**PROGRAM CODE FOR IMAGE SHARPENING USING KNOWLEDGE DISTILLATION**

# Program2 for Teacher and Student model

#Step 1: Mount Google Drive

import os

from google.colab import drive

# drive.mount('/content/gdrive/') # Already mounted

# Program2 for Teacher and Student model

#Step 1: Mount Google Drive

import os

from google.colab import drive

# drive.mount('/content/gdrive/') # Already mounted

# Step 2: Define paths from Google Drive (update if needed)

# Paths were updated in the previous step.

# Step 3: Check if directories exist

if not os.path.exists(blur\_path):

raise FileNotFoundError(f"Blur folder not found at: {blur\_path}")

if not os.path.exists(sharp\_path):

raise FileNotFoundError(f"Sharp folder not found at: {sharp\_path}")

if not os.path.exists(blur\_img\_path):

raise FileNotFoundError(f"Test Blur folder not found at: {blur\_img\_path}")

if not os.path.exists(sharp\_img\_path):

raise FileNotFoundError(f"Test Sharp folder not found at: {sharp\_img\_path}")

# Step 4: Required Libraries

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import Dataset, DataLoader

import torchvision.transforms as transforms

from PIL import Image

import numpy as np

import matplotlib.pyplot as plt

# Dataset class

class BlurSharpDataset(Dataset):

def \_\_init\_\_(self, blur\_dir, sharp\_dir, transform=None):

self.blur\_files = sorted(os.listdir(blur\_dir))

self.sharp\_files = sorted(os.listdir(sharp\_dir))

self.blur\_dir = blur\_dir

self.sharp\_dir = sharp\_dir

self.transform = transform

def \_\_len\_\_(self):

return len(self.blur\_files)

def \_\_getitem\_\_(self, idx):

blur\_image = Image.open(os.path.join(self.blur\_dir, self.blur\_files[idx])).convert("RGB")

sharp\_image = Image.open(os.path.join(self.sharp\_dir, self.sharp\_files[idx])).convert("RGB")

if self.transform:

blur\_image = self.transform(blur\_image)

sharp\_image = self.transform(sharp\_image)

return blur\_image, sharp\_image

# Simple CNN model (Teacher)

class SharpeningCNN(nn.Module):

def \_\_init\_\_(self):

super(SharpeningCNN, self).\_\_init\_\_()

self.encoder = nn.Sequential(

nn.Conv2d(3, 64, 3, padding=1), nn.ReLU(),

nn.Conv2d(64, 64, 3, padding=1), nn.ReLU()

)

self.decoder = nn.Sequential(

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

)

def forward(self, x):

features = self.encoder(x)

output = self.decoder(features)

return output, features

# Simple CNN model (Student - smaller than teacher)

class StudentCNN(nn.Module):

def \_\_init\_\_(self):

super(StudentCNN, self).\_\_init\_\_()

self.encoder = nn.Sequential(

nn.Conv2d(3, 32, 3, padding=1), nn.ReLU(),

nn.Conv2d(32, 32, 3, padding=1), nn.ReLU()

)

self.decoder = nn.Sequential(

nn.Conv2d(32, 3, 3, padding=1), nn.Sigmoid()

)

def forward(self, x):

features = self.encoder(x)

output = self.decoder(features)

return output, features

# Knowledge Distillation Loss

def kd\_loss(student\_output, teacher\_output, sharp\_image, alpha=0.5, temperature=1.0):

mse = nn.MSELoss()

reconstruction\_loss = mse(student\_output, sharp\_image)

# Knowledge distillation loss (using features or outputs)

# You might want to adjust this based on your distillation strategy

feature\_distill\_loss = mse(student\_output, teacher\_output.detach())

# Or if you are using output logits with temperature:

# teacher\_logits = teacher\_output / temperature

# student\_logits = student\_output / temperature

# feature\_distill\_loss = nn.KLDivLoss()(F.log\_softmax(student\_logits, dim=1),

# F.softmax(teacher\_logits, dim=1))

print(f"Reconstruction Loss: {reconstruction\_loss.item():.6f}")

print(f"Feature Distillation Loss: {feature\_distill\_loss.item():.6f}")

print("-" \* 40)

return alpha \* reconstruction\_loss + (1 - alpha) \* feature\_distill\_loss

# Training function

def train\_model(blur\_dir, sharp\_dir, model, teacher\_model=None, epochs=5, batch\_size=2, learning\_rate=1e-3, alpha=0.5, temperature=1.0):

transform = transforms.Compose([

transforms.Resize((128, 128)),

transforms.ToTensor()

