```
import torch
import torch.nn as nn
import tifffile
import torch.nn.parallel
import torch.utils.data
import os
import matplotlib.pyplot as plt
import numpy as np
import glob
#import torch.distributions as td
from reconstruction.dcgan_test import Generator
import torch.backends.cudnn as cudnn
from skimage import measure
import h5py
from pylab import gca
from analyze_pore_samples.plotting_utils import improve_pairplot

def read_file(fname):
    # fname = os.path.join(dir, filename)
    f = h5py.File(fname, 'r')
    return f['data']
def plt_scatter_matrices(prior_realizations, post_realizations, variable_list,
plt_prior=True, plt_post=True, show = False):
    '''
    Plot the prior and posterior scatter matrices (on top of each other) for a
*variable_list*.
    This is from a dictionary of *prior_realizations* and *post_realizations*.
    '''
    from pandas import DataFrame
    from pandas.plotting import scatter_matrix
    import seaborn as sns

    def plt_corr_sns(corr):

        f, ax = plt.subplots(figsize=(10, 8))
        sns.heatmap(corr, mask=np.zeros_like(corr, dtype=np.bool),
cmap=sns.diverging_palette(220, 10, as_cmap=True),
                    square=True, ax=ax, vmin = -1.0, vmax = 1.0)

    shape_prior = np.array(prior_realizations[variable_list[0]]).shape
    shape_post = np.array(post_realizations[variable_list[0]]).shape
    prior_data = prior_realizations.copy()
    prior_data['Case'] = ['Ground Truth']*shape_prior[0]#np.zeros(shape_prior)
    post_data = post_realizations.copy()
    post_data['Case'] = ['Reconstructed']*shape_post[0]#np.ones(shape_post)
    # breakpoint()
    if plt_prior and plt_post:
        for key in prior_data:
            prior_data[key] = np.append(prior_data[key],post_data[key])
        data = DataFrame(prior_data)
    elif plt_post:
        data = DataFrame(post_data)
    else:
        data = DataFrame(prior_data)
```

```
52
53      colors = ['blue','red']
54      # breakpoint()
55      plt.figure(figsize = [8,6], dpi = 150)
56      pplot = sns.pairplot(data, hue = 'Case', kind = 'kde', dropna = True,
    diag_kws=dict(common_norm= 'False'))
57      # breakpoint()
58      replacements = {'Volume': r'$log_{10} Volume$', 'Anisotropy':
    'Anisotropy',  'Orientation': r'Orientation, $\theta$ [rad]'}
59      improve_pairplot(pplot, replacements = replacements)
60      plt.savefig('./reconstruction/gan/figures/pairplot_gan.png')
61      if show:
62          plt.show()
63
64  def save_pore(pore_identity, epoch = None, real = False, fname = '',
    pore_matrix = None, value  =0 ):
65      if real:
66          filename = './analyze_pore_samples/results/individual_pore_samples
    /partsample0/pore_original_{}.hdf5'.format(pore_identity)
67      else:
68          filename   =   './reconstruction/gan/saved_generated_pores/generator_'
    + str(pore_identity) + '.hdf5'
69
70          # breakpoint()
71
72      data = np.array(read_file(filename))
73      labeled = measure.label(data)
74      pores = measure.regionprops(labeled)
75      pore_idx = np.argmax([pore.area for pore in pores])
76      data[labeled != pores[pore_idx].label] = 0
77      zmin  = np.min(np.where(data)[2]) - 1
78      zmax = np.max(np.where(data)[2])+ 1
79      xmin  = np.min(np.where(data)[0]) - 1
80      xmax = np.max(np.where(data)[0])+ 1
81      ymin  = np.min(np.where(data)[1]) - 1
82      ymax = np.max(np.where(data)[1])+ 1
83      bounds_min =  np.min([xmin, ymin, zmin])
84      bounds_max = np.max([zmax, ymax, xmax])
85
86      if not real:
87          directory_saved = './reconstruction/saved_generated_pores
    /pore_{}'.format(pore_identity) + fname
88      else:
89          directory_saved = './reconstruction/saved_example_pores
    /pore_{}'.format(pore_identity) + fname
90
91      os.makedirs(directory_saved, exist_ok=True)
92
    plt.figure().add_subplot(projection='3d').voxels(data[bounds_min:bounds_max,
    bounds_min:bounds_max, bounds_min:bounds_max], edgecolor = 'k')
93      plt.title( '{0:.2f}'.format(value) + ' ' +
    str(pore_matrix[pore_identity]))
94      plt.tight_layout()
95      plt.savefig(directory_saved+ '/pore.png')
96      plt.clf()
```

```
 97
 98 def query_pore(volume, anis, angle, pore_matrix):
 99     return np.argmin(np.abs(volume -
    pore_matrix[:,0])/np.mean(pore_matrix[:,0]) + np.abs(anis -
    pore_matrix[:,2])/np.mean(pore_matrix[:,2]) + np.abs(angle -
    pore_matrix[:,3])/np.mean(pore_matrix[:,3]))
100
101
102 params = {
103     'imsize' : 64,# Spatial size of training images. All images will be
    resized to this size during preprocessing.
