CSC2529-hw6

1010171181 Xinran Zhang

November 16, 2023

1 Task1

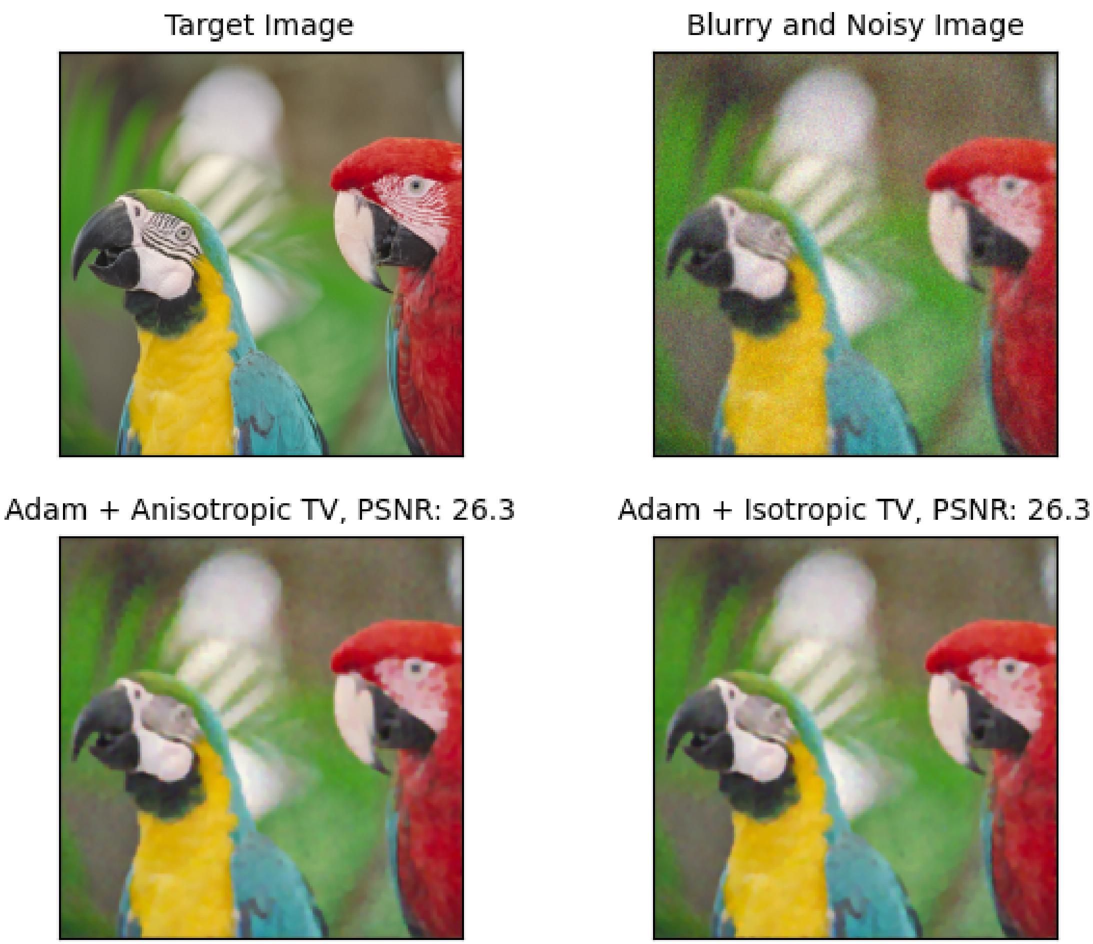

Figure 1: results for Task1

Theoretically,Anisotropic TV quantifies the image gradient, accentuating edge directions, whereas Isotropic TV characterizes the overall variation of an image without emphasizing specific directions. The Anisotropic TV term is good at preserving and enhancing distinct edges and features, effectively smoothing other regions. In contrast, the Isotropic TV term promotes uniform smoothness throughout the image, reducing noise while preserving underlying structures.

However, according to Figure 1, it is evident that the PSNR values for these two distinct methods are identical, implying that their performances are similar. If we round the PSNR results to three decimal places, we will notice slight differences between the results of the two methods, with the second method yielding slightly higher results.