])

dataset = BlurSharpDataset(blur\_dir, sharp\_dir, transform)

dataloader = DataLoader(dataset, batch\_size=batch\_size, shuffle=True)

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

model.to(device)

if teacher\_model:

teacher\_model.to(device)

teacher\_model.eval() # Freeze teacher model

optimizer = optim.Adam(model.parameters(), lr=learning\_rate)

for epoch in range(epochs):

model.train()

total\_loss = 0

for blur, sharp in dataloader:

blur, sharp = blur.to(device), sharp.to(device)

if teacher\_model: # Training student with knowledge distillation

with torch.no\_grad():

teacher\_out, teacher\_feat = teacher\_model(blur)

student\_out, student\_feat = model(blur)

loss = kd\_loss(student\_out, teacher\_out, sharp, alpha, temperature)

else: # Training teacher model

output, features = model(blur)

# Using MSE loss for teacher as a baseline, or modify kd\_loss for teacher if needed

loss = nn.MSELoss()(output, sharp) # Simple MSE for teacher training

optimizer.zero\_grad()

loss.backward()

optimizer.step()

total\_loss += loss.item()

avg\_loss = total\_loss / len(dataloader)

print(f"Epoch {epoch+1}/{epochs}, Loss: {avg\_loss:.4f}")

return model

# PSNR evaluation

def psnr(img1, img2):

mse = torch.mean((img1 - img2) \*\* 2)

if mse == 0:

return float('inf')

return 20 \* torch.log10(1.0 / torch.sqrt(mse))

def evaluate\_model(model, blur\_dir, sharp\_dir):

transform = transforms.Compose([

transforms.Resize((128, 128)),

transforms.ToTensor()

])

dataset = BlurSharpDataset(blur\_dir, sharp\_dir, transform)

dataloader = DataLoader(dataset, batch\_size=1)

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

model.eval()

psnr\_total = 0

with torch.no\_grad():

for blur, sharp in dataloader:

blur, sharp = blur.to(device), sharp.to(device)

output, \_ = model(blur)

psnr\_val = psnr(output, sharp)

psnr\_total += psnr\_val

avg\_psnr = psnr\_total / len(dataloader)

print(f"Average PSNR: {avg\_psnr:.2f} dB")

# Run training and evaluation

print("Training Teacher Model...")

teacher\_model = train\_model(blur\_path, sharp\_path, model=SharpeningCNN(), epochs=5, batch\_size=2, learning\_rate=1e-3)

torch.save(teacher\_model.state\_dict(), "teacher\_model.pth")

print("Evaluating Teacher Model...")

evaluate\_model(teacher\_model, blur\_path, sharp\_path)

print("\nTraining Student Model with Knowledge Distillation...")

student\_model = train\_model(blur\_path, sharp\_path, model=StudentCNN(), teacher\_model=teacher\_model, epochs=5, batch\_size=2, learning\_rate=1e-3, alpha=0.5, temperature=1.0)

torch.save(student\_model.state\_dict(), "student\_model.pth")

print("Evaluating Student Model...")

evaluate\_model(student\_model, blur\_path, sharp\_path)

# Evaluate

def show\_result(input\_img, output\_img, target\_img):

input\_img = input\_img.permute(1, 2, 0).cpu().numpy()

output\_img = output\_img.permute(1, 2, 0).detach().cpu().numpy()

target\_img = target\_img.permute(1, 2, 0).cpu().numpy()

plt.figure(figsize=(12,4))

plt.subplot(1, 3, 1); plt.imshow(input\_img); plt.title("Training\_Blurred")

plt.subplot(1, 3, 2); plt.imshow(output\_img); plt.title("Training\_Student Output")

plt.subplot(1, 3, 3); plt.imshow(target\_img); plt.title("Training\_Sharp Target")

plt.show()

# Prepare loader and student model

transform = transforms.Compose([

transforms.Resize((128, 128)),

transforms.ToTensor()

])

visual\_dataset = BlurSharpDataset(blur\_path, sharp\_path, transform)

visual\_loader = DataLoader(visual\_dataset, batch\_size=1, shuffle=True)

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

student\_model.eval()

with torch.no\_grad():

for blur, sharp in visual\_loader:

blur, sharp = blur.to(device), sharp.to(device)

out, \_ = student\_model(blur) # include the \_ because model returns two outputs

show\_result(blur[0], out[0], sharp[0])

break # show one result

# Testing Phase

# Step 1: Make sure the model is in evaluation mode

student\_model.eval()

