

```
import numpy as np
import argparse
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import random
import os
import h5py
from dataset_test import HDF5Dataset
from matplotlib import pyplot as plt
import torchvision.transforms as transforms
#from hdf5_io import save_hdf5
from torchvision.utils import save_image
from dcgan_test import Generator, Discriminator
from mpl_toolkits.mplot3d import Axes3D
from datetime import datetime, date, time
#import numpy as np
#np.random.seed(43)

# Set random seed for reproducibility.
seed = 500
random.seed(seed)
torch.manual_seed(seed)
print("Random Seed: ", seed)

parser = argparse.ArgumentParser()
parser.add_argument('--dataroot', default='', help='input dataset file')
parser.add_argument('--out_dir_hdf5', default='', help='output file for generated images')
parser.add_argument('--out_dir_model', default='', help='output file for model')
parser.add_argument('--workers', type=int, default=0, help='number of workers')
parser.add_argument('--cuda', action='store_true', help='enables cuda')
parser.add_argument('--ngpu', type=int, default=1, help='number of GPUs to use')
parser.add_argument('--bsize', default=32, help='batch size during training')
parser.add_argument('--imsize', default=64, help='size of training images')
parser.add_argument('--nc', default=2, help='number of channels')
parser.add_argument('--nz', default=100, help='size of z latent vector')
parser.add_argument('--ngf', default=64, help='size of feature maps in generator')
parser.add_argument('--ndf', default=16, help='size of feature maps in discriminator')
parser.add_argument('--nepochs', default=1000, help='number of training epochs')
parser.add_argument('--lr', default=0.00002, help='learning rate for optimisers')
parser.add_argument('--beta1', default=0.5, help='beta1 hyperparameter for Adam optimiser')
parser.add_argument('--save_epoch', default=2, help='step for saving paths')
parser.add_argument('--sample_interval', default=50, help='output image step')
```

```
46
47 opt = parser.parse_args()
48 cudnn.benchmark = True
49 timestamp = datetime.now()
50 str_timestamp = timestamp.strftime('%Y-%m-%d-%H-%M-%S')
51 opt.out_dir_hdf5 = 'reconstruction/results/' + str(opt.imsz) +
    'morehdf5pores/img_out_new2' + str_timestamp
52 opt.out_dir_model = 'reconstruction/results/' + str(opt.imsz) +
    'moremodelpores/mod_out_new' + str_timestamp
53 opt.dataroot = 'analyze_pore_samples/results/individual_pore_samples'
54 ngpu = int(opt.ngpu)
55 nz = int(opt.nz)
56 ngf = int(opt.ngf)
57 ndf = int(opt.ndf)
58 nc = int(opt.nc)
59 workers = int(opt.workers)
60
61 # Use GPU is available else use CPU.
62 device = torch.device("cuda:0" if(torch.cuda.is_available() and ngpu > 0) else
    "cpu")
63 print(device, " will be used.\n")
64
65 # Get the data.
66 dataset = HDF5Dataset(opt.dataroot,
67                       input_transform=transforms.Compose([
68                           transforms.ToTensor()
69                       ]))
70
71 dataloader = torch.utils.data.DataLoader(dataset,
72     batch_size=opt.bsize,
73     shuffle=True, num_workers=workers)
74 sample_batch = next(iter(dataloader))
75
76 os.makedirs(str(opt.out_dir_hdf5), exist_ok=True)
77 os.makedirs(str(opt.out_dir_model), exist_ok=True)
78
79 #####
80 # Functions to be used:
81 #####
82 # weights initialisation
83 def weights_init(w):
84     """
85     Initializes the weights of the layer, w.
86     """
87     classname = w.__class__.__name__
88     if classname.find('Conv') != -1:
89         nn.init.normal_(w.weight.data, 0.0, 0.02)
90     elif classname.find('BatchNorm') != -1:
91         nn.init.normal_(w.weight.data, 1.0, 0.02)
92         nn.init.constant_(w.bias.data, 0)
93
94 # save tensor into hdf5 format
95 def save_hdf5(tensor, filename):
96
97     tensor = tensor.cpu()
```

```
98     ndarr = tensor.mul(255).byte().numpy()
99     with h5py.File(filename, 'w') as f:
100         f.create_dataset('data', data=ndarr, dtype="i8", compression="gzip")
101
102 #####
103
104 # Create the generator
105 netG = Generator(nz, nc, ngf, ngpu, size = int(opt.imsize)).to(device)
106
107 if('cuda' in str(device)) and (ngpu > 1):
108     netG = nn.DataParallel(netG, list(range(ngpu)))
109
110 netG.apply(weights_init)
111 print(netG)
112
113 # Create the discriminator
114 netD = Discriminator(nz, nc, ndf, ngpu, size = int(opt.imsize)).to(device)
115
116 if('cuda' in str(device)) and (ngpu > 1):
117     netD = nn.DataParallel(netD, list(range(ngpu)))
118
119 netD.apply(weights_init)
120 print(netD)
121
122 # Binary Cross Entropy loss function.
123 criterion = nn.BCELoss()
124
125 if(device.type == 'cuda'):
126     netD.cuda()
127     netG.cuda()
128     criterion.cuda()
129
130 real_label = 0.9 # lable smoothing epsilon = 0.1
131 fake_label = 0
132
133 # Optimizer for the discriminator.
134 optimizerD = optim.Adam(netD.parameters(), lr=float(opt.lr), betas=(opt.beta1,
135 0.999))
136 # Optimizer for the generator.
137 optimizerG = optim.Adam(netG.parameters(), lr=float(opt.lr), betas=(opt.beta1,
138 0.999))
139
140 # Stores generated images as training progresses.
141 img_list = []
142 # Stores generator losses during training.
143 G_losses = []
144 # Stores discriminator losses during training.
145 D_losses = []
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```