	Anisotropic	Isotropic
PSNR	26.208	26.288

2 Task2

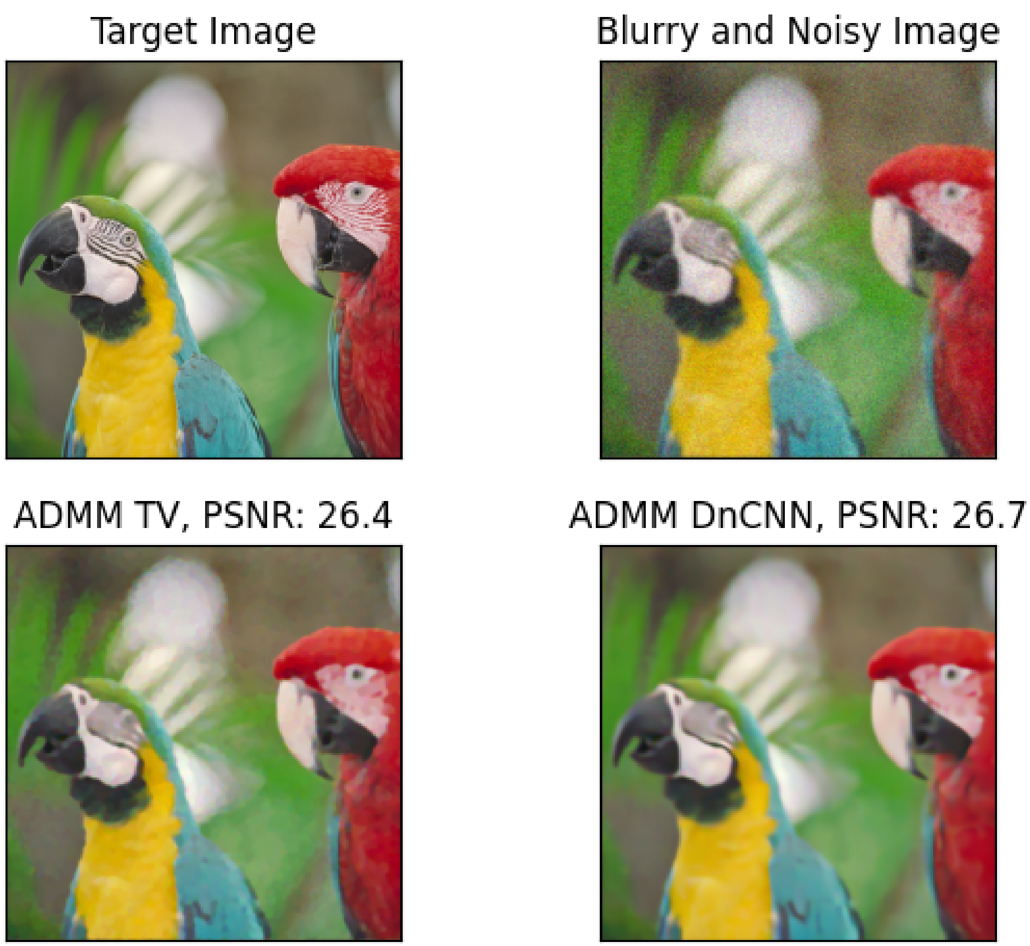

Figure 2: results for task2

3 Task3

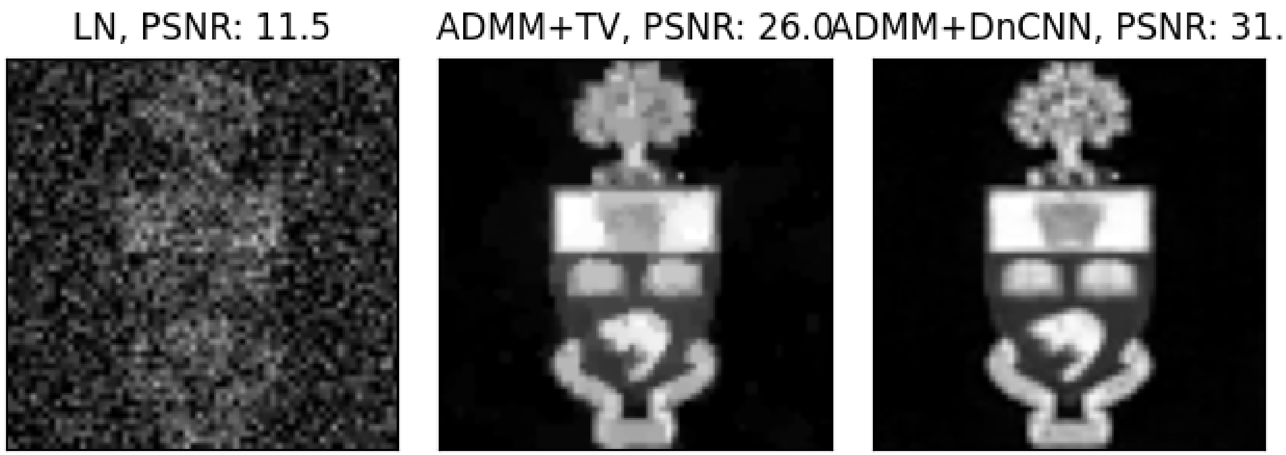

Figure 3: results for task3

	LN	ADMM+TV	ADMM+DnCNN
PSNR	11.5	26.0	31.9

A task1

```
def deconv_adam_tv(b, c, lam, num_iters, learning_rate=5e-2, anisotropic_tv=True):
    # check if GPU is available, otherwise use CPU
    device =torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

    # otf of blur kernel and forward image formation model
    cFT =psf2otf(c, np.shape(b))
    cFT =torch.from_numpy(cFT).to(device)
    Afun =lambda x: torch.real(torch.fft.ifft2(torch.fft.fft2(x) *cFT))

    # finite differences kernels and corresponding otfs
    dx =np.array([[-1., 1.]])
    dy =np.array([[-1.], [1.]])
    dxFT =torch.from_numpy(psf2otf(dx, b.shape)).to(device)
    dyFT =torch.from_numpy(psf2otf(dy, b.shape)).to(device)
    dxyFT =torch.stack((dxFT, dyFT), axis=0)

    # convert b to PyTorch tensor
    b = torch.from_numpy(b).to(device)
    # initialize x and convert to PyTorch tensor
    x = torch.zeros_like(b, requires_grad=True).to(device)
    # initialize Adam optimizer
    optim =torch.optim.Adam(params=[x], lr=learning_rate)

    ################# begin task 1 ###################################

    # Define function handle to compute horizontal and vertical gradients.
    # You can use a local function definition using Python's lamda function
    # or write your own function for this. Use the convolutional image
    # formation in the Fourier domain to implement this using dxFT, dyFT,
