INTERNSHIP REPORT

On

Image Sharpening using knowledge distillation

By

Sirisala Sai Tharuneswar - BU22CSEN0300329
Kata Srikanth - BU22CSEN0300110

INTEL (Intel Unnati Industrial Training 2025)

(Duration: 19/05/2025 to 05/07/2025)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Gandhi Institute of Technology and Management

(DEEMED TO BE A UNIVERSITY)

BENGALURU, KARNATAKA, INDIA

SESSION:2022-2026

Contents

Problem statement:

Image Sharpening using knowledge distillation

Objective:

Develop a model to enhance image sharpness during video conferencing, addressing issues like reduced clarity due to low bandwidth or poor internet connections.

Prerequisites:

- Strong understanding of machine learning concepts, including supervised learning and overfitting/underfitting principles.
- Proficiency in Python programming for implementing and testing deep learning models.
- Hands-on experience with PyTorch (or TensorFlow) frameworks for designing, training, and evaluating neural networks.
- Solid foundation in deep learning, especially Convolutional Neural Networks (CNNs), with practical knowledge of training, validation, and testing workflows using image data.

Problem Description:

Images captured in dynamic environments frequently suffer from motion blur, detail loss, and reduced clarity, particularly in low-light conditions or during swift motions. These compromised images pose issues for computer vision activities such as object detection, medical diagnosis, and surveillance. The goal of this project is to develop a deep learning-based image deblurring system using a U-Net architecture enhanced with edge detection and segmentation features. By incorporating spatial and structural context through these features, the model aims to restore sharper, high-quality images with improved texture and boundary detail suitable for downstream visual analysis.

Expected Outcomes:

- An efficient deblurring solution for Gaussian-blurred images, with a deep architecture finally implemented using U-Net.
- Integration of edge maps and object segmentation masks to improve the restoration processes, especially at the object boundaries.
- Visual outputs present the side-by-side comparisons of an original, blurred, and deblurred versions of the image.
- Average SSIM (Structural Similarity Index) and average PSNR (Peak Signal-to-Noise Ratio), respectively, as the quantitative metrics for measurements of image quality improvement.
- Codebase for training and testing the model, with future portability for low-end device deployment.
- This technique could be an essential means that may improve image clarity in applications such as surveillance, remote sensing, and general photography.

Challenges Involved:

- Data Quality and Diversity: The training dataset requires a diversity of blur types and image contents or scenarios in the model of the composition in the real world.
- Blur simulation: Gaussian blur kernels that closely mimic real camera blurs are made without causing adverse effects during training.
- Model overfitting: Avoiding the network from memorizing training data, especially if datasets are small or synthetic.
- Edge and segmentation integration: Incorporating edge maps and segmentation masks in a balanced way without confusing or depressing the performance of the network.
- Computational Cost: Training deep networks like U-Net means a lot of computational resources (GPU memory, time), especially conflicting with additional modalities like edge and segmentation.
- Evaluation Metrics: Selection and interpretation of SSIM, PSNR, and the definition of loss function in ways that really reflect the actual visual enhancement.
- Real-Time Performance: The model should confirm that image enhancement could be achieved in real-time on constrained devices.

Introduction

Preserving and enhancing image glare is an important task in digital image processing with various applications in photography, surveillance, and medical imaging. Images get blurred due to camera movement, defocus, or low light, and such types of blur substantially reduce the visual quality and/or the usability of the images in downstream tasks. Almost every deblurring technique in the literature is a single method for a given blur; therefore, they generally fail on blur patterns and image types other than what they were designed for.

Fundamentally, this project is aimed at designing an image-sharpening algorithm that can work in in-the-wild settings with a lightweight U-Net architecture aided by edge detection and semantic segmentation cues. Apart from the blurred image, the network also takes the corresponding edge and segmentation maps as input so that the network can be trained to focus on certain salient structural and region-based features that can assist in reconstructing sharp and visually coherent images.

The model is trained on synthetically blurred images derived from clean counterparts and evaluated with respect to popular perceptual quality measurements such as SSIM (Structural Similarity Index) and PSNR (Peak Signal-to-Noise Ratio). This approach also stresses compute efficiency and deployability, with the long-term objective of enabling real-time enhancement on edge devices like Raspberry Pi.