```

151
152 for epoch in range(opt.nepochs):
153     for i, data_labels in enumerate(dataloader, 0):
154         #####
155         # (1) Update D network: maximise log(D(x)) + log(1 - D(G(z)))
156         #####
157         netD.zero_grad()
158
159         data = data_labels[:, None, :, :, :]
160         data = torch.cat((data, 1-data), dim = 1)
161
162         real_data = data.to(device)
163         #print('real', real_data.shape)
164
165         b_size = real_data.size(0)
166         #print(b_size)
167
168         label = torch.full((b_size,), real_label, device=device)
169
170         output = netD(real_data).view(-1)
171         #output from D will be of size (b_size, 1, 1, 1, 1), with view(-1) we
172         #reshape the output to have size (b_size)
173         errD_real = criterion(output, label) # log(D(x))
174         errD_real.backward()
175         D_x = output.mean().item()
176
177
178         noise = torch.randn(b_size, nz, 1, 1, 1, device=device)
179         fake_data = netG(noise)
180         label.data.fill_(fake_label)
181         output = netD(fake_data.detach()).view(-1) # detach() no need for
gradients
182         #print(output.shape)
183         errD_fake = criterion(output, label) # log(1 - D(G(z)))
184         errD_fake.backward()
185         D_G_z1 = output.mean().item()
186         errD = errD_real + errD_fake
187         optimizerD.step()
188
189         #####
190         # (2) Update G network: maximise log(D(G(z)))
191         #####
192
193         gen_it = 1
194         while gen_it != 0:
195             netG.zero_grad()
196             label.data.fill_(real_label)
197             noise = torch.randn(b_size, nz, 1, 1, 1, device=device)
198             fake_data = netG(noise)
199             #print(fake_data.shape)
200             output = netD(fake_data).view(-1)
201             errG = criterion(output, label) # log(D(G(z)))
202             errG.backward()
203             D_G_z2 = output.data.mean().item()
204             optimizerG.step()

```

```

205         gen_it -= 1
206
207     iters += 1
208
209     # Check progress of training.
210     if i%50 == 0:
211         print(['%d/%d']['%d/%d']\tLoss_D: %.4f\tLoss_G: %.4f\tD(x):
%.4f\tD(G(z)): %.4f / %.4f'
212               % (epoch, opt.nepochs, i, len(dataloader),
213                 errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))
214
215     # Save the losses for plotting.
216     G_losses.append(errG.item())
217     D_losses.append(errD.item())
218     batches_done = epoch * len(dataloader) + i
219     if i%50 == 0:
220         plt.plot( G_losses, label = 'G loss')
221         plt.plot( D_losses, label = 'D loss')
222         plt.legend()
223         plt.ylabel('loss')
224         plt.xlabel(' Iterations')
225         plt.savefig(str(opt.out_dir_hdf5)+'/dgloss.png')
226         plt.clf()
227
228     if batches_done % opt.sample_interval == 0:
229         #fake = netG(noise)
230         save_hdf5(fake_data.data,
str(opt.out_dir_hdf5)+'/'fake_{0}.hdf5'.format(batches_done))
231
232 #####
233 #
234 # This section can be included for saving the images produced at each timestep
235 # It increases the processing time more than three times, but is useful to
view
236 # if the algorithm is producing reasonable images
237 print(batches_done)
238 if batches_done % opt.sample_interval == 0:
239     print("SAVING")
240     #fig = plt.figure()
241     # ax = fig.gca(projection='3d')
242
243     output_data = fake_data.argmax(dim=1)
244     plt.imshow(np.sum(output_data[0].cpu().numpy(), axis = 2))
245     plt.savefig(str(opt.out_dir_hdf5)+'/'2d_{0}.png'.format(batches_done))
246     plt.clf()
247     plt.close('all')
248     print("DONE SAVING")
249 #####
250 #
251 if epoch % opt.save_epoch == 0:
252     # Save checkpoints
253     torch.save(netG.state_dict(),
str(opt.out_dir_model)+'/'netG_epoch_{0}.pth'.format(epoch))
254     torch.save(netD.state_dict(),

```

```
    str(opt.out_dir_model)+'/netD_epoch_{}.pth'.format(epoch))
254     torch.save(optimizerG.state_dict(),
    str(opt.out_dir_model)+'/optimG_epoch_{}.pth'.format(epoch))
255     torch.save(optimizerD.state_dict(),
    str(opt.out_dir_model)+'/optimD_epoch_{}.pth'.format(epoch))
256
257 # Save the final trained model
258 torch.save(netG.state_dict(),
    str(opt.out_dir_model)+'/netG_final.pth'.format(epoch))
259 torch.save(netD.state_dict(),
    str(opt.out_dir_model)+'/netD_final.pth'.format(epoch))
260 torch.save(optimizerG.state_dict(),
    str(opt.out_dir_model)+'/optimG_final.pth'.format(epoch))
261 torch.save(optimizerD.state_dict(),
    str(opt.out_dir_model)+'/optimD_final.pth'.format(epoch))
262
```