    # or dxyFT, as discussed in the lecture and in the problem session.

    grad_fn =lambda x: torch.real(torch.fft.ifft2(torch.fft.fft2(x) *dxyFT))

    ################# end task 1 ###################################

    for it in tqdm(range(num_iters)):

        # set all gradients of the computational graph to 0
        optim.zero_grad()

        # this term computes the data fidelity term of the loss function
        loss_data =(Afun(x) -b).pow(2).sum()

        ################# begin task 1 ###################################

        # Complete these parts by calling the grad_fn function, which should
        # give you a full-resolution tensor with the gradients in x and y.
        # Then aggregate these gradients into a single scalar, i.e., the
        # TV pseudo-norm here and store the result in loss_regularizer

        # anisotropic TV term
        if anisotropic_tv:
            loss_regularizer =torch.norm(grad_fn(x), 1) # you need to edit this, it's just a placeholder

        # isotropic TV term
        else:
            loss_regularizer =torch.norm(grad_fn(x), p=2, dim=0).sum() # you need to edit this, it's just a placeholder

        ################# end task 1 ###################################

        # compute weighted sum of data fidelity and regularization term
        loss =loss_data +lam *loss_regularizer

        # compute backwards pass
        loss.backward()

        # take a step with the Adam optimizer
        optim.step()
```

```
    # return the result as a numpy array
    return x.detach().cpu().numpy()
```

