MACHINE LEARNING FOR REMOTE SENSING-II(GNR-638)

MINI PROJECT-2

Deblurring the images by deep learning

Team Members-

Abhinav Vaishnav (210020003)
Lakavath Siddhartha (210020067)

TA Name-

Sanarbh Agarwal

Table of Content-

Introduction-

The objective of this task is to design a deep learning model capable of deblurring images. The process begins with preprocessing, where a set of images are downscaled to a resolution of 256x448 pixels, forming Set A. These images are then augmented by applying different Gaussian filters to create Set B. The next step involves designing a deep learning network that can transform the blurred images from Set B back into the original images from Set A. For evaluation, a test set and corresponding ground truth will be provided at a later stage, along with an evaluation script.

The ultimate goal of this task is to develop a model that can effectively deblur images, as evaluated by the PSNR score on the provided test set.

Methodology

Data description

- Dataset included 200 folders with different object images
- Each folder had 100 different images for a object from different angles.

Pre-processing and image augmentation

The following preprocessing steps are performed:

- **Resize**: Images are resized to a fixed resolution of 256x448 pixels using the *transforms.Resize* function.
- **ToTensor**: The images are converted from PIL Image format to PyTorch tensors using *transforms.ToTensor()*. This step is essential as neural networks in PyTorch expect input data to be in tensor format.
- **Normalization**: Pixel values of the images are normalized to have a mean of 0.5 and a standard deviation of 0.5 using *transforms.Normalize*.

Image augmentation is achieved through the following steps:

- ***CustomDataset Class***: A custom PyTorch dataset class (CustomDataset) is defined to load the images and apply transformations. The class takes a dataframe containing file paths of input (blurred) images and final (original) images.
- ***Transformations***: The specified transformations, including resizing and normalization, are applied to both the input and final images using the transform argument in the CustomDataset class.

Model architecture-

The model architecture consists of two main components: an encoder and a decoder, forming a convolutional neural network (CNN) called DeblurModel.

- Encoder-
  The encoder part of the model comprises a series of convolutional layers followed by batch normalization and ReLU activation functions.
- Decoder-
  It mirrors the structure of the encoder but in reverse, consisting of several transposed convolutional layers followed by batch normalization and ReLU activation functions. The final layer of the decoder uses the Tanh activation function.

Architecture and parameters:

```
Layer (type)               Output Shape         Param #
========================================================
Conv2d                     [-1, 64, 256, 448]   1,792
BatchNorm2d                [-1, 64, 256, 448]   128
ReLU                       [-1, 64, 256, 448]   0
Conv2d                     [-1, 64, 256, 448]   36,928
BatchNorm2d                [-1, 64, 256, 448]   128
ReLU                       [-1, 64, 256, 448]   0
Conv2d                     [-1, 128, 256, 448]  73,856
BatchNorm2d                [-1, 128, 256, 448]  256
ReLU                       [-1, 128, 256, 448]  0
Conv2d                     [-1, 128, 256, 448]  147,584
BatchNorm2d                [-1, 128, 256, 448]  256
ReLU                       [-1, 128, 256, 448]  0
Conv2d                     [-1, 256, 256, 448]  295,168
BatchNorm2d                [-1, 256, 256, 448]  512
ReLU                       [-1, 256, 256, 448]  0
Conv2d                     [-1, 256, 256, 448]  590,080
BatchNorm2d                [-1, 256, 256, 448]  512
ReLU                       [-1, 256, 256, 448]  0
Conv2d                     [-1, 512, 256, 448]  1,180,160
BatchNorm2d                [-1, 512, 256, 448]  1,024
ReLU                       [-1, 512, 256, 448]  0
Conv2d                     [-1, 256, 256, 448]  1,179,904
BatchNorm2d                [-1, 256, 256, 448]  512
ReLU                       [-1, 256, 256, 448]  0
ConvTranspose2d            [-1, 512, 256, 448]  1,180,160
BatchNorm2d                [-1, 512, 256, 448]  1,024
ReLU                       [-1, 512, 256, 448]  0
Conv2d                     [-1, 256, 256, 448]  1,179,904
BatchNorm2d                [-1, 256, 256, 448]  512
ReLU                       [-1, 256, 256, 448]  0
ConvTranspose2d            [-1, 128, 256, 448]  295,040
BatchNorm2d                [-1, 128, 256, 448]  256
ReLU                       [-1, 128, 256, 448]  0
Conv2d                     [-1, 128, 256, 448]  147,584
BatchNorm2d                [-1, 128, 256, 448]  256
ReLU                       [-1, 128, 256, 448]  0
ConvTranspose2d            [-1, 64, 256, 448]   73,792
BatchNorm2d                [-1, 64, 256, 448]   128
ReLU                       [-1, 64, 256, 448]   0
Conv2d                     [-1, 64, 256, 448]   36,928
BatchNorm2d                [-1, 64, 256, 448]   128
ReLU                       [-1, 64, 256, 448]   0
ConvTranspose2d            [-1, 3, 256, 448]    1,731
```

```
Tanh                          [-1, 3, 256, 448]    0
========================================================
Total params: 6,426,243
Trainable params: 6,426,243
Non-trainable params: 0
```

- ## Training process
  - The model architecture is defined using a convolutional neural network (CNN) consisting of encoder and decoder modules. The encoder extracts high-level features from the input images, while the decoder reconstructs the deblurred images from the extracted features. The model is then moved to the available device (GPU if available) for training.
  - During training, the dataset is split into training and validation sets using the *random_split* function. The DataLoader class is used to create iterable data loaders for both sets, enabling efficient batch processing.
  - The training loop iterates over each epoch, where for each batch in the training set, the optimizer's gradients are zeroed, and forward and backward passes are performed. The model is set to train mode, and gradient scaling is applied using PyTorch's GradScaler and autocast functionalities, ensuring numerical stability when training with mixed precision.
  - After each epoch, the model is evaluated on the validation set to monitor its performance. Once training completes, the trained model's state dictionary is saved to a file for future use.

Training loss curve:

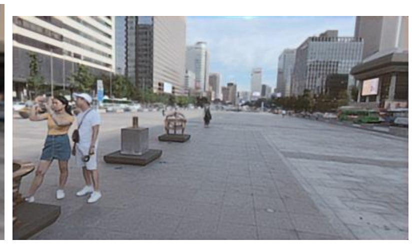
- ## Evaluation matrix

The evaluation of the model's performance is based on two main metrics: training loss and validation loss.

- **Training Loss**: This metric is calculated during each iteration of the training loop. It represents the difference between the model's predictions (outputs) and the ground truth (final images) for the current batch of training data. In this code, **Mean Squared Error (MSE)** is used as the loss function (criterion) to quantify this difference.
- **Validation Loss**: Similarly, the validation loss is computed after each epoch using a separate validation dataset. This metric measures the performance of the model on unseen data, helping to evaluate its generalization ability. Like the training loss, the validation loss is also calculated using the MSE loss function.

Sharp image

Blurred Image



Generated Image

• Results

Average PSNR between sharp and blur images: 26.681706113362516

Average PSNR between generated and blur images: 19.018229983689263

Minimum Loss Value:0.0145