104     'nc' : 2,# Number of channles in the training images. For coloured images
    this is 3.
105     'nz' : 100,# Size of the Z latent vector
106     'ngf' : 64,# Size of feature maps in the generator. The filtes will be
    multiples of this.
107     'ndf' : 16, # Size of features maps in the discriminator. The filters will
    be multiples of this.
108     'ngpu': 1, # Number of GPUs to be used
109     'nepochs' : 15,# Number of training epochs.
110     'lr' : 0.0002,# Learning rate for optimizers
111     'beta1' : 0.5,# Beta1 hyperparam for Adam optimizer
112     'alpha' : 1,# Size of z space
113     'stride' : 16,# Stride on image to crop
114     'num_samples' : 179}# Save step.
115
116 def legend(location = 'best', fontsize = 8):
117     plt.legend(loc = location, fontsize = fontsize, frameon = False)
118 def unit_vector(vector):
119     # """ Returns the unit vector of the vector.  """
120     return vector / np.linalg.norm(vector)
121 def angle_between(v1, v2):
122     """ Returns the angle in radians between vectors 'v1' and 'v2'::
123
124             >>> angle_between((1, 0, 0), (0, 1, 0))
125             1.5707963267948966
126             >>> angle_between((1, 0, 0), (1, 0, 0))
127             0.0
128             >>> angle_between((1, 0, 0), (-1, 0, 0))
129             3.141592653589793
130     """
131     v1_u = unit_vector(v1)
132     v2_u = unit_vector(v2)
133     return np.arccos(np.clip(np.dot(v1_u, v2_u), -1.0, 1.0))
134 def extract_pore(im):
135     size = 32
136     im = np.array(im.cpu().detach().numpy(), dtype = 'int')
137     props = measure.regionprops(measure.label(im))
138     p_idx = np.argmax([pore.area for pore in props])
139     pore_3d = np.zeros((size*2, size*2, size*2))
140     xdist = props[p_idx].slice[0].stop - props[p_idx].slice[0].start
141     ydist = props[p_idx].slice[1].stop - props[p_idx].slice[1].start
142     zdist = props[p_idx].slice[2].stop - props[p_idx].slice[2].start
143
144     pore_3d[size - xdist//2: size+(xdist-xdist//2), size-ydist//2:size+(ydist-
```

```
    ydist//2), size-zdist//2:size+(zdist-zdist//2)]  =
    np.array(props[p_idx].image, dtype = 'float')
145     # breakpoint()
146     return pore_3d
147 def analyze_pore(im):
148     voxelsize = 3.49
149 
150     realvols = []
151     realorientations = []
152     realanisotropies = []
153     realmin_axis_l = []
154     realmaj_axis_l = []
155     pores = measure.regionprops(measure.label(im))
156     if len(pores) > 1:
157         breakpoint()
158     # breakpoint()
159     if len(pores) == 0:
160         # breakpoint()
161         return [-1,0,0,0,0]
162     # try1
163     pore_idx = np.argmax([pore.area for pore in pores])
