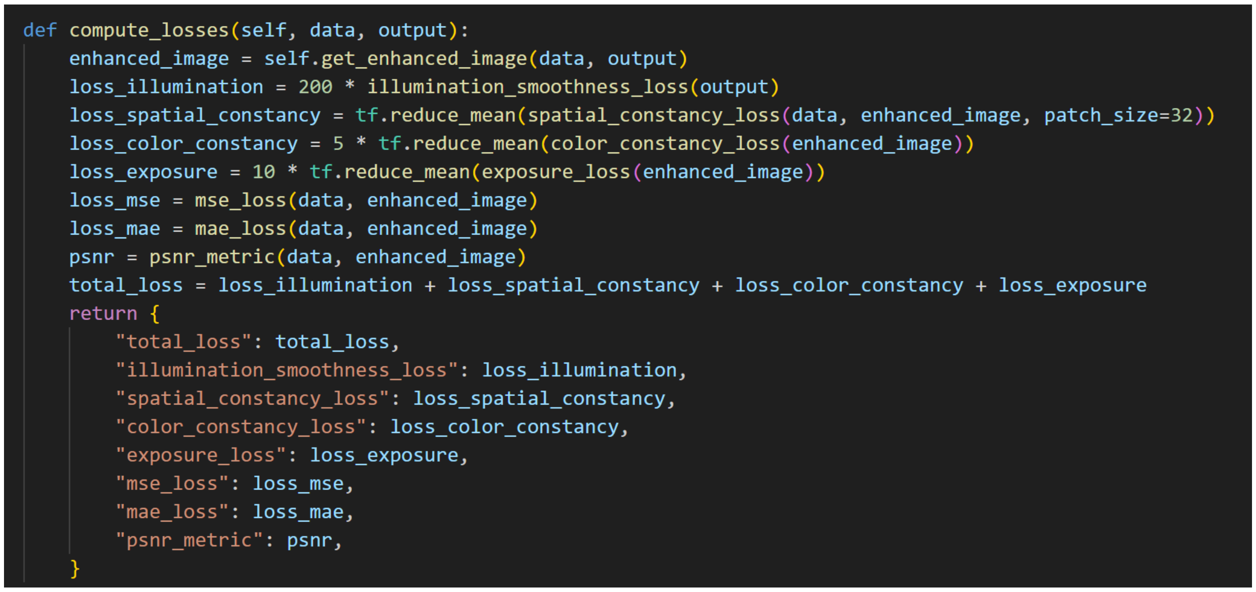
- The call() method performs a forward pass through the DCE-Net model and then applies the obtained enhancement to the input image.



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

Loss Computation:

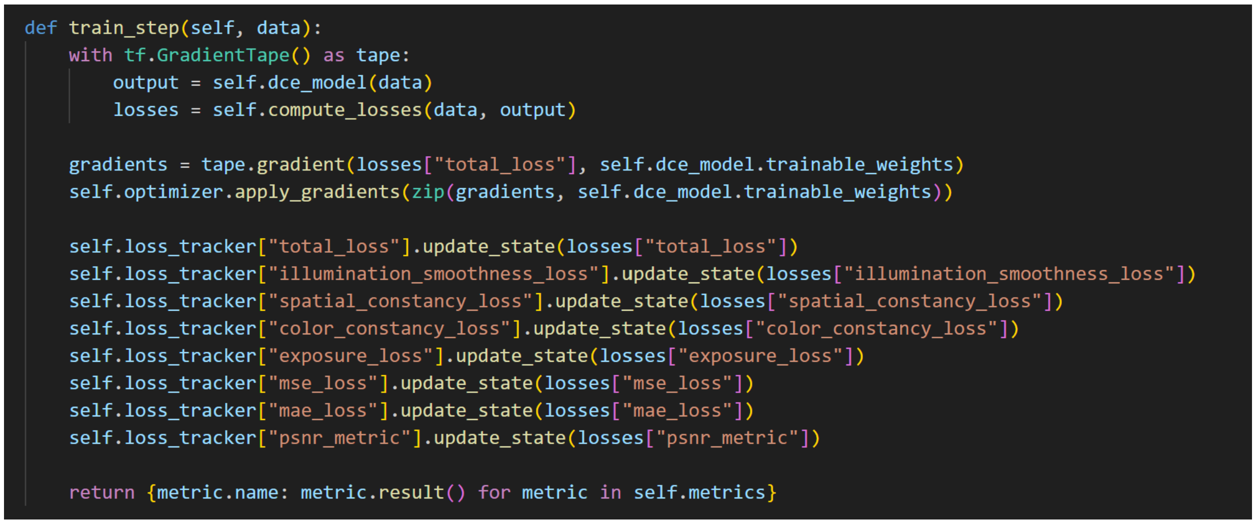
- The compute_losses() method calculates various loss components including illumination smoothness, spatial constancy, color constancy, and exposure loss.



```python
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))
    loss_mse = mse_loss(data, enhanced_image)
    loss_mae = mae_loss(data, enhanced_image)
    psnr = psnr_metric(data, 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,
        "mse_loss": loss_mse,
        "mae_loss": loss_mae,
        "psnr_metric": psnr,
    }
```