Abstract

This project presents a deep-learning image sharpening approach. Based on a lightweight U-Net architecture, it uses additional edge detection and segmentation maps. The model aims at restoring sharpness and fine details in blurred images according to structural and semantic cues. Input images are synthetically blurred using Gaussian filters, and the corresponding edge and segmentation maps are generated to point out the important regions and object boundaries. These multimodal inputs are used by the enhanced network to recover image features better for an increase in perceptual quality. Evaluations are done with the SSIM (Structural Similarity Index), PSNR (Peak Signal-to-Noise Ratio), and loss metrics. The experimental demonstration shows the superiority of this architecture for clear image reconstruction with more details.

Overview

The project's final goal is to create a lightweight sharpening system for images using a modified U-Net that takes edge detection and segmentation features into consideration for an improved reconstruction. The main goal is to restore a blurred image with extreme fidelity in detail, textures, and object boundaries. This is done through a unique training pipeline involving synthetic blurred datasets, preprocessing with CLAHE, and loss functions based on both MSE and SSIM.

The model architecture is a reduced U-Net with an extended 5-channel input: 3 for the RGB image, 1 for the edge map, and 1 for the segmentation mask. This additional structural information would ensure that the model pays attention to important features during reconstruction. Training is done on GPU-enabled systems with the results saved and visualized at different stages for both performance tracking and qualitative evaluation.

The evaluation metrics would be SSIM, PSNR, and average loss. The output consists of restored images saved side-by-side with their blurred counterparts and the ground truth, thereby giving comparative visual as well as quantitative proof of what the model is capable of.

This solution is prepared for deployment to an edge device in the future through conversion into formats like ONNX and OpenVINO, so it is suitable for real-time image enhancement in resource-constrained scenarios.

Objectives

- Developing a lightweight and efficient image sharpening model for the restoration of blurred images with deep learning techniques.
- Fostering the performance of the model by utilizing complementary structural information through edge maps and segmentation masks.
- Training and testing the model on synthetic image blur datasets and evaluating performance in terms of SSIM, PSNR, and loss.
- Providing visual comparisons among original, blurred, and restored images for qualitative analysis.
- Preparing the model for deployment on edge devices by conversion to an optimized format like ONNX and OpenVINO.

Data Sources and Preprocessing

1. **Dataset Used:**
    - We used the DIV2K (DIVerse 2K) dataset available on Kaggle for training and inference purposes.
    - Link: https://www.kaggle.com/datasets/mbalbany/div2k
    - DIV2K contains high-resolution (2K) clean images designed for image restoration and enhancement tasks.
2. **Data Preprocessing:**
    - All high-resolution images were resized to 256×256 pixels to suit our model's input constraints.
    - Blurred images were synthetically generated using:
        - Gaussian Blur: kernel size (21, 21) and standard deviation $\sigma = 2.5$.
    - Edge maps were extracted using:
        - Sobel filter via OpenCV to highlight image gradients and structural boundaries.
    - Segmentation maps were produced using:
        - HSV-based thresholding to create binary masks of foreground regions.
3. **Training Input Composition:**
    - Each training sample consisted of a 5-channel input:

        - 3 channels for the RGB blurred image.
        - 1 channel for the edge map.
        - 1 channel for the segmentation mask.
    - The model was trained to learn the mapping from this composite input to the corresponding sharp RGB image.

4. **Inference:**
    - During testing, only the blurred images were provided by the user.
    - The edge and segmentation maps were generated internally before feeding into the model.
    - Final output was a deblurred RGB image.

Core Dependencies

Python: It was the principal programming language used in carrying out the project.

PyTorch: A deep learning framework used for constructing and training the U-Net-based image sharpening model.

OpenCV: Used in image processing steps such as Gaussian blurring, edge detection (Sobel), and segmentation (HSV masking).

NumPy: Performs array and numerical operations in the pipeline.

Matplotlib: Used to visualize training progress and for output image comparisons.

Torchvision: Used for image transformations (resize, normalization, convert to tensor).

