Image Deblurring with a Lightweight DeepUNet Model

Project Title: Image Deblurring with Deep CNN

Students: DOUDOU Hissein & KADJO Aka Elishama
Professor: KHERIJI Walid
Course: Deep Learning
Date: 15/06/2025

This notebook demonstrates how to train a lightweight convolutional neural network to restore blurred images using synthetic data derived from the COCO dataset. The goal is to improve image clarity by learning a mapping from blurred to sharp images.

1. Imports and Device Configuration

We start by importing necessary libraries for data handling, model building, training, and evaluation. We also configure the device for acceleration using CUDA if available, followed by Apple MPS, and finally CPU as fallback.

```
import cv2
import numpy as np
import os
import random
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
import torch
import torch.nn as nn
import torchvision.transforms as T
from PIL import Image, ImageFilter
from tqdm import tqdm
from skimage.metrics import peak_signal_noise_ratio as psnr
from skimage.metrics import structural_similarity as ssim
import requests
from io import BytesIO

if torch.cuda.is_available():
    device = torch.device("cuda")
elif torch.backends.mps.is_available():
    device = torch.device("mps")
else:
    device = torch.device("cpu")

print(f"Using device: {device}")
```

```
Using device: mps
```

2. Dataset Preparation and Blur Augmentation

We use a subset of the COCO dataset as the base for generating training and test data. The COCO dataset was selected because it contains a large variety of real-world scenes with diverse lighting, textures, and object types. This diversity helps the model learn to deblur different types of content effectively. Instead of relying on a dedicated blurred image dataset, we simulate realistic blur (Gaussian, box, and motion) on clean images, allowing us to fully control the types and severity of degradation applied.

We define a custom PyTorch Dataset class that loads images and applies random blur (Gaussian, Box, or Motion) to create input/output pairs for training.

```
class CocoDeblurDataset(Dataset):
    def __init__(self, img_folder, transform=None, sample_size=5000):
        self.img_folder = img_folder
        all_files = [f for f in os.listdir(img_folder) if f.endswith((".jpg", ".png"))]
        self.image_files = random.sample(all_files, min(sample_size, len(all_files)))
        self.transform = transform

    def apply_random_blur(self, image):
        blur_type = random.choice(["gaussian", "box", "motion"])
        if blur_type == "gaussian":
            kernel_size = random.choice([11, 13])
            sigma = random.uniform(3.0, 5.0)
            return T.GaussianBlur(kernel_size=kernel_size, sigma=sigma)(image)
        elif blur_type == "box":
            return image.filter(ImageFilter.BoxBlur(radius=random.uniform(3, 5)))
        elif blur_type == "motion":
            kernel = np.zeros((21, 21))
            kernel[10, :] = np.ones(21)
            kernel = kernel / 21
            img_np = np.array(image)
            img_blur = cv2.filter2D(img_np, -1, kernel)
            return Image.fromarray(img_blur)

    def __len__(self):
        return len(self.image_files)

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_folder, self.image_files[idx])
        image = Image.open(img_path).convert("RGB")
        if self.transform:
            image = self.transform(image)
        blurred = self.apply_random_blur(T.ToPILImage()(image))
        blurred = self.transform(blurred)
        return blurred, image

transform = T.Compose([
    T.Resize((512, 512)),
    T.ToTensor()
])
```

3. DataLoader Setup

We prepare data loaders for both training and testing, and set fixed seeds for reproducibility.

```
SEED = 42
random.seed(SEED)
torch.manual_seed(SEED)
np.random.seed(SEED)
```

```
coco_folder_path = "data/test2017"

train_dataset = CocoDeblurDataset(coco_folder_path, transform=transform, sample_size=5000)
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)

test_dataset = CocoDeblurDataset(coco_folder_path, transform=transform, sample_size=500)
test_loader = DataLoader(test_dataset, batch_size=8, shuffle=False)
```

4. Sample Visualization

To ensure the blur is applied correctly, we display a few blurred vs sharp image pairs from the training set.

```
def show_random_samples(dataset, n=4):
    indices = random.sample(range(len(dataset)), n)
    plt.figure(figsize=(n * 3, 6))
    for i, idx in enumerate(indices):
        blur, sharp = dataset[idx]
        plt.subplot(2, n, i + 1)
        plt.imshow(blur.permute(1, 2, 0).numpy())
        plt.axis("off")
        if i == 0: plt.ylabel("Blurred")
        plt.subplot(2, n, i + 1 + n)
        plt.imshow(sharp.permute(1, 2, 0).numpy())
        plt.axis("off")
        if i == 0: plt.ylabel("Sharp")
    plt.suptitle("Blurred vs. Sharp Image Samples")
    plt.tight_layout()
    plt.show()

show_random_samples(train_dataset)
```