164     realvols.append(pores[pore_idx].area)
165     if pores[pore_idx].area< 3:
166         return [-1,0,0,0,0]
167     realmaj_axis_l.append((pores[pore_idx].major_axis_length)*voxelsize)
168     realmin_axis_l.append((pores[pore_idx].minor_axis_length)*voxelsize)
169 
170 
171     pore = pores[pore_idx]
172 
173     inertia_eigval = pore.inertia_tensor_eigvals
174     inertia = pore.inertia_tensor
175     maxeig = np.argmax(inertia_eigval)
176     eigvec = np.linalg.eig(pore.inertia_tensor)[1]
177     eigvals = np.linalg.eig(pore.inertia_tensor)[0]
178     if np.sum(eigvals) == 0:
179         breakpoint()
180     try:
181         anis = 1 - np.min(eigvals)/np.max(eigvals)
182     except:
183         anis = 0
184 
185     if np.max(eigvals) == 0:
186         anis = 0
187     realanisotropies.append(anis)
188     maxvector = eigvec[:, maxeig]
189     orientation = angle_between(maxvector, np.array([0,0,1]))
190     phi = angle_between(maxvector, np.array([0,1,0]))
191     realorientations.append(orientation)
192 
193     return [realvols[0], 1, anis, orientation, phi]
194 def save_hdf5(ndarr, filename):
195 
196     # tensor = tensor.cpu()
197     # ndarr = tensor.mul(255).byte().numpy()
```

```
198     with h5py.File(filename, 'w') as f:
199         f.create_dataset('data', data=ndarr, dtype="i8", compression="gzip")
200 def test_generator(epoch  = 38, folder_index = 0, num_samples = 100, show =
    False):
201     pore_list_attr = []
202     print("Epoch, ", epoch)
203     cudnn.benchmark = True
204 
205     # Use GPU is available else use CPU.
206     device = torch.device("cuda:0" if(torch.cuda.is_available()) else "cpu")
207     print(device, " will be used.\n")
208     # out_dir ='./reconstruction/generated_pore_samples' + str(epoch) +'/'
209     out_dir = './reconstruction/gan/saved_generated_pores/'
210     os.makedirs(str(out_dir), exist_ok=True)
211 
212     checkpoint = torch.load('./reconstruction
    /gan/netG_epoch_62.pth')##torch.load('./reconstruction/gan/saved_model.pth')
213     def frame_tick(frame_width = 2, tick_width = 1.5):
214         ax = gca()
215         for axis in ['top','bottom','left','right']:
216             ax.spines[axis].set_linewidth(frame_width)
217         plt.tick_params(direction = 'in',
218                         width = tick_width)
219 
220 
221     if('cuda' in str(device)):
222         # Create the generator.
223         netG = Generator(params['nz'], params['nc'], params['ngf'],
    params['ngpu'], size =64).to(device)
224         netG.load_state_dict(checkpoint)
225         netG = nn.DataParallel(netG)
226 
227     else:
228         # Create the generator.