B task2

B.1 admm_tv

```
def deconv_admm_tv(b, c, lam, rho, num_iters, anisotropic_tv=False):
    # Blur kernel
    cFT = psf2otf(c, b.shape)
    cTFT = np.conj(cFT)

    # First differences
    dx = np.array([[-1., 1.]])
    dy = np.array([[-1.], [1.]])
    dxFT = psf2otf(dx, b.shape)
    dyFT = psf2otf(dy, b.shape)
    dxTFT = np.conj(dxFT)
    dyTFT = np.conj(dyFT)
    dxyFT = np.stack((dxFT, dyFT), axis=0)
    dxyTFT = np.stack((dxTFT, dyTFT), axis=0)

    # Fourier transform of b
    bFT = fft2(b)

    # initialize x,z,u with all zeros
    x = np.zeros_like(b)
    z = np.zeros((2, *b.shape))
    u = np.zeros((2, *b.shape))

    ################# begin task 2 ###################################
    # Complete these parts by first copying over the grad_fn you
    # implemented for task 1 here. What's important here (and wasn't
    # in task 1) is that grad_fn takes as input a 2D image of size [M N]
    # and outputs the horizontal and vertical gradients in a stack of
    # size [2 M N]! Please keep this in mind, otherwise the z-update
    # which is already implemented for you won't work

    # Then, you can pre-compute the denominator for the x-update
    # here, because that doesn't change unless rho changes, which is
    # not the case here

    # define function handle to compute horizontal and vertical gradients
    grad_fn = lambda x: (ifft2(fft2(x) * dxyFT)).real # you need to edit this, it's just a placeholder

    # precompute the denominator for the x-update
    # denom = (ifft2(cTFT * cFT + rho * (dxTFT * dxFT + dyTFT * dyFT))).real # you need to edit this, it's just a
                                                                             placeholder
    denom = cTFT * cFT + rho * (dxTFT * dxFT + dyTFT * dyFT)
    ################# end task 2 ###################################

    for it in tqdm(range(num_iters)):

        ################# begin task 2 ###################################

        # Complete this part by implementing the x-update discussed in
        # class and in the problem session. If you implemented the
        # denominator term above, you only need to compute the nominator
        # here as well as the rest of the x-update

        # x update - inverse filtering: Fourier multiplications and divisions
        # part2 = (ifft2(cTFT * bFT + rho * (dxTFT * fft2(v[0:, :, ]) + dyTFT * fft2(v[1:, :, ])))).real
        # part2 = (ifft2(cTFT * bFT + rho * np.sum(dxyTFT * fft2(z-u), axis=0))).real
        part2 = cTFT * bFT + rho * np.sum(dxyTFT * fft2(z - u), axis=0)
        x = ifft2(part2 / denom).real
        ################# end task 2 ###################################
```

```
        # z update - soft shrinkage
        kappa = lam / rho
        v = grad_fn(x) + u

        # proximal operator of anisotropic TV term
        if anisotropic_tv:
            z = np.maximum(1 - kappa / np.abs(v), 0) * v

        # proximal operator of isotropic TV term
        else:
            vnorm = np.sqrt(v[0, :, :]**2 + v[1, :, :]**2)
            z[0, :, :] = np.maximum(1 - kappa / vnorm, 0) * v[0, :, :]
            z[1, :, :] = np.maximum(1 - kappa / vnorm, 0) * v[1, :, :]

        # u-update
        u = u + grad_fn(x) - z

    return x
```

B.2 amdd_dncnn

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
#device = torch.device("cuda:0" if torch.cuda.is_available() else "mps" if torch.cuda.is_mps_supported else "cpu")
#device = torch.device("mps")
from network_dncnn import DnCNN as net


def deconv_admm_dncnn(b, c, lam, rho, num_iters):
    # Blur kernel
    cFT = psf2otf(c, b.shape)
    cTFT = np.conj(cFT)

