Lab 2c: ADMM Inpainting with Explicit Regularizers

Advanced Image Processing

Goal. Implement ADMM for pixel-wise inpainting using four different regularizers:

- Plug-and-Play with DRUNet
- Total Variation (TV) prox
- Tikhonov regularization with identity operator
- Tikhonov regularization with gradient operator

Compare convergence and reconstruction quality across these choices.

Dependencies. PyTorch, torchvision, matplotlib, scikit-image, PIL, optional `lpips`. If needed: `pip install scikit-image lpips deepinv`.

Block 0: Setup and data

```
import os, math, random, numpy as np
import torch, torch.nn.functional as F
import torchvision.transforms as T
import matplotlib.pyplot as plt
from skimage import data
from PIL import Image

OUTDIR = "results_lab2b"; os.makedirs(OUTDIR, exist_ok=True)
SEED = 12345
device = "cuda" if torch.cuda.is_available() else "cpu"

p_mask = 20.0   # % pixels masked
noise_sigma = 0.00
rho, iters = 0.2, 50

reg_cfgs = [
    dict(name="drunet",        lam=0.08, tv_iters=0),
    dict(name="tv",            lam=0.05, tv_iters=60),
    dict(name="tikh_identity", lam=0.01, tv_iters=0),
    dict(name="tikh_grad",     lam=0.5,  tv_iters=0),
]

# Reproducibility
random.seed(SEED); np.random.seed(SEED); torch.manual_seed(SEED)
if torch.cuda.is_available(): torch.cuda.manual_seed_all(SEED)

# Load sample image
def load_rgb01():
    img = Image.fromarray(data.astronaut())
    return T.ToTensor()(img).unsqueeze(0).to(device)

x_clean = load_rgb01()
B,C,H,W = x_clean.shape
N = H*W
```

Block 1: Inpainting mask and observations

```
# Random pixel mask
num_drop = int(p_mask/100.0 * N)
idx = np.arange(N); np.random.shuffle(idx)
drop = idx[:num_drop]
M = np.ones((N,), dtype=np.float32); M[drop] = 0.0
M = torch.from_numpy(M.reshape(H, W)).to(device)
M3 = M.view(1,1,H,W).repeat(1,C,1,1)

# Observed image
y = (x_clean + noise_sigma*torch.randn_like(x_clean)).clamp(0,1) if noise_sigma>0 else
    x_clean.clone()
y = y * M3
```

Block 2: Regularizers

DRUNet (PnP).

```
from deepinv.models import DRUNet
DRUNET = DRUNet(pretrained="download", device=device).eval()

def pnp_drunet(x, sigma):
    with torch.no_grad():
        return DRUNET(x.clamp(0,1), float(sigma)).clamp(0,1)
```

TV prox (Chambolle).

```
def grad_periodic(u):
    dx = torch.roll(u, shifts=-1, dims=-1) - u
    dy = torch.roll(u, shifts=-1, dims=-2) - u
    return dx, dy

def div_periodic(px, py):
    dxT = px - torch.roll(px, shifts=1, dims=-1)
    dyT = py - torch.roll(py, shifts=1, dims=-2)
    return dxT + dyT

def prox_tv_chambolle(x, weight, iters=60, tau=0.25):
    if weight <= 0: return x
    p1 = torch.zeros_like(x); p2 = torch.zeros_like(x)
    for _ in range(iters):
        divp = div_periodic(p1, p2)
        u = divp - x/weight
        ux, uy = grad_periodic(u)
        p1_new, p2_new = p1 + tau*ux, p2 + tau*uy
        norm = torch.maximum(torch.ones_like(p1_new),
                             torch.sqrt(p1_new**2+p2_new**2))
        p1, p2 = p1_new/norm, p2_new/norm
    return (x - weight*div_periodic(p1, p2)).clamp(0,1)
```

Tikhonov prox (identity).

```
def prox_tikh_identity(x, lam_over_rho):
    return (1.0/(1.0+lam_over_rho))*x
```

Tikhonov prox (gradient, FFT).

```
_fft_cache = {}
def _fft_denom(H, W, tau, device, dtype):
    key = (H,W,float(tau),device,str(dtype))
    if key in _fft_cache: return _fft_cache[key]
    wy = 2*math.pi*torch.fft.fftfreq(H, d=1.0, device=device, dtype=dtype).view(H,1)
    wx = 2*math.pi*torch.fft.fftfreq(W, d=1.0, device=device, dtype=dtype).view(1,W)
    lap_spec = 4 - 2*torch.cos(wy) - 2*torch.cos(wx)
    denom = 1.0 + tau*lap_spec
    _fft_cache[key] = denom; return denom

def prox_tikh_grad_fft(x, lam_over_rho):
    if lam_over_rho <= 0: return x
    B,C,H,W = x.shape
    denom = _fft_denom(H,W,lam_over_rho,x.device,x.dtype)
    X = torch.fft.fft2(x)
    return torch.fft.ifft2(X/denom).real.clamp(0,1)
```

Block 3: ADMM solver

```
def psnr(ref, out, eps=1e-12):
    mse = F.mse_loss(out, ref)
    return float(10.0*torch.log10(1.0/(mse+eps)))

def admm_inpaint(y, M3, x_ref, reg_name, lam, rho, iters, tv_iters=60):
    x, z, u = y.clone(), y.clone(), torch.zeros_like(y)
    psnrs, r_hist, s_hist = [], [], []
    lam_over_rho = lam/rho
    for _ in range(iters):
        rhs = y + rho*(z - u); denom = M3 + rho
        x = (rhs/denom).clamp(0,1)
        x_plus_u = (x+u).clamp(0,1)
        if reg_name=="drunet":
            z_new = pnp_drunet(x_plus_u, sigma=lam)
        elif reg_name=="tv":
            z_new = prox_tv_chambolle(x_plus_u, lam_over_rho, tv_iters)
        elif reg_name=="tikh_identity":
            z_new = prox_tikh_identity(x_plus_u, lam_over_rho)
        elif reg_name=="tikh_grad":
            z_new = prox_tikh_grad_fft(x_plus_u, lam_over_rho)
        r = (x-z_new).reshape(-1); s = (rho*(z_new-z)).reshape(-1)
        r_hist.append(float(torch.norm(r))); s_hist.append(float(torch.norm(s)))
        z, u = z_new, u+x-z
        psnrs.append(psnr(x_ref, x))
    return x, dict(psnr_hist=psnrs, r_hist=r_hist, s_hist=s_hist)
```

Tasks

- Run ADMM inpainting with each regularizer (`drunet`, `tv`, `tikh_identity`, `tikh_grad`).
- Plot the PSNR vs iteration and primal/dual residuals for each case.
- Compare the reconstructed images. Which regularizers oversmooth? Which preserve edges?
- Discuss the effect of λ and ρ. How sensitive are TV and Tikhonov to parameter choice?
- Compare explicit regularization (TV/Tikhonov) with Plug-and-Play (DRUNet).