229         netG = Generator(params['nz'], params['nc'], params['ngf'],
    params['ngpu']).to(device)
230         netG.load_state_dict(checkpoint)
231         netG = nn.DataParallel(netG)
232 
233         noise = torch.FloatTensor(1, params['nz'], params['alpha'],
    params['alpha'], params['alpha']).normal_(0, 1)
234         noise = noise.to(device)
235 
236     i =0
237     # Clean generated pores
238     print("Removing previously generated pores")
239     file_list = glob.glob(os.path.join(out_dir,'generator*hdf5'))
240     for fname in file_list:
241         os.remove(fname)
242 
243     while i < num_samples:#params['num_samples']:
244         noise = torch.FloatTensor(1, params['nz'], params['alpha'],
    params['alpha'], params['alpha']).normal_(0, 1)
245         noise = noise.to(device)
246         fake_data = netG(noise)
```

```
247         fake_argmax = fake_data.argmax(dim=1)
248         im = 1-fake_argmax[0]
249         if torch.sum(im) ==0:
250             continue
251         pore = extract_pore(1-fake_argmax[0])
252         fake_stats = analyze_pore(pore)
253         if fake_stats[0] < 0:
254             print('An unresolved pore filtered, continuing generation
    process')
255         else:
256 
257             pore_list_attr.append(fake_stats)
258             if i%100 == 0:
259                 print('========================')
260                 print(str(i) + ' pores processed')
261                 print('========================')
262             save_hdf5(pore, str(out_dir)+'/generator_{0}.hdf5'.format(i))
263             i+= 1
264     print('{} pores generated, saved in '.format(i), out_dir)
265     pore_matrix = np.loadtxt('analyze_pore_samples/results
    /individual_pore_samples/partsample{}/pore_matrix'.format(folder_index))
266     pore_matrix_generated = np.squeeze(np.array(pore_list_attr))
267 
268     np.savetxt('generated_pore_matrix.csv',pore_matrix_generated )
269     pore_identity = 100
270     filename   = str(out_dir)+'generator_{0}.hdf5'.format(pore_identity)
271     data = read_file(filename)
272     pore_sizes = pore_matrix[:,0]
273 
274     vol_target = np.sort(pore_matrix[:,0])[int(9*len(pore_matrix)//10)]
275     anis =np.sort(pore_matrix[:,2])[len(pore_matrix)//2]
276     angle = np.sort(pore_matrix[:,3])[len(pore_matrix)//2]
277     pore_identity_real = query_pore(vol_target, anis, angle, pore_matrix)
278     pore_identity_generated = query_pore(vol_target, anis, angle,
    pore_matrix_generated)
279 
280     save_pore(pore_identity_real, epoch = epoch, real  = True, fname =
    'baseline', pore_matrix=pore_matrix, value = vol_target)
281     save_pore(pore_identity_generated, epoch = epoch, fname = 'baseline',
    pore_matrix = pore_matrix_generated, value = vol_target)
282 
283 
284 
285     vol_target = np.sort(pore_matrix[:,0])[int(9.92*len(pore_matrix)//10)]
286     anis =np.sort(pore_matrix[:,2])[len(pore_matrix)//2]
287     angle = np.sort(pore_matrix[:,3])[len(pore_matrix)//2]
288     pore_identity_real = query_pore(vol_target, anis, angle, pore_matrix)
289     pore_identity_generated = query_pore(vol_target, anis, angle,
    pore_matrix_generated)
290 
291     save_pore(pore_identity_real, epoch = epoch, real  = True, fname =
    'bigvol', pore_matrix=pore_matrix, value = vol_target)
292     save_pore(pore_identity_generated, epoch = epoch, fname = 'bigvol',
    pore_matrix = pore_matrix_generated, value = vol_target)
293 
```

```
294
295     vol_target = np.sort(pore_matrix[:,0])[int(9*len(pore_matrix)//10)]
296     anis =np.sort(pore_matrix[:,2])[int(9.9*len(pore_matrix)//10)]
297     angle = np.sort(pore_matrix[:,3])[len(pore_matrix)//2]
298     pore_identity_real = query_pore(vol_target, anis, angle, pore_matrix)
