Wensong Hu
EECS
University of Michigan

April 17th, 2024

HW6 — 3D Deep Learning

1 Task 1: Estimating F and Epipoles

1.1 Coding: find_fundamental_matrix()

Code submitted to Canvas.

```python
def find_fundamental_matrix(shape, pts1, pts2):
    """
    Computes Fundamental Matrix F that relates points in two images by the:

        [u' v' 1] F [u v 1]^T = 0
        or
        l = F [u v 1]^T  -- the epipolar line for point [u v] in image 2
        [u' v' 1] F = l'   -- the epipolar line for point [u' v'] in image 1

    Where (u,v) and (u',v') are the 2D image coordinates of the left and
    the right images respectively.

    Inputs:
    - shape: Tuple containing shape of img1
    - pts1: Numpy array of shape (N,2) giving image coordinates in img1
    - pts2: Numpy array of shape (N,2) giving image coordinates in img2

    Returns:
    - F: Numpy array of shape (3,3) giving the fundamental matrix F
    """

    #This will give you an answer you can compare with
    #Your answer should match closely once you've divided by the last entry
    FOpenCV, _ = cv2.findFundamentalMat(pts1, pts2, cv2.FM_8POINT)

    F = np.eye(3)
    # #########################################################################
    # TODO: Your code here                                                    #
    # #########################################################################
```

```
    # Normalize the points to increase accuracy
    pts1_hom = homogenize(pts1)
    pts2_hom = homogenize(pts2)

    # Center and scale points for numerical stability
    mean1 = np.mean(pts1, axis=0)
    mean2 = np.mean(pts2, axis=0)
    std1 = np.std(pts1)
    std2 = np.std(pts2)

    # Transformation matrices for normalization
    T1 = np.array([
        [1/std1, 0, -mean1[0]/std1],
        [0, 1/std1, -mean1[1]/std1],
        [0, 0, 1]
    ])
    T2 = np.array([
        [1/std2, 0, -mean2[0]/std2],
        [0, 1/std2, -mean2[1]/std2],
        [0, 0, 1]
    ])

    # Normalize points
    pts1_norm = (T1 @ pts1_hom.T).T
    pts2_norm = (T2 @ pts2_hom.T).T

    # Create matrix A for the linear equation system Ax = 0
    A = np.zeros((len(pts1), 9))
    for i in range(len(pts1)):
        x1, y1, _ = pts1_norm[i]
        x2, y2, _ = pts2_norm[i]
        A[i] = [x2*x1, x2*y1, x2, y2*x1, y2*y1, y2, x1, y1, 1]

    # Solve the homogeneous equation system using SVD
    U, S, Vt = np.linalg.svd(A)
    F = Vt[-1].reshape(3, 3)

    # Enforce the rank constraint (rank 2)
    U, S, Vt = np.linalg.svd(F)
    S[2] = 0  # Set smallest singular value to 0
    F = U @ np.diag(S) @ Vt

    # Denormalize the fundamental matrix
    F = T2.T @ F @ T1
    print("F error: ", np.sum(F - FOpenCV))
    #
   ##########################################################################
    #                              END OF YOUR CODE
       #
    #
   ##########################################################################
    return F
```

1.2 Coding: compute_epipoles()

Code submitted to Canvas.

```python
def compute_epipoles(F):
    """
    Given a Fundamental Matrix F, return the epipoles represented in
    homogeneous coordinates.

    Check: e2@F and F@e1 should be close to [0,0,0]

    Inputs:
    - F: the fundamental matrix

    Return:
    - e1: the epipole for image 1 in homogeneous coordinates
    - e2: the epipole for image 2 in homogeneous coordinates
    """
    # ############################################################################
    # TODO: Your code here                                                       #
    # ############################################################################
    # Compute the right epipole (e2): null space of F
    U, S, Vt = np.linalg.svd(F)
    e2 = Vt[-1] + 1e-10 # The last row of V^T, corresponding to the smallest singular value

    # Compute the left epipole (e1): null space of F^T
    U, S, Vt = np.linalg.svd(F.T)
    e1 = U[:, -1] + 1e-10  # The last column of U, corresponding to the smallest singular value
    # ############################################################################
    #                             END OF YOUR CODE                               #
    # ############################################################################


    return e1, e2
```

