Low Light Image Enhancement

Shashank Yadav (22123041)

Introduction :

With the objective of restoring high-quality image content from its degraded versions, the field of image restoration has numerous applications in areas such as photography, security, medical imaging, and remote sensing. In this report, we implement the `MIRNet` model for low-light image enhancement.

This `fully-convolutional architecture` is designed to learn an enriched set of features by combining contextual information from multiple scales, while simultaneously preserving high-resolution spatial details. The innovative approach of MIRNet effectively revitalizes degraded images, enhancing their clarity and quality.

Code Explanation :

The initial code sets up the environment and imports essential libraries for the low light enhancement project. It specifies `TensorFlow` as the backend for `Keras`, ensuring compatibility and performance.

Key libraries imported include `os` for environment management, `random` and `numpy` for data manipulation, `glob` for file handling, `PIL` for image processing, `matplotlib.pyplot` for visualization, and both `keras` and `tensorflow` for building and training the neural network model.

This setup is crucial for tasks such as loading and pre-processing images, constructing the deep learning model, and visualizing results.

This further code defines functions to read, preprocess, and create datasets for low light enhancement tasks. The `read_image` function reads and normalizes images, while `random_crop` crops them to a specified size. The `load_data` function applies these operations to pairs of low light and enhanced images. Finally, `get_dataset` creates a TensorFlow dataset from lists of image paths, maps the `load_data` function to them, and

batches the data. This setup prepares the images for training a deep learning model efficiently.

Then prepares datasets for training and validation for a low light enhancement model. It starts by defining the paths for low light and enhanced images from the LOL dataset, splitting them into training and validation sets based on the `MAX_TRAIN_IMAGES` limit.

The `glob` function is used to list the image files, and `sorted` ensures the images are processed in a consistent order. The `get_dataset` function, previously defined, is then used to create TensorFlow datasets from these image paths. Finally, the structure of the training and validation datasets is printed using `element_spec`, helping verify the dataset setup.

The function, `selective_kernel_feature_fusion`, fuses multi-scale features using a selective kernel mechanism to enhance feature representation in a neural network. It takes three multi-scale feature maps as input and processes them as follows:

1. **Combining Features**:
    - The three input feature maps are combined using the `layers.Add()` function.
2. **Global Average Pooling**:
    - A global average pooling layer (`GlobalAveragePooling2D`) reduces each feature map to a single value per channel, summarizing the global spatial information.
3. **Channel-Wise Statistics**:
    - The pooled features are reshaped to match the channel dimension.
4. **Compact Feature Representation**:
    - A `Conv2D` layer with a 1x1 kernel and ReLU activation reduces the channel dimension, creating a compact feature representation.
5. **Feature Descriptors**:
    - Three separate `Conv2D` layers with 1x1 kernels and softmax activation generate three feature descriptors, one for each input feature map. These descriptors highlight different aspects of the features.
6. **Weighted Features**:
    - The input feature maps are weighted by their corresponding feature descriptors using element-wise multiplication.
7. **Aggregated Feature**:
    - The weighted feature maps are combined using the `layers.Add()` function to produce the final aggregated feature.

This process enhances the feature maps by emphasizing important channels and integrating multi-scale information, which is crucial for tasks like low light image enhancement where different scales and features need to be fused effectively.

```python
class ChannelPooling(layers.Layer):
    def __init__(self, axis=-1, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.axis = axis
        self.concat = layers.Concatenate(axis=self.axis)

    def call(self, inputs):
        average_pooling = tf.expand_dims(tf.reduce_mean(inputs, axis=-1), axis=-1)
        max_pooling = tf.expand_dims(tf.reduce_max(inputs, axis=-1), axis=-1)
        return self.concat([average_pooling, max_pooling])

    def get_config(self):
        config = super().get_config()
        config.update({"axis": self.axis})
```

This `ChannelPooling` class defines a custom Keras layer that performs channel pooling by combining average and max pooling operations. The purpose of this layer is to extract robust feature representations by leveraging both the average and maximum values of the channels in the input tensor. Here's a breakdown of the functionality:

1. **Initialization**:
    - The `__init__` method initializes the layer with a specified axis for concatenation. It also initializes a `Concatenate` layer to combine the pooled outputs along the given axis.
2. **Forward Pass (call method)**:
    - The `call` method computes the average and maximum values across the specified axis (-1 by default, which refers to the channel dimension).
    - `tf.reduce_mean` and `tf.reduce_max` calculate the average and maximum values, respectively.
    - `tf.expand_dims` adds an extra dimension to these pooled results to make them compatible for concatenation.
    - The results of average and max pooling are concatenated along the specified axis to form the final output.
3. **Configuration (get_config method)**:
    - The `get_config` method returns the configuration of the layer, including the axis parameter. This is useful for saving and loading the model with this custom layer.