# Step 2: Define the same transform as used in training

transform = transforms.Compose([

transforms.Resize((256, 256)),

transforms.ToTensor()

# Include normalization here if used during training:

# transforms.Normalize(mean=[0.5]\*3, std=[0.5]\*3)

])

# Step 3: Set the paths for test images (blurred + sharp)

# Paths were updated in the previous step.

# Step 4: Load and transform both images

blur\_img = Image.open(os.path.join(blur\_img\_path, os.listdir(blur\_img\_path)[0])).convert('RGB')

sharp\_img = Image.open(os.path.join(sharp\_img\_path, os.listdir(sharp\_img\_path)[0])).convert('RGB')

blur\_tensor = transform(blur\_img).unsqueeze(0).to(device) # shape: [1, 3, 256, 256]

sharp\_tensor = transform(sharp\_img).unsqueeze(0).to(device)

# Step 5: Pass through model

with torch.no\_grad():

output\_tensor, \_ = student\_model(blur\_tensor) # your model returns two outputs

# Knowledge Distillation Loss

def kd\_loss(student\_output, teacher\_output, sharp\_image, alpha=0.5, temperature=1.0):

mse = nn.MSELoss()

reconstruction\_loss = mse(student\_output, sharp\_image)

# Knowledge distillation loss (using features or outputs)

# You might want to adjust this based on your distillation strategy

feature\_distill\_loss = mse(student\_output, teacher\_output.detach())

# Or if you are using output logits with temperature:

# teacher\_logits = teacher\_output / temperature

# student\_logits = student\_output / temperature

# feature\_distill\_loss = nn.KLDivLoss()(F.log\_softmax(student\_logits, dim=1),

# F.softmax(teacher\_logits, dim=1))

print(f"Reconstruction Loss: {reconstruction\_loss.item():.6f}")

print(f"Feature Distillation Loss: {feature\_distill\_loss.item():.6f}")

print("-" \* 40)

return alpha \* reconstruction\_loss + (1 - alpha) \* feature\_distill\_loss

# Step 6: Display results

def show\_result(input\_img, output\_img, target\_img):

input\_img = input\_img.permute(1, 2, 0).cpu().numpy()

output\_img = output\_img.permute(1, 2, 0).detach().cpu().numpy()

target\_img = target\_img.permute(1, 2, 0).cpu().numpy()

plt.figure(figsize=(12, 4))

plt.subplot(1, 3, 1); plt.imshow(input\_img); plt.title("Testing Blurred Input")

plt.subplot(1, 3, 2); plt.imshow(output\_img); plt.title("Testing Student Output")

plt.subplot(1, 3, 3); plt.imshow(target\_img); plt.title("Testing Ground Truth Sharp")

plt.show()

show\_result(blur\_tensor[0], output\_tensor[0], sharp\_tensor[0])

# Step 4: Required Libraries

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import Dataset, DataLoader

import torchvision.transforms as transforms

from PIL import Image

import numpy as np

import matplotlib.pyplot as plt

# Dataset class

class BlurSharpDataset(Dataset):

def \_\_init\_\_(self, blur\_dir, sharp\_dir, transform=None):

self.blur\_files = sorted(os.listdir(blur\_dir))

self.sharp\_files = sorted(os.listdir(sharp\_dir))

self.blur\_dir = blur\_dir

self.sharp\_dir = sharp\_dir

self.transform = transform

def \_\_len\_\_(self):

return len(self.blur\_files)

def \_\_getitem\_\_(self, idx):

blur\_image = Image.open(os.path.join(self.blur\_dir, self.blur\_files[idx])).convert("RGB")

sharp\_image = Image.open(os.path.join(self.sharp\_dir, self.sharp\_files[idx])).convert("RGB")

if self.transform:

blur\_image = self.transform(blur\_image)

sharp\_image = self.transform(sharp\_image)

return blur\_image, sharp\_image

# Simple CNN model (Teacher)

class SharpeningCNN(nn.Module):

def \_\_init\_\_(self):

super(SharpeningCNN, self).\_\_init\_\_()

self.encoder = nn.Sequential(

nn.Conv2d(3, 64, 3, padding=1), nn.ReLU(),

nn.Conv2d(64, 64, 3, padding=1), nn.ReLU()

)

self.decoder = nn.Sequential(

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

)

def forward(self, x):

features = self.encoder(x)

output = self.decoder(features)

return output, features

# Simple CNN model (Student - smaller than teacher)

class StudentCNN(nn.Module):

def \_\_init\_\_(self):

super(StudentCNN, self).\_\_init\_\_()

self.encoder = nn.Sequential(

nn.Conv2d(3, 32, 3, padding=1), nn.ReLU(),

nn.Conv2d(32, 32, 3, padding=1), nn.ReLU()