1.3 Show epipolar lines for temple, reallyInwards, and another dataset of your choice.

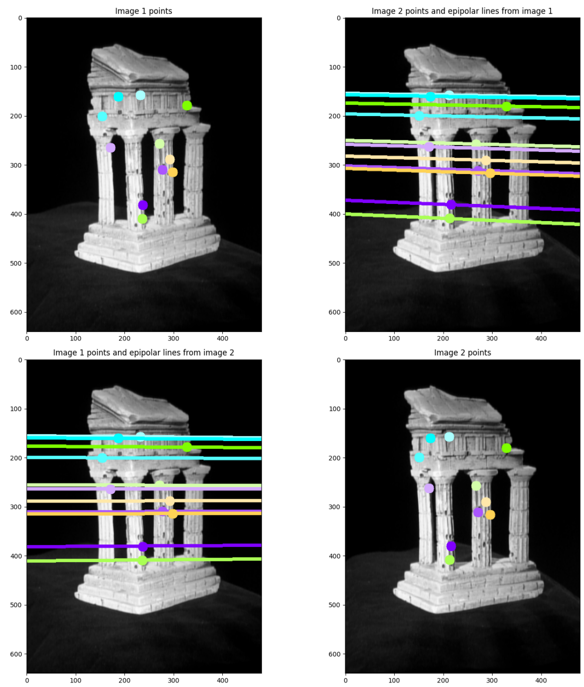

Figure 1: Temple

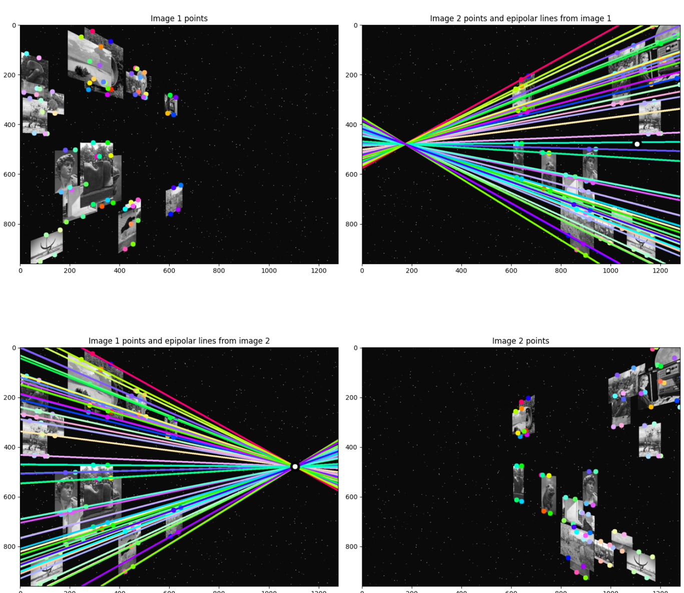

Figure 2: reallyInwards

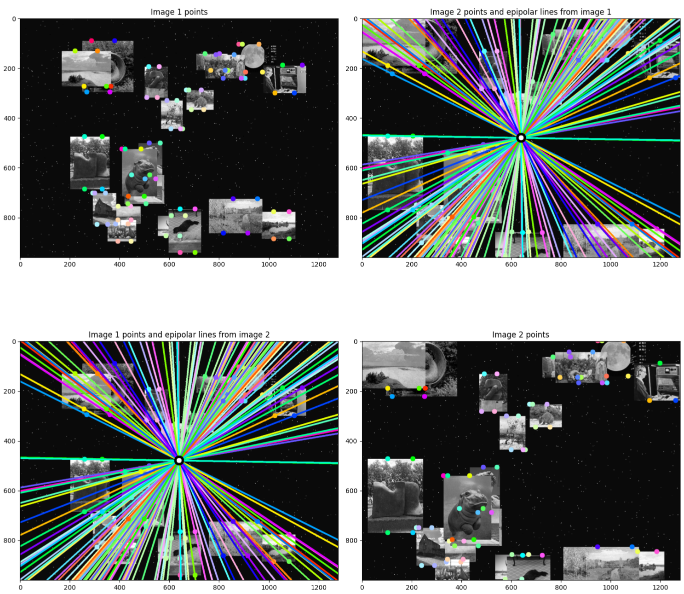

Figure 3: ztrans

1.4 Report the epipoles for reallyInwards and xtrans.

```
xtrans epipoles
[1.00000000e+00 9.99985793e-11 1.00000000e-10]
[-1.00000000e+00  1.00001421e-10  1.00000014e-10]
xtrans epipoles real coordinates
[1.00000000e+10 9.99985793e-01]
[-9.99999860e+09  1.00001407e+00]

reallyInwards epipoles
[9.17529952e-01 3.97665813e-01 8.29633180e-04]
[9.17529952e-01 3.97665813e-01 8.29633180e-04]
reallyInwards epipoles real coordinates
[1105.94654942  479.32727702]
[1105.94654942  479.32727702]
```

2 Task 2: NeRF

2.1 Coding: positional encoding

Code submitted to Canvas.

```
def positional_encoder(x, L_embed=6):
  """
  This function applies positional encoding to the input tensor.
   Positional encoding is used in NeRF
```

```
    to allow the model to learn high-frequency details more effectively. It applies sinusoidal functions
    at different frequencies to the input.

    Parameters:
    x (torch.Tensor): The input tensor to be positionally encoded.
    L_embed (int): The number of frequency levels to use in the encoding. Defaults to 6.

    Returns:
    torch.Tensor: The positionally encoded tensor.
    """

    # Initialize a list with the input tensor.
    rets = [x]

    # Loop over the number of frequency levels.
    for i in range(L_embed):
        # ###########################################################################
        #                                    TODO                                    #
        # ###########################################################################
        rets.append(torch.sin(2 ** i * x))
        rets.append(torch.cos(2 ** i * x))
        # ###########################################################################
        #                              END OF YOUR CODE                              #
        # ###########################################################################


    # Concatenate the original and encoded features along the last dimension.
    return torch.cat(rets, -1)
```

2.2 Coding: render()

Code submitted to Canvas.

```
def render(model, rays_o, rays_d, near, far, n_samples, rand=False):
    """
    Render a scene using a Neural Radiance Field (NeRF) model. This function samples points along rays,
    evaluates the NeRF model at these points, and applies volume rendering techniques to produce an image.

```

```python
    Parameters:
    model (torch.nn.Module): The NeRF model to be used for rendering.
    rays_o (torch.Tensor): Origins of the rays.
    rays_d (torch.Tensor): Directions of the rays.
    near (float): Near bound for depth sampling along the rays.
    far (float): Far bound for depth sampling along the rays.