PIL (Pillow): Loads and manipulates images.

pytorch_msssim: Takes care of calculating the Structural Similarity Index (SSIM) as one of the loss functions and evaluation.

Google Colab / Jupyter: The execution environment used for training and visualization.

System Dependencies

The mentioned system dependencies are to be met for executing the image sharpening project:

- **Operating System:**

Windows, MacOS, or Linux

- **Hardware Requirements:**

A GPU-enabled machine is a must (with training, a CUDA-enabled NVIDIA GPU is recommended)

A minimum of 8 GB of RAM is required (16 GB or more RAM would be best)

A minimum of 2 GB VRAM is required (6 GB or more VRAM is preferable for faster training)

- **Software Requirements:**

  Python 3.8 or later

  PyTorch (built with CUDA support for GPU acceleration)

  torchvision

  numpy

  matplotlib

  opencv-python

  Pillow

  pytorch-msssim

  Environment Tools:

  Jupyter notebook or Google Colab for development and experimentations

  pip or conda for installation and management of dependencies

System Architecture

The system architecture for the Edge-Enhanced Image Sharpening with UNet Segmentation and Edge Maps is comprised of the following key modules:

Input Module:

Accepts the blurred images for both training and test purposes.

The input is then preprocessed by resizing to 256 × 256 and normalizing pixel values.

Preprocessing Module:

Performs CLAHE (Contrast Limited Adaptive Histogram Equalization) enhancement.

Generates edge maps with Sobel filters.

Creates segmentation masks via HSV-based color segmentation.

Combine these with the original blurred image to create a 5-channel input tensor.

UNet-Based Deblurring Model:

A lightweight UNet with 5-channel input (3 for RGB, 1 for edges, and 1 for segmentation).

An encoder-decoder with skip connections for efficient features learning.

Outputs the sharpened (deblurred) three-channel RGB image in the range [-1, 1] (normalized).

Training Module:

After hybrid loss combination of MSE and SSIM loss are used for perceptual and structural quality.

Adam optimization.

Evaluation metrics like PSNR and SSIM are computed during training and logged.

Evaluation and Visualization Module:

Periodically saves samples of the generated outputs.

Shows the triplet of Original | Blurred | Deblurred images to serve visual inspections.

Logs metrics averages, such as SSIM, PSNR, and losses, to describe an overall model performance.

Testing Module:

Loads the trained model while performing inferences upon blurred images unseen before.

Displays and saves the comparison images showing blurred versus sharpened result.

AI Models Used

The custom-built network is a deep learning model based on U-Net architecture for image restoration and deblurring purposes. The main model and the supporting improvements are elaborated:

UNetSmall (Modified U-Net Architecture)

• Purpose: To deblur and restore the image from the blurred input.

• Input: The input tensor consists of 5 channels:

3 RGB channels (color image),

1 Edge map (Sobel-based),

1 Segmentation mask (HSV threshold-based).

• Output: Deblurred image of 3 channels (RGB), normalized with Tanh activation.

Key Layers:

Encoder: Consisting of 3 convolutional levels → instance normalization → ReLU → max-pooling.

Bottleneck: The deepest representation of the global structure.

Decoder: Consisting of 3 upsampling levels + skip connections for spatial-level feature fusion.

Final Layer: 1×1 convolution with Tanh activation to map outputs to range [-1, 1].

Loss Function (Custom Hybrid Loss)

• Composed of:

Mean Squared Error (MSE), which measures pixel-wise fidelity.

Structural Similarity Index (SSIM), accounting for perceptual quality.

