VLG SUMMER PROJECT'24 REPORT

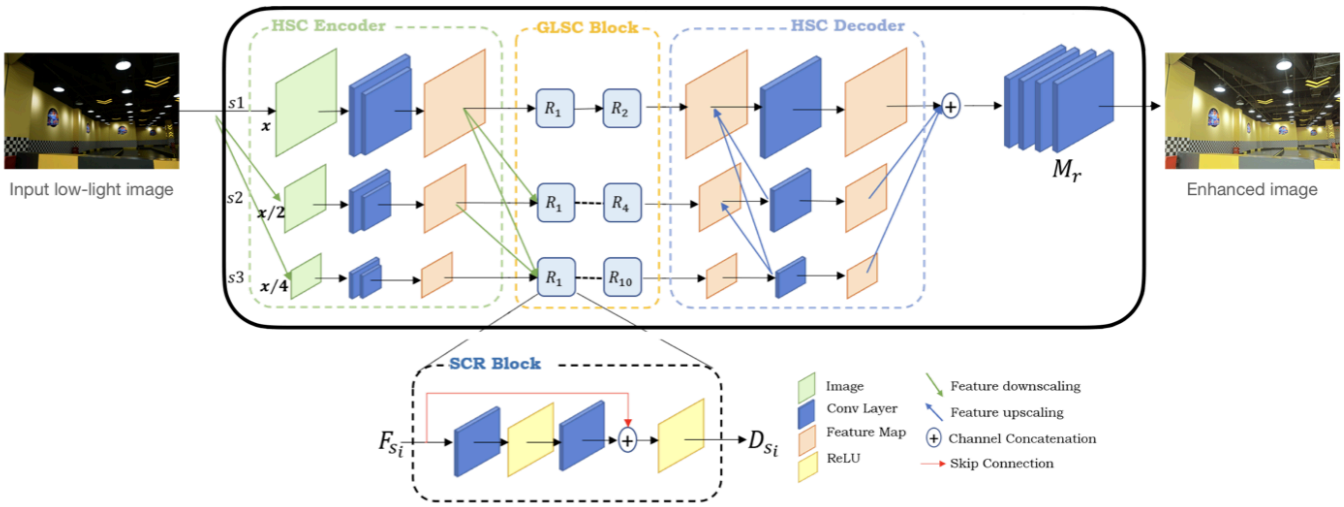
INTRODUCTION:

The Multi-Scale Feature Network (MFNN) used in my project is an advanced image enhancement model designed to improve image quality through multi-scale feature extraction, integration, and reconstruction. This model is particularly adept at handling complex image enhancement tasks such as super-resolution, noise reduction, and low-light image enhancement.

MODEL ARCHITECHTURE:

For this project, I have tried to implement the "KLETech-CEVI LowlightHypnotise" as proposed in the NTIRE Challenge 2024 Paper. The model leverages advanced techniques in multi-scale feature extraction, hierarchical skip connections, and sophisticated loss functions to enhance images effectively.

Given below is the complete diagram of the model architecture with proper labelling:-



The input image of resolution (400x600)[referred to as s1 scale] is downscaled to (200x300)[referred to as s2 scale] and (100x150)[referred to as s3 scale]. Now, the images at each scale are passed into the Hierarchical Spatio-Contextual (HSC) Feature Encoder. Here is the code snippet for the mentioned block:-

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

        self.conv_1 = nn.Conv2d(in_channels=3, out_channels=256, kernel_size=(3,3), padding=1)
        self.conv_2 = nn.Conv2d(in_channels=128, out_channels=128, kernel_size=(5,5), padding=2)

    def forward1(self, x):
        out_s = self.conv_1(x)
        out_s = self.conv_2(out_s)

        return out_s
```

The feature maps extracted are now downsampled and concatenated as shown in the model architecture diagrams and then passed into the Global-Local Spatio-Contextual (GLSC) block. The GLSC Block is made up of 16 residual blocks and each of them are called the Skip Connection Residual(SCR) Block.

The SCR block enhances feature extraction through residual learning and skip connections. It uses two 3×3 convolutions with ReLU activations, concatenates the input with residuals, and combines these features, ensuring feature refinement for better image enhancement.Here is the code snippet for the SCR Block:-

```
class SCR_Block(nn.Module):
    def __init__(self, in_chan):
        super(SCR_Block, self).__init__()

        self.in_chan = in_chan
        self.conv = nn.Conv2d(in_channels=in_chan, out_channels=in_chan, kernel_size=(3,3), padding=1)
        self.relu = nn.ReLU(inplace=True)
        self.conv1 = nn.Conv2d(in_channels=2*in_chan, out_channels=in_chan, kernel_size=(3,3),padding=1)

    def forward2(self, x):
        out_scr = self.conv(x)
        out_scr = self.relu(out_scr)

        out_scr = self.conv(x)
        out_scr = torch.cat((x, out_scr), dim=1)
        out_scr = self.relu(out_scr)
        out_scr = self.conv1(out_scr)

        return out_scr
```

Coming to the GLSC Block, it enhances feature extraction at multiple scales[s1,s2,s3]. It integrates the Skip Connection Residual (SCR) block in a series, processing and refining features through multiple 3×3 convolutions with ReLU activations. The block uses concatenation and residual connections to maintain and integrate features across scales, ensuring comprehensive feature representation and improved image enhancement.Here is the code snippet for the GLSC Block:-