299     pore_identity_generated = query_pore(vol_target, anis, angle,
    pore_matrix_generated)
300
301     save_pore(pore_identity_real, epoch = epoch, real  = True, fname =
    'biganis', pore_matrix=pore_matrix, value = anis)
302     save_pore(pore_identity_generated, epoch = epoch, fname = 'biganis',
    pore_matrix = pore_matrix_generated, value = anis)
303
304
305     vol_target = np.sort(pore_matrix[:,0])[int(9*len(pore_matrix)//10)]
306     anis =np.sort(pore_matrix[:,2])[len(pore_matrix)//2]
307     angle = np.sort(pore_matrix[:,3])[int(9.9*len(pore_matrix)//10)]
308     pore_identity_real = query_pore(vol_target, anis, angle, pore_matrix)
309     pore_identity_generated = query_pore(vol_target, anis, angle,
    pore_matrix_generated)
310
311     save_pore(pore_identity_real, epoch = epoch, real  = True, fname =
    'bigangle', pore_matrix=pore_matrix, value = angle)
312     save_pore(pore_identity_generated, epoch = epoch, fname = 'bigangle',
    pore_matrix = pore_matrix_generated, value = angle)
313     os.makedirs('./reconstruction/gan/figures',exist_ok = True)
314     np.savetxt('./reconstruction/gan/figures
    /pore_matrix_updated.csv',pore_matrix_generated )
315     plt.figure(figsize = [4,3], dpi = 150 )
316
317     plt.hist(pore_matrix[:,0]*(3.49)**3, density = True, alpha =
    0.7,bins=np.logspace(np.log10(10e1),np.log10(10e5)), edgecolor = 'k',  label =
    "Ground Truth")
318     plt.hist(pore_matrix_generated[:,0]*(3.49)**3, density = True, alpha =
    0.7,bins=np.logspace(np.log10(10e1),np.log10(10e5)), edgecolor = 'k',  label =
    "Generated")
319     plt.plot([8*3.49**3,8*3.49**3], [0, 0.004])
320     # plt.ylim([0, 0.003])
321     plt.xscale('log')
322     plt.yscale('log')
323
324     legend()
325     plt.xlabel(r"Volume [$\mu m^3$]")
326     plt.ylabel("Probability")
327
328     frame_tick()
329     plt.tight_layout()
330     plt.savefig('./reconstruction/gan/figures/volume' +str(epoch))
331     if show:
332         plt.show()
333     plt.clf()#savefig("vols_new" + str(k+ 100) + ".png")
334
335     plt.figure(figsize = [4,3], dpi = 150 )
336     frame_tick()
337     plt.hist((np.array(pore_matrix[:,3])/np.pi)*180, density = True, bins=30,
```

```
    edgecolor = 'k', alpha  =0.7, label = "Ground Truth")
338     plt.hist((np.array(pore_matrix_generated[:,3])/np.pi)*180,density = True,
    bins=30,alpha = 0.7, edgecolor = 'k', label = "Generated")
339 
340     legend()
341     plt.xlabel(r"Angle [Degrees]")
342     plt.ylabel("Probability")
343     plt.tight_layout()
344     plt.savefig('./reconstruction/gan/figures/angle' +str(epoch))
345     if show:
346         plt.show()
347     plt.clf()
348 
349     plt.figure(figsize = [4,3], dpi = 150 )
350     frame_tick()
351 
352     plt.hist(pore_matrix[:,2], density = True, bins=30, edgecolor = 'k', label
    = "Ground Truth", alpha  = 0.7)
353     plt.hist(pore_matrix_generated[:,2], density = True, bins=30, edgecolor =
    'k',  label = "Generated", alpha  = 0.7)
354 
355     legend()
356     plt.xlabel(r"Anisotropy")
357     plt.ylabel("Probability")
358     plt.tight_layout()
359     plt.savefig('./reconstruction/gan/figures/anisotropy' +str(epoch))
360     if show:
361         plt.show()
362     plt.clf()
363 
364     # plt.clf()
365     dict_one = {'Volume': np.log10(pore_matrix[:,0]),
    'Anisotropy':pore_matrix[:,2], 'Orientation':pore_matrix[:,3]}#,'phis':
    gt_total_phi }
366     dict_two = {'Volume': np.log10(pore_matrix_generated[:,0]),
    'Anisotropy':pore_matrix_generated[:,2],
    'Orientation':pore_matrix_generated[:,3]}
367     plt_scatter_matrices(dict_one, dict_two, list(dict_one.keys()), show =
    show)
368 
369     plt.clf()
370 if __name__ == "__main__":
371     test_generator()
```