Training Step:

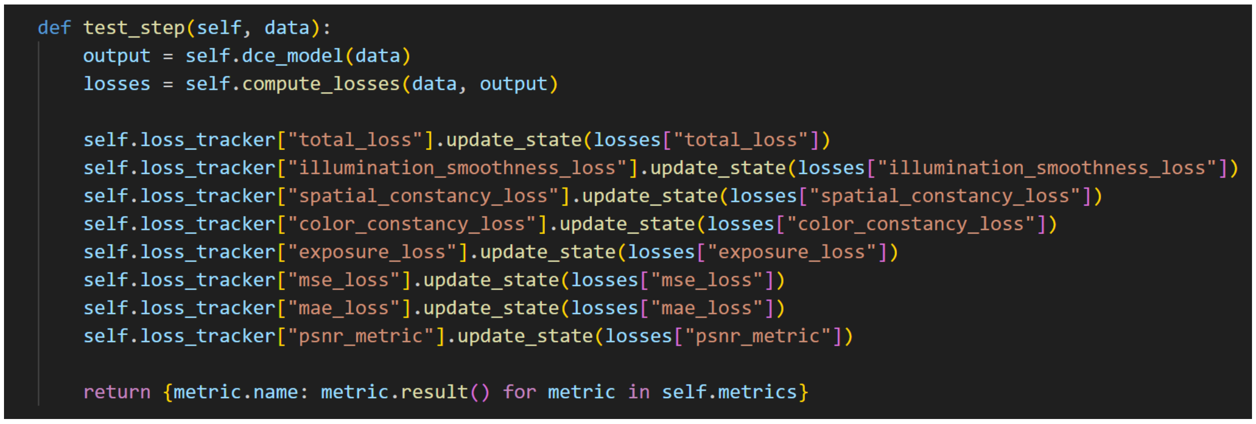
- The train_step() method defines the training logic, including forward pass, loss computation, gradient calculation, and optimizer update.



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

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

    self.loss_tracker["total_loss"].update_state(losses["total_loss"])
    self.loss_tracker["illumination_smoothness_loss"].update_state(losses["illumination_smoothness_loss"])
    self.loss_tracker["spatial_constancy_loss"].update_state(losses["spatial_constancy_loss"])
    self.loss_tracker["color_constancy_loss"].update_state(losses["color_constancy_loss"])
    self.loss_tracker["exposure_loss"].update_state(losses["exposure_loss"])
    self.loss_tracker["mse_loss"].update_state(losses["mse_loss"])
    self.loss_tracker["mae_loss"].update_state(losses["mae_loss"])
    self.loss_tracker["psnr_metric"].update_state(losses["psnr_metric"])

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

Testing Step:

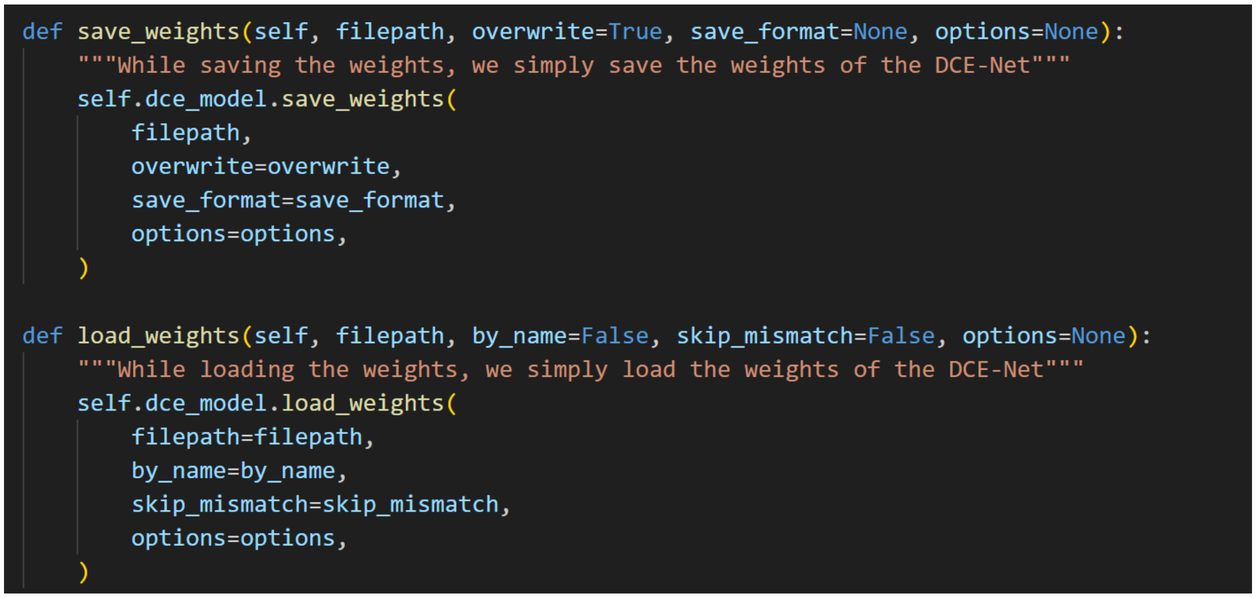
- The test_step() method specifies the testing logic, similar to the training step but without gradient computation.



```python
def test_step(self, data):
    output = self.dce_model(data)
    losses = self.compute_losses(data, output)

    self.loss_tracker["total_loss"].update_state(losses["total_loss"])
    self.loss_tracker["illumination_smoothness_loss"].update_state(losses["illumination_smoothness_loss"])
    self.loss_tracker["spatial_constancy_loss"].update_state(losses["spatial_constancy_loss"])
    self.loss_tracker["color_constancy_loss"].update_state(losses["color_constancy_loss"])
    self.loss_tracker["exposure_loss"].update_state(losses["exposure_loss"])
    self.loss_tracker["mse_loss"].update_state(losses["mse_loss"])
    self.loss_tracker["mae_loss"].update_state(losses["mae_loss"])
    self.loss_tracker["psnr_metric"].update_state(losses["psnr_metric"])

    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.

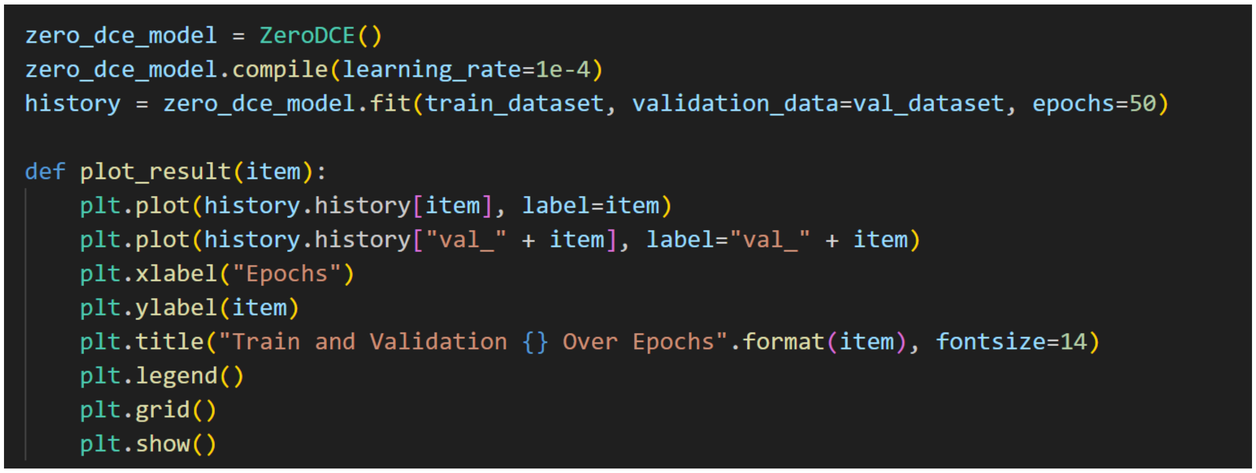


```python
def save_weights(self, filepath, overwrite=True, save_format=None, options=None):
    """While saving the weights, we simply save the weights of the DCE-Net"""
    self.dce_model.save_weights(
        filepath,
        overwrite=overwrite,
        save_format=save_format,
        options=options,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

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=50

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.