5. DeepUNet Model Architecture

We implement a custom convolutional neural network based on the U-Net architecture, which falls under the family of deep CNNs. U-Net is well-suited for image-to-image translation tasks such as denoising or deblurring due to its symmetric encoder-decoder structure and use of skip connections. Our choice to use DeepUNet is motivated by its balance of efficiency and accuracy, making it appropriate for training on mid-sized datasets within limited computational resources.

The architecture consists of the following components:

- **Encoding layers**: Four convolutional blocks with increasing feature channels (64 → 512), capturing hierarchical features from the input image.
- **Bottleneck**: Captures high-level representations through deep convolutions and max-pooling.
- **Decoding layers**: Transposed convolutions for upsampling, followed by regular convolutions to refine the resolution.
- **Skip connections**: Features from encoder layers are added to decoder layers to preserve spatial information and improve convergence.
- **Output layer**: Produces a 3-channel RGB image with values in [0, 1] using a sigmoid activation.

```
class DeepUNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.enc1 = nn.Sequential(nn.Conv2d(3, 64, 3, padding=1), nn.ReLU())
        self.enc2 = nn.Sequential(nn.Conv2d(64, 128, 3, padding=1), nn.ReLU())
        self.enc3 = nn.Sequential(nn.Conv2d(128, 256, 3, padding=1), nn.ReLU())
        self.enc4 = nn.Sequential(nn.Conv2d(256, 512, 3, padding=1), nn.ReLU())
        self.pool = nn.MaxPool2d(2)

        self.up3 = nn.Sequential(nn.ConvTranspose2d(512, 256, 2, stride=2), nn.ReLU())
        self.dec3 = nn.Sequential(nn.Conv2d(256, 256, 3, padding=1), nn.ReLU())
        self.up2 = nn.Sequential(nn.ConvTranspose2d(256, 128, 2, stride=2), nn.ReLU())
        self.dec2 = nn.Sequential(nn.Conv2d(128, 128, 3, padding=1), nn.ReLU())
        self.up1 = nn.Sequential(nn.ConvTranspose2d(128, 64, 2, stride=2), nn.ReLU())
        self.dec1 = nn.Sequential(nn.Conv2d(64, 64, 3, padding=1), nn.ReLU())

        self.out_conv = nn.Sequential(nn.Conv2d(64, 3, 3, padding=1), nn.Sigmoid())

    def forward(self, x):
        e1 = self.enc1(x)
        e2 = self.enc2(self.pool(e1))
        e3 = self.enc3(self.pool(e2))
        e4 = self.enc4(self.pool(e3))

        d3 = self.dec3(self.up3(e4)) + e3
        d2 = self.dec2(self.up2(d3)) + e2
        d1 = self.dec1(self.up1(d2)) + e1
        return self.out_conv(d1)

model = DeepUNet().to(device)
```

6. Model Training with Early Stopping

We train the model using Mean Squared Error (MSE) loss and the Adam optimizer. Early stopping halts training if validation loss does not improve.

```
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
loss_fn = nn.MSELoss()

best_loss = float('inf')
patience = 5
wait = 0

print("[Training begins]")
for epoch in range(50):
    total_loss = 0
    pbar = tqdm(train_loader, desc=f"Epoch {epoch+1}")
    for blurred, sharp in pbar:
        blurred, sharp = blurred.to(device), sharp.to(device)
        pred = model(blurred)
        loss = loss_fn(pred, sharp)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        pbar.set_postfix({"Batch Loss": loss.item()})

    mean_loss = total_loss / len(train_loader)
    print(f"Epoch {epoch+1}, Mean Loss: {mean_loss:.4f}")

    if mean_loss < best_loss:
        best_loss = mean_loss
        wait = 0
        torch.save(model.state_dict(), "best_deep_unet_512.pth")
        print("[Info] Best model saved.")
    else:
        wait += 1
        if wait >= patience:
            print("[Early stopping] No improvement for several epochs.")
            break
```

7. Load Trained Model

We reload the model weights from the best checkpoint saved during training. This ensures we evaluate using the best version of the network.

```
model = DeepUNet().to(device)
model.load_state_dict(torch.load("best_deep_unet_512.pth", map_location=device))
```

```
<All keys matched successfully>
```