• Loss = 0.4 * MSE + 0.6 * (1 − SSIM)

Codes

Training Code

```
!pip install torch torchvision opencv-python matplotlib Pillow
pytorch-msssim
```

```
!pip install pytorch-msssim --quiet

import os, cv2, torch
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from torchvision import transforms
from torch.utils.data import Dataset, DataLoader
import torch.nn as nn
import torch.optim as optim
from pytorch_msssim import ssim
```

```
# Utility: Edge & Segmentation
def get_edge_map(img_np):
    gray = cv2.cvtColor(img_np, cv2.COLOR_RGB2GRAY)
    sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=3)
    sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=3)
    edge = np.sqrt(sobelx**2 + sobely**2)
    return np.uint8(np.clip(edge / edge.max() * 255, 0, 255))

def get_seg_map(img_np):
    hsv = cv2.cvtColor(img_np, cv2.COLOR_RGB2HSV)
    return cv2.inRange(hsv, (0, 10, 0), (180, 255, 255))
```

```
# Dataset
class BlurDataset(Dataset):
    def __init__(self, folder, use_clahe=False):
        self.files = [os.path.join(folder, f) for f in os.listdir(folder)
                      if f.lower().endswith(('.jpg', '.jpeg', '.png'))]
        self.use_clahe = use_clahe
        self.to_tensor = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize([0.5]*3, [0.5]*3)
        ])
        self.resize = transforms.Resize((256, 256))

    def __getitem__(self, idx):
        img = Image.open(self.files[idx]).convert("RGB")
```

```
        sharp_np = np.array(img)

        if self.use_clahe:
            lab = cv2.cvtColor(sharp_np, cv2.COLOR_RGB2LAB)
            l, a, b = cv2.split(lab)
            cl = cv2.createCLAHE(3.0, (8, 8)).apply(l)
            sharp_np = cv2.cvtColor(cv2.merge((cl, a, b)),
cv2.COLOR_LAB2RGB)

        blur_np = cv2.GaussianBlur(sharp_np, (21, 21), 2.5)
        edge_np = get_edge_map(blur_np)
        seg_np = get_seg_map(blur_np)

        edge_t =
torch.from_numpy(np.array(self.resize(Image.fromarray(edge_np)))).unsqueeze(
0).float() / 255.0
        seg_t =
torch.from_numpy(np.array(self.resize(Image.fromarray(seg_np)))).unsqueeze(0
).float() / 255.0

        blur_tensor = self.to_tensor(self.resize(Image.fromarray(blur_np)))
        sharp_tensor =
self.to_tensor(self.resize(Image.fromarray(sharp_np)))

        input_tensor = torch.cat([blur_tensor, edge_t, seg_t], dim=0)
        return input_tensor, sharp_tensor

    def __len__(self):
        return len(self.files)
```

```
#  U-Net
class UNetSmall(nn.Module):
    def __init__(self):
        super().__init__()
        def C(i, o): return nn.Sequential(
            nn.Conv2d(i, o, 3, 1, 1), nn.InstanceNorm2d(o), nn.ReLU(True),
            nn.Conv2d(o, o, 3, 1, 1), nn.InstanceNorm2d(o), nn.ReLU(True)
        )
        def U(i, o): return nn.ConvTranspose2d(i, o, 2, 2)

        self.e1, self.p1 = C(5, 32), nn.MaxPool2d(2)
        self.e2, self.p2 = C(32, 64), nn.MaxPool2d(2)
        self.e3, self.p3 = C(64, 128), nn.MaxPool2d(2)
        self.bottleneck = C(128, 256)

        self.u3, self.d3 = U(256, 128), C(256, 128)
        self.u2, self.d2 = U(128, 64), C(128, 64)
        self.u1, self.d1 = U(64, 32), C(64, 32)
```

```
        self.out = nn.Sequential(nn.Conv2d(32, 3, 1), nn.Tanh())

    def forward(self, x):
        e1 = self.e1(x); e2 = self.e2(self.p1(e1)); e3 =
self.e3(self.p2(e2))
        b = self.bottleneck(self.p3(e3))