    n_samples (int): Number of samples to take along each ray.
    rand (bool): If True, randomize sample depths. Default is False.

    Returns:
    tuple: A tuple containing the RGB map and depth map of the rendered scene.
    """

    # Sample points along each ray, from 'near' to 'far'.
    z = torch.linspace(near, far, n_samples).to(device)
    if rand:
      mids = 0.5 * (z[..., 1:] + z[..., :-1])
      upper = torch.cat([mids, z[..., -1:]], -1)
      lower = torch.cat([z[..., :1], mids], -1)
      t_rand = torch.rand(z.shape).to(device)
      z = lower + (upper - lower) * t_rand

    # ############################################################################
    #                                    TODO                                    #
    # ############################################################################
    # Compute 3D coordinates of the sampled points along the rays.
    points = rays_o[..., None, :] + rays_d[..., None, :] * z[..., :, None]
    # ############################################################################
    #                              END OF YOUR CODE                              #
    # ############################################################################

    # Flatten the points and apply positional encoding.
    flat_points = torch.reshape(points, [-1, points.shape[-1]])
    flat_points = positional_encoder(flat_points)

    # Evaluate the model on the encoded points in chunks to manage memory usage.
    chunk = 1024 * 32
    raw = torch.cat([model(flat_points[i:i + chunk]) for i in range(0, flat_points.shape[0], chunk)], 0)
    raw = torch.reshape(raw, list(points.shape[:-1]) + [4])

    # Compute densities (sigmas) and RGB values from the model's output.
```

```python
    sigma = F.relu(raw[..., 3])
    rgb = torch.sigmoid(raw[..., :3])

    # Perform volume rendering to obtain the weights of each point.
    one_e_10 = torch.tensor([1e10], dtype=rays_o.dtype).to(device)
    dists = torch.cat((z[..., 1:] - z[..., :-1], one_e_10.expand(z[..., :1].shape)), dim=-1)
    alpha = 1. - torch.exp(-sigma * dists)
    weights = alpha * cumprod_exclusive(1. - alpha + 1e-10)

    # #############################################################################
    #                                    TODO                                     #
    # #############################################################################
    # Compute the weighted sum of RGB values along each ray to get the final pixel color.
    rgb_map = torch.sum(rgb * weights[..., None], dim=-2)
    # Compute the depth map as the weighted sum of sampled depths.
    depth_map = torch.sum(weights * z, dim=-1)
    # #############################################################################
    #                              END OF YOUR CODE                               #
    # #############################################################################

    return rgb_map, depth_map
```