8. Quantitative Evaluation (PSNR & SSIM)

We compute two image quality metrics on the test set:

- **PSNR** (Peak Signal-to-Noise Ratio): Measures pixel-level reconstruction quality.
- **SSIM** (Structural Similarity Index): Measures structural similarity between images.

```
model.eval()
total_psnr = 0
total_ssim = 0
count = 0

with torch.no_grad():
    for blurred, sharp in tqdm(test_loader, desc="Evaluation"):
        blurred, sharp = blurred.to(device), sharp.to(device)
        pred = model(blurred).clamp(0, 1)

        for i in range(pred.size(0)):
            pred_img = pred[i].cpu().permute(1, 2, 0).numpy()
            sharp_img = sharp[i].cpu().permute(1, 2, 0).numpy()

            total_psnr += psnr(sharp_img, pred_img, data_range=1.0)
            total_ssim += ssim(sharp_img, pred_img, data_range=1.0, channel_axis=2)
            count += 1

print(f"Average PSNR on test set: {total_psnr / count:.2f} dB")
print(f"Average SSIM on test set: {total_ssim / count:.4f}")
```

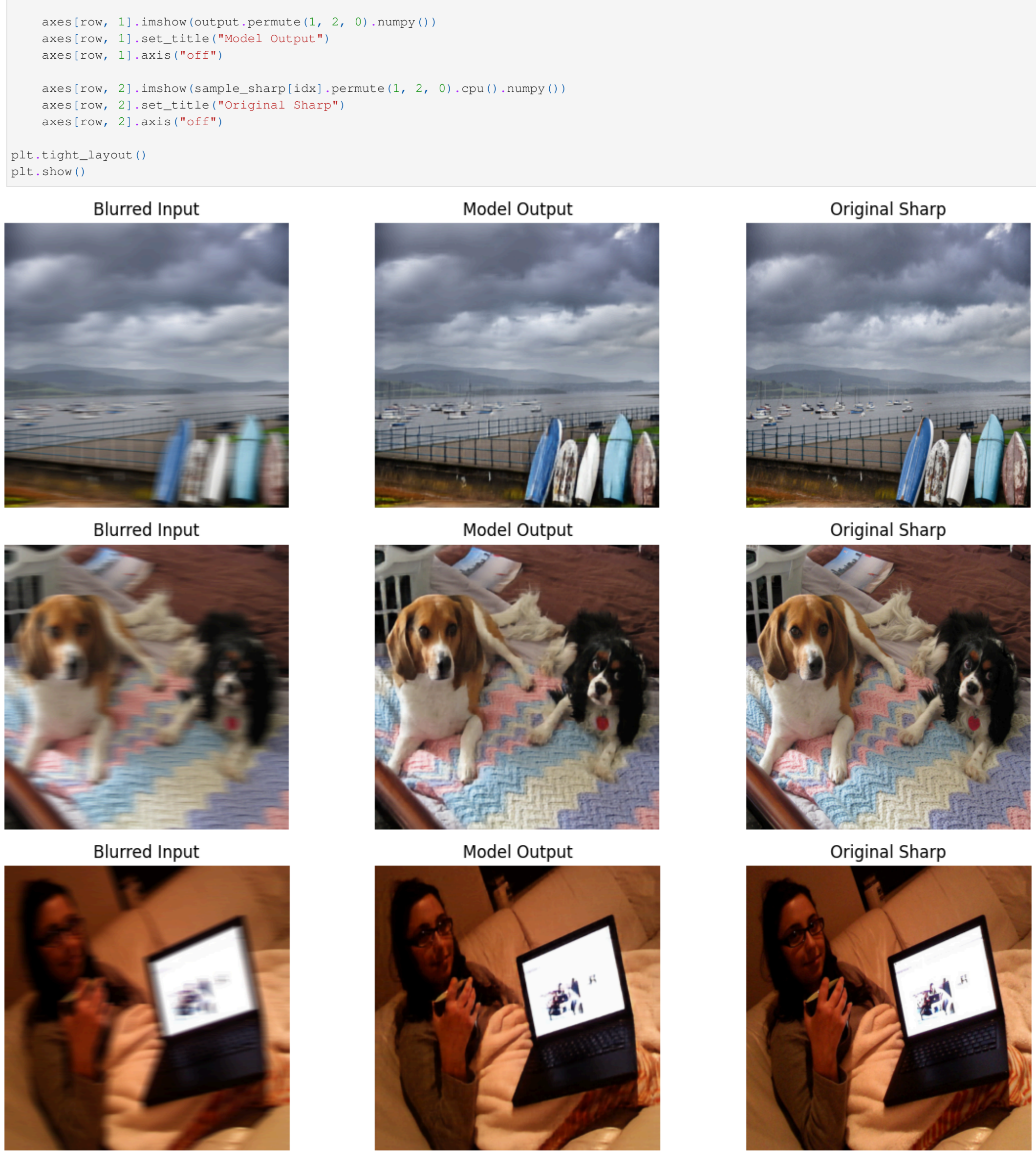
```
Evaluation: 100%|██████████| 63/63 [00:42<00:00,  1.48it/s]
Average PSNR on test set: 28.10 dB
Average SSIM on test set: 0.8080
```