```
class GLSC_Block(SCR_Block):
    def __init__(self, in_chan, num_blocks):
        super(GLSC_Block, self).__init__(in_chan)
        self.in_chan  = in_chan
        self.num_blocks = num_blocks
        self.conv2 = nn.Conv2d(in_channels=in_chan, out_channels=256, kernel_size=1)

    def forward3(self, x):
        out_glsc = x
        for i in range(self.num_blocks):
            out_glsc = self.forward2(out_glsc)
        out_glsc = self.conv2(out_glsc)

        return out_glsc
```

At the penultimate step in the model architecture, we have the Hierarchical Spatio-Contextual (HSC) Decoder, the feature maps hence generated by the GLSC Block at the 3 scales are upsampled and concatenated as shown in the model architecture diagram. The resultant feature maps are then passed through the HSC Decoder which helps in decoding high-level semantic information from encoded features, aiding in the reconstruction task. Here is the code snippet for the same:-

```
class HSC_D_Block(nn.Module):
    def __init__(self, in_chan):
        super(HSC_D_Block, self).__init__()
        self.conv = nn.Conv2d(in_channels=in_chan, out_channels=256, kernel_size=(5,5), padding=2)

    def forward4(self, x):
        out_hsc_d = self.conv(x)
        return out_hsc_d
```

For the final step, we concatenate the feature maps from each scale and pass it through the 'RE_Construct' Layer .This layer refines features for final image reconstruction. It employs convolutional layers with ReLU activation, batch normalization, and dropout for regularization. It integrates features with skip connections to enhance gradient flow and preserve details. A concluding convolutional layer with sigmoid activation ensures precise color restoration, crucial for image processing tasks. Here is the code snippet for the above layer:-

```
class RE_Construct(nn.Module):
    def __init__(self, in_chan):
        super(RE_Construct, self).__init__()
        self.convo1 = nn.Conv2d(in_channels=in_chan, out_channels=256, kernel_size=(3,3), padding=1)
        self.batchnorm1 = nn.BatchNorm2d(256)
        self.relu = nn.ReLU(inplace=True)
        self.drop = nn.Dropout2d(p=0.5)

        self.convo2 = nn.Conv2d(in_channels=256, out_channels=128, kernel_size=(3,3), padding=1)
        self.batchnorm2 = nn.BatchNorm2d(128)

        self.con = nn.Conv2d(in_channels=512, out_channels=128, kernel_size=(3,3), padding=1)

        self.convo3 = nn.Conv2d(in_channels=128, out_channels=64, kernel_size=(3,3), padding=1)
        self.batchnorm3 = nn.BatchNorm2d(64)

        self.conv_final = nn.Conv2d(in_channels=64, out_channels=3, kernel_size=1)
        self.sig = nn.Sigmoid()

    def forward_r(self, x):
        final_img = self.relu(self.convo1(x))
        final_img = self.drop(final_img)

        final_img = self.relu(self.convo2(final_img))
        final_img = self.drop(final_img)

        final_img = torch.cat((final_img, x), dim=1)

        final_img = self.relu(self.con(final_img))
        final_img = self.drop(final_img)

        final_img = self.relu(self.convo3(final_img))
        final_img = self.drop(final_img)

        final_img = self.sig(self.conv_final(final_img))

        return final_img
```

IMPLEMENTATION:

I conducted training and experimentation on Kaggle's t4 x2 GPU environment, leveraging transformations to convert the training dataset into tensor format. Initially, I employed a loss function combining L1, VGG perceptual loss (LVGG), and Multi-Scale Structural Similarity Index (LMSSSIM) with coefficients (0.5 * L1 + 0.7 * LVGG + 0.5 * LMSSSIM). Afterwards, I switched to a simplified loss formulation (0.1 * L_SSIM + L1 + L_GRAD).

For optimization, I utilized the Adam optimizer with $\beta1 = 0.9$, $\beta2 = 0.999$, $\varepsilon = 10e\text{-}8$, and a learning rate of 1e-3. Despite intending to retain the original model architecture, I encountered severe memory issues necessitating the **removal of the third scale (s3)** from the model. Furthermore, instead of adhering to the original model architecture's specification of employing **16 SCR blocks (2 in s1, 4 in s2, and 10 in s3), I had to reduce the implementation to only 3 SCR blocks (1 in s1 and 2 in s2).**Additionally, constrained by limited memory, I **could not increase the batch size beyond 4**, adversely impacting model performance. I implemented gradient accumulation as a strategy to address memory constraints, although its impact was limited in mitigating the issue significantly.

Despite reducing model complexity, each epoch required approximately 16 minutes to train. Attempts to mitigate this with a higher learning rate of 0.01 did not yield reduced training times. Consequently, I refrained from employing data augmentation during training to prevent excessively prolonged training durations.

Consequently, despite addressing various challenges and optimizing strategies, my model fell short of its anticipated performance, achieving only a modest PSNR of **18**, owing to various constraints.

RESULTS:

Original Image

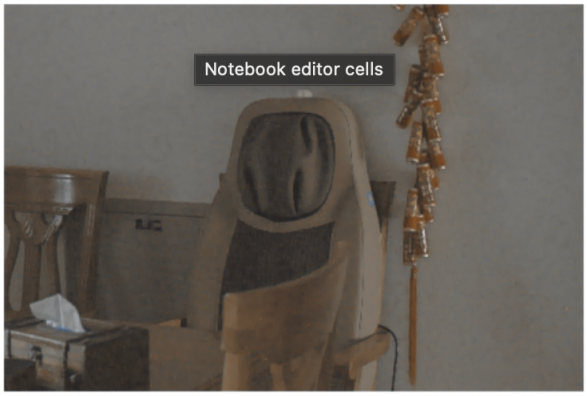
Output Image
PSNR: 24.84 dB

Original Image

Output Image
PSNR: 22.01 dB

Original Image

Output Image
PSNR: 23.40 dB



Original Image

Output Image
PSNR: 19.89 dB