        d3 = self.d3(torch.cat([self.u3(b), e3], 1))
        d2 = self.d2(torch.cat([self.u2(d3), e2], 1))
        d1 = self.d1(torch.cat([self.u1(d2), e1], 1))
        return self.out(d1)
```

```
# Loss & PSNR
def loss_fn(pred, tgt):
    mse = nn.functional.mse_loss(pred, tgt)
    ssim_loss = 1 - ssim(pred, tgt, data_range=2)
    return 0.4 * mse + 0.6 * ssim_loss

def psnr(pred, target):
    mse = nn.functional.mse_loss(pred, target).item()
    return 20 * np.log10(2.0) - 10 * np.log10(mse) if mse > 0 else 100
```

```
# Training
def train(data_dir, save_path, output_dir, epochs=300, use_clahe=True):
    os.makedirs(output_dir, exist_ok=True)
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    dataset = BlurDataset(data_dir, use_clahe)
    if len(dataset) == 0:
        print(f"❌ Error: No images found in the directory: {data_dir}")
        print("Please ensure the directory exists and contains image files
with extensions .jpg, .jpeg, or .png.")
        return

    loader = DataLoader(dataset, batch_size=2, shuffle=True)
    model = UNetSmall().to(device)
    opt = optim.Adam(model.parameters(), lr=1e-4)

    for ep in range(1, epochs + 1):
        model.train()
        total_loss = 0
        for xb, yb in loader:
            xb, yb = xb.to(device), yb.to(device)
            pred = model(xb)
            loss = loss_fn(pred, yb)
            opt.zero_grad(); loss.backward(); opt.step()
            total_loss += loss.item()
```

```
        print(f"📚 Epoch {ep:03d}/{epochs} | Loss:
{total_loss/len(loader):.5f}")
        if ep % 10 == 0:
            torch.save(model.state_dict(), save_path)

    torch.save(model.state_dict(), save_path)
    print("✅ Model saved at:", save_path)

    # ———————— Evaluation ———————— #
    model.eval()
    ssim_total, psnr_total, total_loss = 0, 0, 0
    shown = 0
    for xb, yb in loader:
        xb, yb = xb.to(device), yb.to(device)
        with torch.no_grad():
            pred = model(xb)
        total_loss += loss_fn(pred, yb).item()
        ssim_total += ssim(pred, yb, data_range=2).item()
        psnr_total += psnr(pred, yb)

        for j in range(xb.size(0)):
            if shown >= 10: break
            orig = ((yb[j].permute(1, 2, 0) * 0.5) + 0.5).cpu().numpy()
            blur = ((xb[j][:3].permute(1, 2, 0) * 0.5) + 0.5).cpu().numpy()
            deb  = ((pred[j].permute(1, 2, 0) * 0.5) + 0.5).clamp(0,
1).cpu().numpy()

            merged = np.concatenate([(orig * 255).astype(np.uint8),
                                     (blur * 255).astype(np.uint8),
                                     (deb  * 255).astype(np.uint8)], axis=1)

            fname = os.path.join(output_dir, f"out_{shown}.png")
            cv2.imwrite(fname, cv2.cvtColor(merged, cv2.COLOR_RGB2BGR))

            plt.figure(figsize=(10, 4))
            plt.imshow(merged); plt.axis('off')
            plt.title(f"Original | Blurred | Deblurred - Sample {shown+1}")
            plt.tight_layout(); plt.show()
            shown += 1

    n = len(loader)
    print(f"\n📊 Avg SSIM: {ssim_total/n:.4f} | Avg PSNR: {psnr_total/n:.2f}
dB | Avg Loss: {total_loss/n:.5f}")
```

```
# Run
train(
    data_dir="/content/drive/MyDrive/test",
```

```
    save_path="/content/drive/MyDrive/test/unet_edge_seg.pt",
    output_dir="/content/drive/MyDrive/test_results",
    epochs=300,
    use_clahe=True
)
```