2.3 Coding: train()

Code submitted to Canvas.

```python
mse2psnr = lambda x : -10. * torch.log(x) / torch.log(torch.Tensor([10.])).to(device)

def train(model, optimizer, n_iters=3000):
    """
    Train the Neural Radiance Field (NeRF) model. This function performs training over a specified number of iterations,
    updating the model parameters to minimize the difference between rendered and actual images.

    Parameters:
    model (torch.nn.Module): The NeRF model to be trained.
    optimizer (torch.optim.Optimizer): The optimizer used for training the model.
    n_iters (int): The number of iterations to train the model. Default is 3000.
```

```python
"""

psnrs = []
iternums = []

plot_step = 500
n_samples = 64   # Number of samples along each ray.

for i in tqdm(range(n_iters)):
  # Randomly select an image from the dataset and use it as the target for training.
  images_idx = np.random.randint(images.shape[0])
  target = images[images_idx]
  pose = poses[images_idx]


  # ############################################################################
  #                                    TODO                                      #
  # ############################################################################
  # Perform training. Use mse loss for loss calculation and update the model parameter using the optimizer.
  # Hint: focal is defined as a global variable in previous section
  rays_o, rays_d = get_rays(H, W, focal=focal, pose=pose)
  rgb, depth = render(model=model, rays_o=rays_o, rays_d=rays_d, near=1., far=6., n_samples=n_samples, rand=True)

  loss = torch.nn.functional.mse_loss(rgb, target)
  optimizer.zero_grad()
  loss.backward()
  optimizer.step()

  # ############################################################################
  #                              END OF YOUR CODE                                #
  # ############################################################################

  if i % plot_step == 0:
    torch.save(model.state_dict(), 'ckpt.pth')
    # Render a test image to evaluate the current model performance.
    with torch.no_grad():
      rays_o, rays_d = get_rays(H, W, focal, testpose)
      rgb, depth = render(model, rays_o, rays_d, near=2., far=6., n_samples=n_samples)
      loss = torch.nn.functional.mse_loss(rgb, testimg)
      # Calculate PSNR for the rendered image.
```

```
        psnr = mse2psnr(loss)

        psnrs.append(psnr.detach().cpu().numpy())
        iternums.append(i)

        # Plotting the rendered image and PSNR over iterations.
        plt.figure(figsize=(9, 3))

        plt.subplot(131)
        picture = rgb.cpu()  # Copy the rendered image from GPU to CPU.
        plt.imshow(picture)
        plt.title(f'RGB Iter {i}')

        plt.subplot(132)
        picture = depth.cpu() * (rgb.cpu().mean(-1)>1e-2)
        plt.imshow(picture, cmap='gray')
        plt.title(f'Depth Iter {i}')

        plt.subplot(133)
        plt.plot(iternums, psnrs)
        plt.title('PSNR')
        plt.show()
```