This custom layer is useful in scenarios where different pooling strategies need to be combined to capture varied features from the input tensor. In the context of low light enhancement, it helps in retaining significant information from the feature maps by considering both average and peak values.

`spatial_attention_block`

- Performs spatial attention by compressing the feature map using `ChannelPooling`, applying a 1x1 convolution, and then using sigmoid activation to emphasize spatially significant regions.

`channel_attention_block`

- Computes channel-wise attention by globally averaging the input tensor, reshaping it, applying convolutional layers with ReLU and sigmoid activations to emphasize important channels.

`dual_attention_unit_block`

- Integrates both spatial and channel attention mechanisms into a dual attention unit. It enhances feature maps by combining channel and spatial attention outputs and concatenating them before applying a 1x1 convolution and adding them back to the original input tensor.

`down_sampling_module(input_tensor)`

- Reduces spatial dimensions and increases channel dimensions using convolution and max pooling.
- Combines results from main and skip branches.

`up_sampling_module(input_tensor)`

- Increases spatial dimensions and decreases channel dimensions using convolution and upsampling.
- Combines results from main and skip branches.

`multi_scale_residual_block(input_tensor, channels)`

- Integrates downsampling, attention mechanisms, and feature fusion for enhanced feature representation.
- Utilizes dual attention units and selective kernel feature fusion.
- Includes a residual connection to preserve original information.

`multi_scale_residual_block(input_tensor, channels)`:

- Implements a basic multi-scale residual block with a convolutional layer followed by ReLU activation, and adds the input tensor back using element-wise addition.

`recursive_residual_group(input_tensor, num_mrb, channels)`:

- Creates a group of `num_mrb` multi-scale residual blocks (`multi_scale_residual_block`) applied recursively, with a final convolutional layer and addition of the input tensor.

`mirnet_model(num_rrg, num_mrb, channels)`:

- Defines a MIRNet model architecture using `num_rrg` recursive residual groups (`recursive_residual_group`), each containing `num_mrb` multi-scale residual blocks.
- Input shape is defined as `[None, None, 3]` to accommodate varying image sizes.
- The model ends with a convolutional layer to produce a 3-channel output and adds it back to the input tensor.

Layer (type)	Output Shape	Param #	Connected to
input_layer_1 (InputLayer)	(None, None, None, 3)	0	-
conv2d_638 (Conv2D)	(None, None, None, 64)	1,792	input_layer_1[0][0]
conv2d_639 (Conv2D)	(None, None, None, 64)	36,928	conv2d_638[0][0]
conv2d_640 (Conv2D)	(None, None, None, 64)	36,928	conv2d_639[0][0]
re_lu (ReLU)	(None, None, None, 64)	0	conv2d_640[0][0]
add_172 (Add)	(None, None, None, 64)	0	re_lu[0][0], conv2d_639[0][0]
conv2d_641 (Conv2D)	(None, None, None, 64)	36,928	add_172[0][0]
re_lu_1 (ReLU)	(None, None, None, 64)	0	conv2d_641[0][0]
add_173 (Add)	(None, None, None, 64)	0	re_lu_1[0][0], add_172[0][0]
conv2d_642 (Conv2D)	(None, None, None, 64)	36,928	add_173[0][0]
add_174 (Add)	(None, None, None, 64)	0	conv2d_642[0][0], conv2d_638[0][0]
conv2d_643 (Conv2D)	(None, None, None, 64)	36,928	add_174[0][0]
conv2d_644 (Conv2D)	(None, None, None, 64)	36,928	conv2d_643[0][0]
re_lu_2 (ReLU)	(None, None, None, 64)	0	conv2d_644[0][0]