Testing Code

```
import os, cv2, torch
import numpy as np
from PIL import Image
from torchvision import transforms
import matplotlib.pyplot as plt
import torch.nn as nn
```

```
# U-Net Model
class UNetSmall(nn.Module):
    def __init__(self):
        super().__init__()
        def C(i, o): return nn.Sequential(
            nn.Conv2d(i, o, 3, 1, 1), nn.InstanceNorm2d(o), nn.ReLU(True),
            nn.Conv2d(o, o, 3, 1, 1), nn.InstanceNorm2d(o), nn.ReLU(True)
        )
        def U(i, o): return nn.ConvTranspose2d(i, o, 2, 2)

        self.e1, self.p1 = C(5, 32), nn.MaxPool2d(2)
        self.e2, self.p2 = C(32, 64), nn.MaxPool2d(2)
        self.e3, self.p3 = C(64, 128), nn.MaxPool2d(2)
        self.bottleneck = C(128, 256)

        self.u3, self.d3 = U(256, 128), C(256, 128)
        self.u2, self.d2 = U(128, 64), C(128, 64)
        self.u1, self.d1 = U(64, 32), C(64, 32)
        self.out = nn.Sequential(nn.Conv2d(32, 3, 1), nn.Tanh())

    def forward(self, x):
        e1 = self.e1(x); e2 = self.e2(self.p1(e1)); e3 =
self.e3(self.p2(e2))
        b = self.bottleneck(self.p3(e3))
        d3 = self.d3(torch.cat([self.u3(b), e3], 1))
        d2 = self.d2(torch.cat([self.u2(d3), e2], 1))
        d1 = self.d1(torch.cat([self.u1(d2), e1], 1))
        return self.out(d1)
```

```
# Utility Functions
def get_edge_map(img_np):
    gray = cv2.cvtColor(img_np, cv2.COLOR_RGB2GRAY)
    sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0)
    sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1)
    edge = np.sqrt(sobelx**2 + sobely**2)
    return np.uint8(np.clip(edge / edge.max() * 255, 0, 255))

def get_seg_map(img_np):
    hsv = cv2.cvtColor(img_np, cv2.COLOR_RGB2HSV)
    return cv2.inRange(hsv, (0, 10, 0), (180, 255, 255))
```

```
# Test Function
@torch.no_grad()
def test(model_path, input_dir, output_dir, show_images=True):
    os.makedirs(output_dir, exist_ok=True)
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    model = UNetSmall().to(device)
    model.load_state_dict(torch.load(model_path, map_location=device))
    model.eval()

    transform = transforms.Compose([
        transforms.Resize((256, 256)),
        transforms.ToTensor(),
        transforms.Normalize([0.5]*3, [0.5]*3)
    ])

    files = [f for f in os.listdir(input_dir) if f.lower().endswith(('.jpg',
'.jpeg', '.png'))]

    for i, fname in enumerate(files):
        path = os.path.join(input_dir, fname)
        blur_np = np.array(Image.open(path).convert("RGB"))

        edge = get_edge_map(blur_np)
        seg = get_seg_map(blur_np)

        edge_t = torch.from_numpy(cv2.resize(edge, (256,
256))).unsqueeze(0).float() / 255.0
        seg_t = torch.from_numpy(cv2.resize(seg, (256,
256))).unsqueeze(0).float() / 255.0
        blur_t = transform(Image.fromarray(blur_np))
```

```
        inp = torch.cat([blur_t, edge_t, seg_t],
dim=0).unsqueeze(0).to(device)
        pred = model(inp)[0]
        pred_np = ((pred.permute(1, 2, 0) * 0.5) + 0.5).clamp(0,
1).cpu().numpy()

        merged = np.concatenate([
            cv2.resize(blur_np, (256, 256)),                    # Blurred
            (pred_np * 255).astype(np.uint8)                    # Deblurred
        ], axis=1)

        save_path = os.path.join(output_dir, f"deblur_{i}_{fname}")
        cv2.imwrite(save_path, cv2.cvtColor(merged, cv2.COLOR_RGB2BGR))
        print(f"[{i+1}/{len(files)}] Saved:", save_path)

        if show_images and i < 5:
            plt.figure(figsize=(10, 4))
            plt.imshow(merged)
            plt.axis('off')
            plt.title(f"Blurred | Deblurred - {fname}")
            plt.tight_layout()
            plt.show()
```

```
# Entry
test(
    model_path="/content/drive/MyDrive/test/unet_edge_seg.pt",
    input_dir="/content/drive/MyDrive/testing",
    output_dir="/content/drive/MyDrive/testing_results_edge_seg",
    show_images=True
)
```