    # Fourier transform of b
    bFT = fft2(b)

    # initialize x,z,u with all zeros
    x = np.zeros_like(b)
    z = np.zeros_like(b)
    u = np.zeros_like(b)

    # set up DnCNN model
    n_channels = 1
    model = net(in_nc=n_channels, out_nc=n_channels, nc=64, nb=17, act_mode='R')
    model.load_state_dict(torch.load('dncnn_25.pth'), strict=True)

    model.eval()
    for k, v in model.named_parameters():
        v.requires_grad = False
    model = model.to(device)

    ################# begin task 2 ###################################

    # ADMM with DnCNN doesn't require a gradient function, so we don't
    # need it here. Just pre-compute the denominator for the x-update
    # here, because that doesn't change unless rho changes, which is
    # not the case here

    # pre-compute denominator of x update
    denom = cTFT * cFT + rho # you need to edit this placeholder

    ################# end task 2 ###################################

    for it in tqdm(range(num_iters)):
        ################# begin task 2 ###################################

        # Complete this part by implementing the x-update discussed in
        # class and in the problem session. If you implemented the
```

```
        # denominator term above, you only need to compute the nominator
        # here as well as the rest of the x-update

        # x update - inverse filtering: Fourier multiplications and divisions
        part2 =cTFT *bFT +rho *fft2(z -u)
        x =ifft2(part2 /denom).real
        ################# end task 2 ####################################

        # z update
        v =x +u

        # run DnCNN denoiser
        v_tensor =torch.reshape(torch.from_numpy(v).float().to(device), (1, 1, v.shape[0], v.shape[1]))
        v_tensor_denoised =model(v_tensor)
        z =torch.squeeze(v_tensor_denoised).cpu().numpy()

        # u update
        u =u +x -z

    return x
```

C task3

C.1 leastnorm

```
def leastnorm(b, Afun, Atfun, num_iters, imageResolution):
    # number of measurements
    N =b.shape[0]
    # convergence tolerance of cg solver
    cg_tolerance =1e-12

    # initialize x with all zeros
    x =np.zeros(imageResolution)

    ################# begin task 3 ###################################
    #   Your task: implement a matrix-free conjugate gradient solver
    #       using the scipy.sparse.linalg.cg function in combination
    #       with the provided function handles Afun and Atfun. Use
    #       num_iters as the number of iterations of cg and
    #       cg_tolerance as the "tol" parameters for the cg function.
    #
    #   Hints:
    #       1. solve the problem (AA')y = b using CG first (CG only
    #           works for positive semi-definite matrices, like AA')
    #       2. then multiply the result by A' to get x as x=A'y
    #
    #   Be careful with your image dimensions. The cg function expects
    #   vector inputs and outputs, whereas b, x, Afun, Atfun all
    #   work with 2D images. So make sure you work with the vectorized
    #   versions for the function handles you pass into cg and then reshape
    #   the result to a 2D image again after.
    method =lambda x: Afun(Atfun(x))
    A_operator =LinearOperator((N, N), matvec=method)
    u =cg(A=A_operator, b=b, tol=cg_tolerance, maxiter=num_iters)
    x =Atfun(u[0]) # you need to edit this, it's just a placeholder

    ################# end task 3 ###################################

    return x
```

C.2 admm_tv

```
def admm_tv(b, Afun, Atfun, lam, rho, num_iters, imageResolution, anisotropic_tv=True):
    # initialize x,z,u with all zeros
```

```
    x = np.zeros(imageResolution)
    z = np.zeros((2, imageResolution[0], imageResolution[1]))
    u = np.zeros((2, imageResolution[0], imageResolution[1]))

    Ahat = lambda x: Atfun(Afun(x.reshape(imageResolution))) + rho * opDtx(opDx(x.reshape(imageResolution)))

    for it in tqdm(range(num_iters)):

        # x update using cg solver
        v = z - u

        cg_iters = 25 # number of iterations for CG solver
        cg_tolerance = 1e-12 # convergence tolerance of cg solver