add_174 (Add)	(None, None, None, 64)	0	conv2d_642[0][0], conv2d_638[0][0]
conv2d_643 (Conv2D)	(None, None, None, 64)	36,928	add_174[0][0]
conv2d_644 (Conv2D)	(None, None, None, 64)	36,928	conv2d_643[0][0]
re_lu_2 (ReLU)	(None, None, None, 64)	0	conv2d_644[0][0]
add_175 (Add)	(None, None, None, 64)	0	re_lu_2[0][0], conv2d_643[0][0]
conv2d_645 (Conv2D)	(None, None, None, 64)	36,928	add_175[0][0]
re_lu_3 (ReLU)	(None, None, None, 64)	0	conv2d_645[0][0]
add_176 (Add)	(None, None, None, 64)	0	re_lu_3[0][0], add_175[0][0]
conv2d_646 (Conv2D)	(None, None, None, 64)	36,928	add_176[0][0]
add_177 (Add)	(None, None, None, 64)	0	conv2d_646[0][0], add_174[0][0]
conv2d_647 (Conv2D)	(None, None, None, 64)	36,928	add_177[0][0]
conv2d_648 (Conv2D)	(None, None, None, 64)	36,928	conv2d_647[0][0]
re_lu_4 (ReLU)	(None, None, None, 64)	0	conv2d_648[0][0]
add_178 (Add)	(None, None, None, 64)	0	re_lu_4[0][0], conv2d_647[0][0]
conv2d_649 (Conv2D)	(None, None, None, 64)	36,928	add_178[0][0]
conv2d_648 (Conv2D)	(None, None, None, 64)	36,928	conv2d_647[0][0]
re_lu_4 (ReLU)	(None, None, None, 64)	0	conv2d_648[0][0]
add_178 (Add)	(None, None, None, 64)	0	re_lu_4[0][0], conv2d_647[0][0]
conv2d_649 (Conv2D)	(None, None, None, 64)	36,928	add_178[0][0]
re_lu_5 (ReLU)	(None, None, None, 64)	0	conv2d_649[0][0]
add_179 (Add)	(None, None, None, 64)	0	re_lu_5[0][0], add_178[0][0]
conv2d_650 (Conv2D)	(None, None, None, 64)	36,928	add_179[0][0]
add_180 (Add)	(None, None, None, 64)	0	conv2d_650[0][0], add_177[0][0]
conv2d_651 (Conv2D)	(None, None, None, 3)	1,731	add_180[0][0]
add_181 (Add)	(None, None, None, 3)	0	input_layer_1[0][0], conv2d_651[0][0]

Total params: 446,659 (1.70 MB)

Trainable params: 446,659 (1.70 MB)

Non-trainable params: 0 (0.00 B)

This summary provides a clear view of the model's structure, including the types and shapes of each layer, the number of parameters, and the connections between layers.

the PSNR values calculated for each image, along with the average PSNR across all images:

```
PSNR for image 1: 27.698441204307937 dB
PSNR for image 2: 27.74321468372671 dB
PSNR for image 3: 27.33666576129719 dB
PSNR for image 4: 27.51143520634492 dB
PSNR for image 5: 28.30195293555368 dB
PSNR for image 6: 27.824335888501352 dB
PSNR for image 7: 28.24426688612625 dB
PSNR for image 8: 27.424972013516594 dB
PSNR for image 9: 27.853276486707323 dB
PSNR for image 10: 27.78484588750278 dB
PSNR for image 11: 27.381863243342018 dB
PSNR for image 12: 27.524323108641372 dB
PSNR for image 13: 27.592061788026264 dB
PSNR for image 14: 27.948638803315283 dB
PSNR for image 15: 28.619142339707963 dB

Average PSNR: 27.785962415774513 dB
```

These PSNR values indicate significantly higher quality of enhancement, with an average PSNR of approximately 27.79 dB across all images. This suggests that the model has performed exceptionally well in enhancing the low-light images, achieving high fidelity compared to their original versions.

Conclusion :

This project successfully implemented a custom image enhancement model for low-light images and evaluated its performance using PSNR (Peak Signal-to-Noise Ratio). The script automated the process of enhancing images, calculating PSNR, and saving results, providing quantitative insights into image quality improvements.

Future Recommendations

Moving forward, consider exploring advanced enhancement techniques like GANs or CNNs, integrating additional evaluation metrics beyond PSNR for comprehensive assessment,

enhancing dataset diversity, optimizing for real-time processing, developing user-friendly interfaces, and continuously refining performance for broader applicability and usability.