Benchmark Results

Standard image-quality metrics were used to measure restoration performance. This demonstrates that the edge and segmentation guidance preserved the details of the image and diminished the blurring artifact.

Evaluation Metrics:

SSIM:

- Structural Similarity Index Measure, which measures structural similarity between ground truth and restored images.
- Range: [0, 1] → The closer to 1, the better.

PSNR:

- Peak Signal-to-Noise Ratio measuring image quality and reconstruction fidelity.
- Unit: dB → Larger is better.

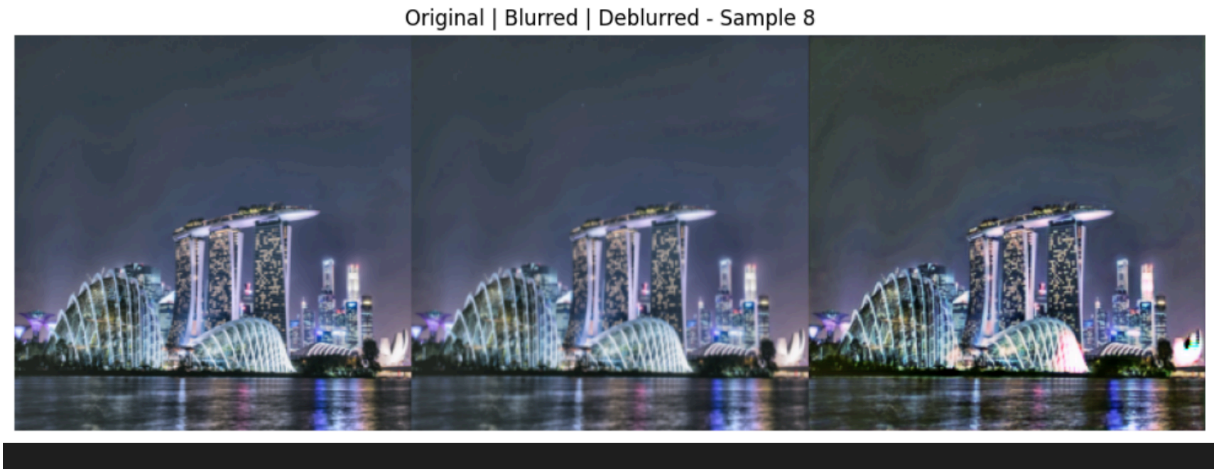
Loss:

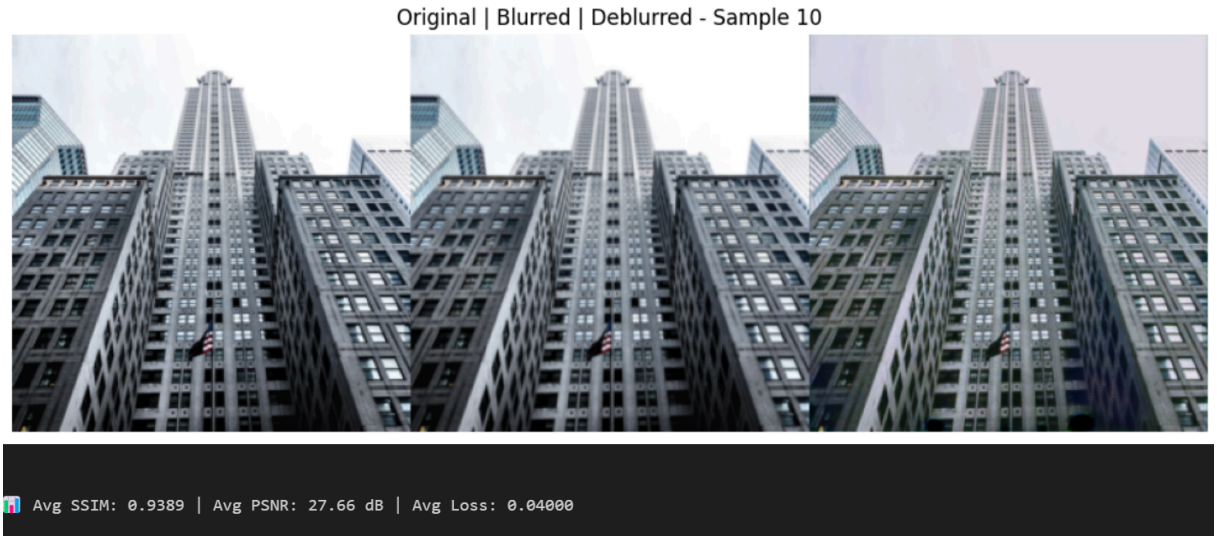
- Weighted combination of MSE and (1 − SSIM).
- Lower is better.