2.4 Result

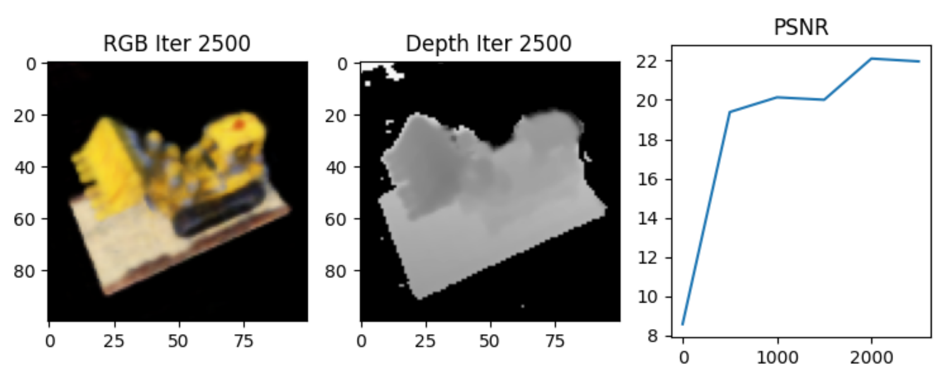

Figure 4: Training Result

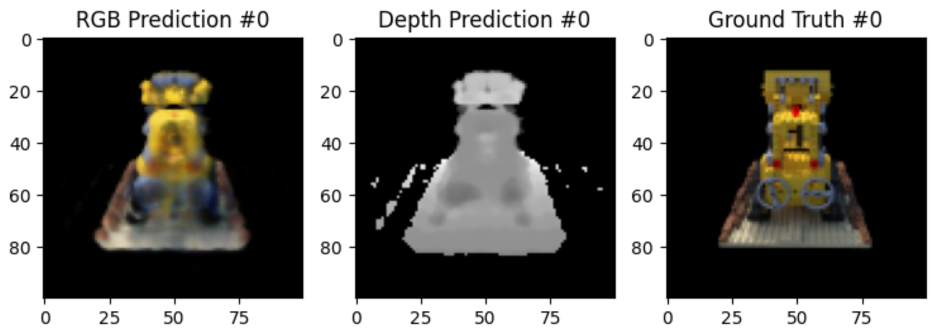

Figure 5: RGB Prediction #0

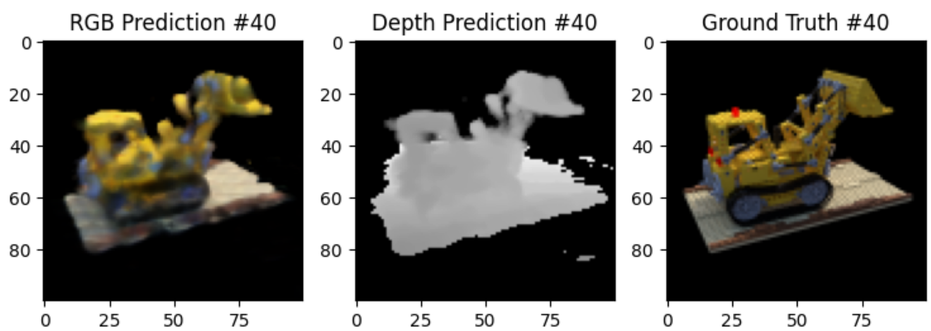

Figure 6: RGB Prediction #40

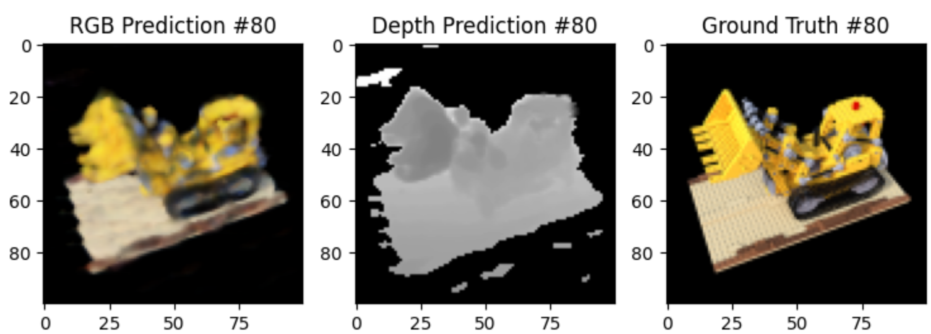

Figure 7: RGB Prediction #80

3 Appendix

Full Notebook pdf given in next page

Submitted by Wensong Hu on April 17th, 2024.