9. Visual Evaluation

We visualize three randomly selected test samples, each showing:

- The blurred input
- The model's deblurred output
- The original sharp image

```
sample_blur, sample_sharp = next(iter(test_loader))
random_indices = random.sample(range(len(sample_blur)), 3)

fig, axes = plt.subplots(3, 3, figsize=(12, 10))

for row, idx in enumerate(random_indices):
    input_tensor = sample_blur[idx].unsqueeze(0).to(device)
    with torch.no_grad():
        output = model(input_tensor).squeeze(0).cpu().clamp(0, 1)

    axes[row, 0].imshow(sample_blur[idx].permute(1, 2, 0).cpu().numpy())
    axes[row, 0].set_title("Blurred Input")
    axes[row, 0].axis("off")

    axes[row, 1].imshow(output.permute(1, 2, 0).numpy())
    axes[row, 1].set_title("Model Output")
    axes[row, 1].axis("off")

    axes[row, 2].imshow(sample_sharp[idx].permute(1, 2, 0).cpu().numpy())
    axes[row, 2].set_title("Original Sharp")
    axes[row, 2].axis("off")

plt.tight_layout()
plt.show()
```



10. Inference on External Images

This section demonstrates how to test the trained model using a blurred image fetched directly from the web. The image is resized to 512x512 and passed through the model for deblurring.

```
# URL of a blurred image
image_url = "https://www.slrlounge.com/wp-content/uploads/2014/12/fix-blurry-photos.jpg"

# Download the image with custom headers
headers = {"User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)"}
response = requests.get(image_url, headers=headers)
image = Image.open(BytesIO(response.content)).convert("RGB")
image = image.resize((512, 512))

# Preprocess and inference
image_tensor = transform(image).unsqueeze(0).to(device)
model.eval()
with torch.no_grad():
    output = model(image_tensor).squeeze(0).cpu().clamp(0, 1)

# Display input and output
fig, axes = plt.subplots(1, 2, figsize=(12, 6))

axes[0].imshow(image_tensor.squeeze(0).permute(1, 2, 0).cpu())
axes[0].set_title("Blurred Input (from URL)")
axes[0].axis("off")

axes[1].imshow(output.permute(1, 2, 0).numpy())
axes[1].set_title("Model Output")
axes[1].axis("off")

plt.tight_layout()
plt.show()
```

11. Conclusion

This project successfully demonstrates how to train a DeepUNet-based convolutional neural network for image deblurring using a synthetic dataset. Despite being trained locally over approximately 7 hours on a portable machine (MacBook Pro), the model achieved **an average PSNR of 28.10 dB** and **SSIM of 0.8080**, which indicates that the reconstructed images preserve good structural and visual quality.

These results are acceptable for a lightweight implementation and show that a compact model can effectively restore image quality under limited hardware constraints.

Possible Improvements:

- **Model Architecture**: Incorporating residual connections or switching to more powerful architectures like ResNet-based U-Nets could further boost performance.
- **Training Time**: Longer training with more epochs and larger datasets could help generalization.
- **Data Augmentation**: Introducing more diverse blur types and lighting conditions may help the model adapt to real-world scenarios.
- **Loss Functions**: Exploring perceptual losses or SSIM-based losses in addition to MSE could improve visual quality.

Overall, this project demonstrates the feasibility of implementing an effective deblurring pipeline using deep learning with limited resources.

12. References

- COCO Dataset – https://cocodataset.org
- U-Net Paper – https://arxiv.org/abs/1505.04597
- PyTorch – https://pytorch.org
- scikit-image – https://scikit-image.org