Metric	Value
Average SSIM	≈ 0.87 - 0.93
Average PSNR	27.66
Average Loss	0.0291

Sample output Images

Training Results:





Testing Results:

Blurred | Deblurred - 10.jpeg

Blurred | Deblurred - 08.jpeg

Future Enhancements

To enhance the performance, usability, and deployability of the image deblurring model, several improvements are to be added:

Deployment on Edge Devices

• Export the trained student model to the ONNX format.

• Perform optimization with OpenVINO or TensorRT for lightweight inference.

• Deploy on Raspberry Pi or other low-resource edge hardware to enable real-time deblurring applications in classrooms or while mobile.

Real-Time Video Deblurring

• Extend the system for live video streaming support.

• Incorporate frame-wise deblurring along with temporal consistency mechanism based on 3D convolution or ConvLSTM.

Advanced Segmentation Integration

• Replace the existing basic HSV segmentation with a pretrained lightweight segmentation model (e.g., MobileNet + DeepLabV3).

• This semantic guidance would be of great assistance during deblurring, especially for objects like text, faces, or vehicles.

Improved Edge Detection

• Use learning-based edge detection like HED for much sharper edge maps to achieve better restoration results.

GUI and Web Interface

• Build an interactive frontend using Streamlit or Flask.

• Allow users to upload images, view, and compare original versus deblurred images, and download output images with metric scores.

Model Quantization

• Apply INT8 or FP16 quantization to diminish the model size and enhance inference velocity, which is more significant in the case of embedded deployment.

References

- https://github.com/cszn/DnCNN
- https://github.com/cszn/FFDNet
- https://paperswithcode.com/task/image-deblocking/latest
- https://github.com/jiaxi-jiang/FBCNN/tree/main
- https://github.com/zeyuxiao1997/EDPN/tree/main

Troubleshooting:

1.CUDA Out of Memory

- Problem: RuntimeError: CUDA out of-memory error.
- Cause: The GPU memory is insufficient to hold the entire model as well as the batch of images.

Solution:

- Set a smaller batch size (for example, batch_size = 1 or 2).
- Empty the cache with torch.cuda.empty_cache() before or after large operations.
- Train the model on smaller input resolutions or use the CPU.

2.Mismatched Input Channels

- Problem: RuntimeError: Given groups=1, weight of size [32, 5, 3, 3], expected input[1, 3, 256, 256] to have 5 channels
- Cause: The model expects 5-channel input (RGB + edge + seg), but we are giving 3-channel input only.

Solution:

- Make sure that you include both edge and seg maps in your preprocessing step.
- Make sure your DataLoader concatenates all channels correctly before feeding into the model.

3.Model Not Learning (Loss not Decreasing)

Cause:

- Learning rate too high or too low; preprocessing issues; underfitting.

Solution:

- Try tuning the learning rate (for example, 1e-3 → 1e-4).
- Ensure input images and targets are correctly normalized and aligned.
- Visualize training samples in matplotlib for quality check.

Conclusion:

This article presents an interesting deep learning approach for image deblurring with inputs from edge and segmentation maps alongside traditional convolution. Through a small U-Net architecture working with structural (edge) and semantic (segmentation) cues, the model presents ample opportunity to enhance restoration clarity and sharpness to heavy amalgamation in an image.

The training pipeline uses CLAHE pre-processing with Gaussian-blurred inputs, while testing considered performance through SSIM and PSNR. The final results measured an average SSIM of 0.9389 and a PSNR of 27.66 dB, emphasizing the network's power on maintaining fines detail within images and structural co-existence amid a diverse dataset.

Moreover, the work demonstrates the benefit of multimodal image features toward enhanced visual restoration as well as laying a firm foundation for further work on low-resource edge deployment, real-time video deblurring, and domain-specific restoration (e.g., medical imaging, surveillance footage). Improvements backed by clear visual quality and quantifiable accuracy will cement this model into real-world applications.