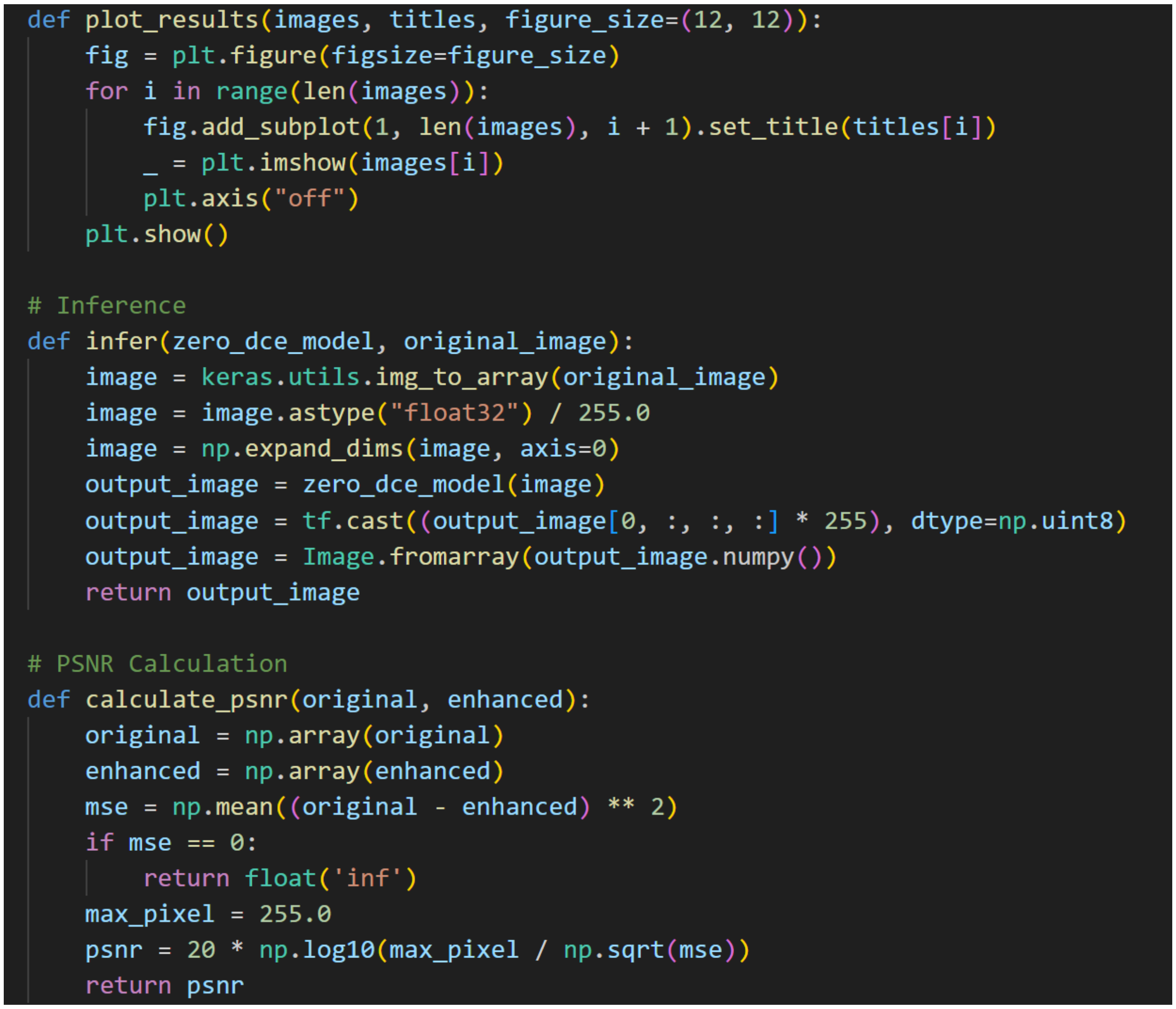


```
zero_dce_model = ZeroDCE()
zero_dce_model.compile(learning_rate=1e-4)
history = zero_dce_model.fit(train_dataset, validation_data=val_dataset, epochs=50)

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_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.

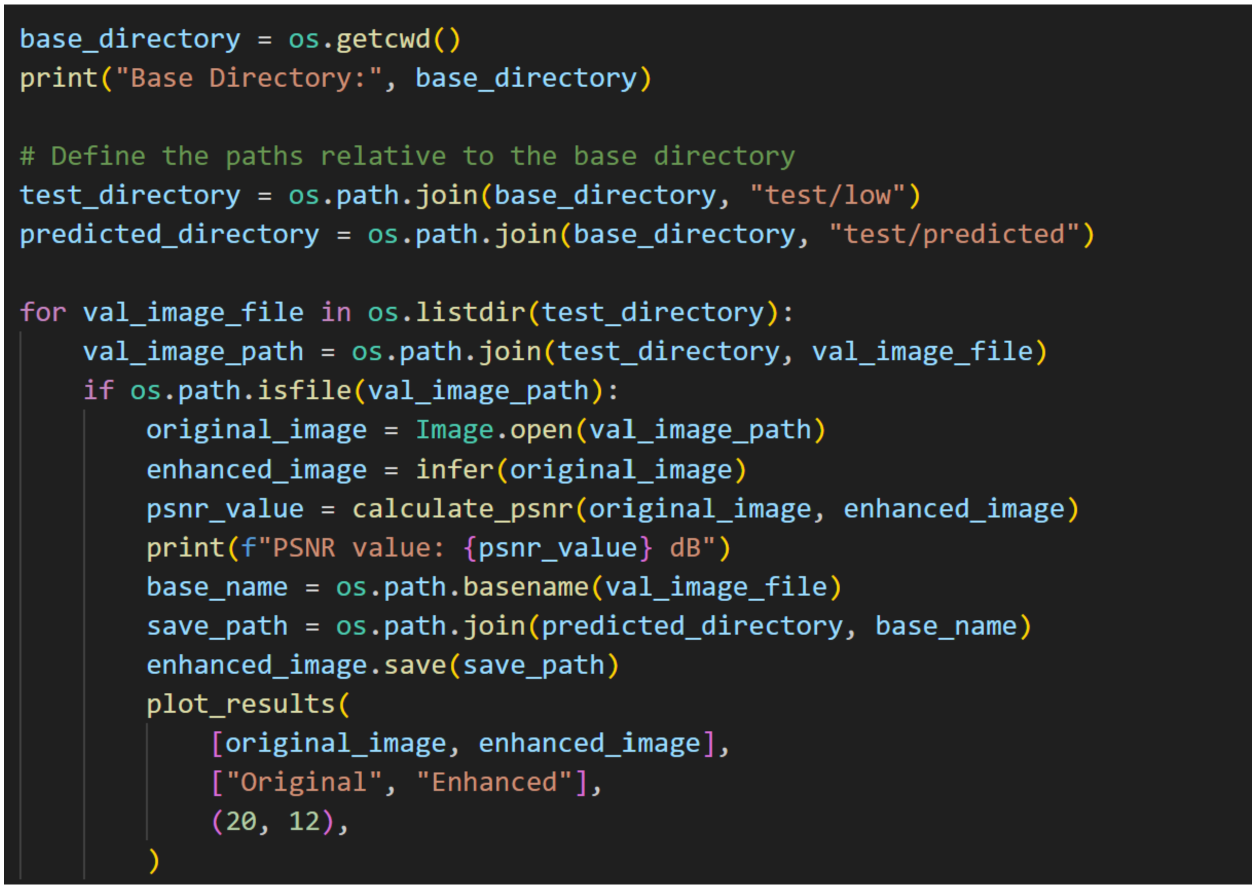
infer facilitates the application of the trained Zero-DCE model to a single input image, providing an enhanced version as output, which can be further analyzed or compared.

calculate_psnr offers a quantitative measure (PSNR) to evaluate the quality of the enhanced images compared to the originals, aiding in assessing the performance of the Zero-DCE model objectively.



```python
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()

# Inference
def infer(zero_dce_model, 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

# PSNR Calculation
def calculate_psnr(original, enhanced):
    original = np.array(original)
    enhanced = np.array(enhanced)
    mse = np.mean((original - enhanced) ** 2)
    if mse == 0:
        return float('inf')
    max_pixel = 255.0
    psnr = 20 * np.log10(max_pixel / np.sqrt(mse))
    return psnr
```

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



```python
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")

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)
        psnr_value = calculate_psnr(original_image, enhanced_image)
        print(f"PSNR value: {psnr_value} dB")
        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),
        )
```