)

self.decoder = nn.Sequential(

nn.Conv2d(32, 3, 3, padding=1), nn.Sigmoid()

)

def forward(self, x):

features = self.encoder(x)

output = self.decoder(features)

return output, features

# Knowledge Distillation Loss

def kd\_loss(student\_output, teacher\_output, sharp\_image, alpha=0.5, temperature=1.0):

mse = nn.MSELoss()

reconstruction\_loss = mse(student\_output, sharp\_image)

# Knowledge distillation loss (using features or outputs)

# You might want to adjust this based on your distillation strategy

feature\_distill\_loss = mse(student\_output, teacher\_output.detach())

# Or if you are using output logits with temperature:

# teacher\_logits = teacher\_output / temperature

# student\_logits = student\_output / temperature

# feature\_distill\_loss = nn.KLDivLoss()(F.log\_softmax(student\_logits, dim=1),

# F.softmax(teacher\_logits, dim=1))

print(f"Reconstruction Loss: {reconstruction\_loss.item():.6f}")

print(f"Feature Distillation Loss: {feature\_distill\_loss.item():.6f}")

print("-" \* 40)

return alpha \* reconstruction\_loss + (1 - alpha) \* feature\_distill\_loss

# Training function

def train\_model(blur\_dir, sharp\_dir, model, teacher\_model=None, epochs=5, batch\_size=2, learning\_rate=1e-3, alpha=0.5, temperature=1.0):

transform = transforms.Compose([

transforms.Resize((128, 128)),

transforms.ToTensor()

])

dataset = BlurSharpDataset(blur\_dir, sharp\_dir, transform)

dataloader = DataLoader(dataset, batch\_size=batch\_size, shuffle=True)

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

model.to(device)

if teacher\_model:

teacher\_model.to(device)

teacher\_model.eval() # Freeze teacher model

optimizer = optim.Adam(model.parameters(), lr=learning\_rate)

for epoch in range(epochs):

model.train()

total\_loss = 0

for blur, sharp in dataloader:

blur, sharp = blur.to(device), sharp.to(device)

if teacher\_model: # Training student with knowledge distillation

with torch.no\_grad():

teacher\_out, teacher\_feat = teacher\_model(blur)

student\_out, student\_feat = model(blur)

loss = kd\_loss(student\_out, teacher\_out, sharp, alpha, temperature)

else: # Training teacher model

output, features = model(blur)

# Using MSE loss for teacher as a baseline, or modify kd\_loss for teacher if needed

loss = nn.MSELoss()(output, sharp) # Simple MSE for teacher training

optimizer.zero\_grad()

loss.backward()

optimizer.step()

total\_loss += loss.item()

avg\_loss = total\_loss / len(dataloader)

print(f"Epoch {epoch+1}/{epochs}, Loss: {avg\_loss:.4f}")

return model

# PSNR evaluation

def psnr(img1, img2):

mse = torch.mean((img1 - img2) \*\* 2)

if mse == 0:

return float('inf')

return 20 \* torch.log10(1.0 / torch.sqrt(mse))

def evaluate\_model(model, blur\_dir, sharp\_dir):

transform = transforms.Compose([

transforms.Resize((128, 128)),

transforms.ToTensor()

])

dataset = BlurSharpDataset(blur\_dir, sharp\_dir, transform)

dataloader = DataLoader(dataset, batch\_size=1)

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

model.eval()

psnr\_total = 0

with torch.no\_grad():

for blur, sharp in dataloader:

blur, sharp = blur.to(device), sharp.to(device)

output, \_ = model(blur)

psnr\_val = psnr(output, sharp)

psnr\_total += psnr\_val

avg\_psnr = psnr\_total / len(dataloader)

print(f"Average PSNR: {avg\_psnr:.2f} dB")

# Run training and evaluation

print("Training Teacher Model with knowledge Distillation...")

teacher\_model = train\_model(blur\_path, sharp\_path, model=SharpeningCNN(), epochs=5, batch\_size=2, learning\_rate=1e-3)

torch.save(teacher\_model.state\_dict(), "teacher\_model.pth")

print("Evaluating Teacher Model...")

evaluate\_model(teacher\_model, blur\_path, sharp\_path)

print("\nTraining Student Model with Knowledge Distillation...")