        ################# begin task 3 ###################################
        #   Your task: implement a matrix-free conjugate gradient solver
        #       using the scipy.sparse.linalg.cg function in combination
        #       with the provided function handles Afun and Atfun to
        #       implement the x-update of ADMM. Use cg_iters as the number
        #       of iterations of cg and cg_tolerance as the "tol" parameters
        #       for the cg function.
        #
        #   Be careful with your image dimensions. The cg function expects
        #   vector inputs and outputs, whereas b, x, Afun, Atfun all
        #   work with 2D images. So make sure you work with the vectorized
        #   versions for the function handles you pass into cg and then reshape
        #   the result to a 2D image again after.

        bhat = Atfun(b) + rho * opDtx(v) # 64*64
        bhat = bhat.reshape(-1) # 4096
        N = bhat.shape[0]
        A_operator = LinearOperator((N, N), matvec=Ahat)
        x = cg(A=A_operator, b=bhat, tol=cg_tolerance,
               maxiter=cg_iters) # you need to edit this, it's just a placeholder
        x = x[0].reshape(imageResolution)

        ################# end task 3 ###################################

        # z update - soft shrinkage
        kappa = lam / rho
        v = opDx(x) + u

        # proximal operator of anisotropic TV term
        if anisotropic_tv:
            z = np.maximum(1 - kappa / np.abs(v), 0) * v

        # proximal operator of isotropic TV term
        else:
            vnorm = np.sqrt(v[0, :, :]**2 + v[1, :, :]**2)
            z[0, :, :] = np.maximum(1 - kappa / vnorm, 0) * v[0, :, :]
            z[1, :, :] = np.maximum(1 - kappa / vnorm, 0) * v[1, :, :]

        # u-update
        u = u + opDx(x) - z

    return x
```

C.3 admm_dncnn

```
def admm_dncnn(b, Afun, Atfun, lam, rho, num_iters, imageResolution):
    # initialize x,z,u with all zeros
    x = np.zeros(imageResolution)
    z = np.zeros(imageResolution)
    u = np.zeros(imageResolution)

    # load pre-trained DnCNN model
    model = net(in_nc=1, out_nc=1, nc=64, nb=17, act_mode='R')
    model.load_state_dict(torch.load('dncnn_25.pth'), strict=True)
```

```python
model.eval()
for k, v in model.named_parameters():
    v.requires_grad =False
model =model.to(device)

Ahat =lambda x: Atfun(Afun(x.reshape(imageResolution))) +rho *x.reshape(imageResolution)

for it in tqdm(range(num_iters)):
    # x update using cg solver
    v =z -u

    cg_iters =25 # number of iterations for CG solver
    cg_tolerance =1e-12 # convergence tolerance of cg solver

    ################# begin task 3 ###################################
    #  Your task: implement a matrix-free conjugate gradient solver
    #      using the scipy.sparse.linalg.cg function in combination
    #      with the provided function handles Afun and Atfun to
    #      implement the x-update of ADMM. Use cg_iters as the number
    #      of iterations of cg and cg_tolerance as the "tol" parameters
    #      for the cg function.
    #
    #  Be careful with your image dimensions. The cg function expects
    #  vector inputs and outputs, whereas b, x, Afun, Atfun all
    #  work with 2D images. So make sure you work with the vectorized
    #  versions for the function handles you pass into cg and then reshape
    #  the result to a 2D image again after.

    bhat =Atfun(b) +rho *v # 64*64

    bhat =bhat.reshape(-1) # 4096
    N =bhat.shape[0]
    A_operator =LinearOperator((N, N), matvec=Ahat)
    x =cg(A=A_operator, b=bhat, tol=cg_tolerance,
         maxiter=cg_iters) # you need to edit this, it's just a placeholder
    x =x[0].reshape(imageResolution)

    ################# end task 3 ###################################

    # z-update using DnCNN denoiser
    v =x +u
    v_tensor =torch.reshape(torch.from_numpy(v).float().to(device), (1, 1, v.shape[0], v.shape[1]))
    v_tensor_denoised =model(v_tensor)
    z =torch.squeeze(v_tensor_denoised).cpu().numpy()

    # u update
    u =u +x -z

return x
```