student\_model = train\_model(blur\_path, sharp\_path, model=StudentCNN(), teacher\_model=teacher\_model, epochs=5, batch\_size=2, learning\_rate=1e-3, alpha=0.5, temperature=1.0)

torch.save(student\_model.state\_dict(), "student\_model.pth")

print("Evaluating Student Model...")

evaluate\_model(student\_model, blur\_path, sharp\_path)

# Evaluate

def show\_result(input\_img, output\_img, target\_img):

input\_img = input\_img.permute(1, 2, 0).cpu().numpy()

output\_img = output\_img.permute(1, 2, 0).detach().cpu().numpy()

target\_img = target\_img.permute(1, 2, 0).cpu().numpy()

plt.figure(figsize=(12,4))

plt.subplot(1, 3, 1); plt.imshow(input\_img); plt.title("Training\_Blurred")

plt.subplot(1, 3, 2); plt.imshow(output\_img); plt.title("Training\_Student Output")

plt.subplot(1, 3, 3); plt.imshow(target\_img); plt.title("Training\_Sharp Target")

plt.show()

# Prepare loader and student model

transform = transforms.Compose([

transforms.Resize((128, 128)),

transforms.ToTensor()

])

visual\_dataset = BlurSharpDataset(blur\_path, sharp\_path, transform)

visual\_loader = DataLoader(visual\_dataset, batch\_size=1, shuffle=True)

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

student\_model.eval()

with torch.no\_grad():

for blur, sharp in visual\_loader:

blur, sharp = blur.to(device), sharp.to(device)

out, \_ = student\_model(blur) # include the \_ because model returns two outputs

show\_result(blur[0], out[0], sharp[0])

break # show one result

# Testing Phase

# Step 1: Make sure the model is in evaluation mode

student\_model.eval()

# Step 2: Define the same transform as used in training

transform = transforms.Compose([

transforms.Resize((256, 256)),

transforms.ToTensor()

# Include normalization here if used during training:

# transforms.Normalize(mean=[0.5]\*3, std=[0.5]\*3)

])

# Step 3: Set the paths for test images (blurred + sharp)

# Paths were updated in the previous step.

# Step 4: Load and transform both images

blur\_img = Image.open(os.path.join(blur\_img\_path, os.listdir(blur\_img\_path)[0])).convert('RGB')

sharp\_img = Image.open(os.path.join(sharp\_img\_path, os.listdir(sharp\_img\_path)[0])).convert('RGB')

blur\_tensor = transform(blur\_img).unsqueeze(0).to(device) # shape: [1, 3, 256, 256]

sharp\_tensor = transform(sharp\_img).unsqueeze(0).to(device)

# Step 5: Pass through model

with torch.no\_grad():

output\_tensor, \_ = student\_model(blur\_tensor) # your model returns two outputs

# Knowledge Distillation Loss

def kd\_loss(student\_output, teacher\_output, sharp\_image, alpha=0.5, temperature=1.0):

mse = nn.MSELoss()

reconstruction\_loss = mse(student\_output, sharp\_image)

# Knowledge distillation loss (using features or outputs)

# You might want to adjust this based on your distillation strategy

feature\_distill\_loss = mse(student\_output, teacher\_output.detach())

# Or if you are using output logits with temperature:

# teacher\_logits = teacher\_output / temperature

# student\_logits = student\_output / temperature

# feature\_distill\_loss = nn.KLDivLoss()(F.log\_softmax(student\_logits, dim=1),

# F.softmax(teacher\_logits, dim=1))

print(f"Reconstruction Loss: {reconstruction\_loss.item():.6f}")

print(f"Feature Distillation Loss: {feature\_distill\_loss.item():.6f}")

print("-" \* 40)

return alpha \* reconstruction\_loss + (1 - alpha) \* feature\_distill\_loss

# Step 6: Display results

def show\_result(input\_img, output\_img, target\_img):

input\_img = input\_img.permute(1, 2, 0).cpu().numpy()

output\_img = output\_img.permute(1, 2, 0).detach().cpu().numpy()

target\_img = target\_img.permute(1, 2, 0).cpu().numpy()

plt.figure(figsize=(12, 4))

plt.subplot(1, 3, 1); plt.imshow(input\_img); plt.title("Testing Blurred Input")

plt.subplot(1, 3, 2); plt.imshow(output\_img); plt.title("Testing Student Output")

plt.subplot(1, 3, 3); plt.imshow(target\_img); plt.title("Testing Ground Truth Sharp")

plt.show()

show\_result(blur\_tensor[0], output\_tensor[0], sharp\_tensor[0])