Generalized zero-shot learning

# example code(CE\_GZSL.py)

|  |
| --- |
| *import* torch.optim *as* optim  *import* torch.backends.cudnn *as* cudnn  *from* torch.autograd *import* Variable  *import* util  *import* classifier\_embed\_contras  *import* model  *import* losses  *import* torch.nn.functional *as* F  parser = argparse.ArgumentParser()  parser.add\_argument('--dataset', *default*='CUB', *help*='FLO')  parser.add\_argument('--dataroot', *default*='data/xlsa17/data/', *help*='path to dataset')  parser.add\_argument('--matdataset', *default*=True, *help*='Data in matlab format')  parser.add\_argument('--image\_embedding', *default*='res101')  parser.add\_argument('--class\_embedding', *default*='sent',*help*='att or sent')  parser.add\_argument('--syn\_num', *type*=int, *default*=100, *help*='number features to generate per class')  parser.add\_argument('--gzsl', *type*=bool, *default*=True, *help*='enable generalized zero-shot learning')  parser.add\_argument('--preprocessing', *type*=bool, *default*=True, *help*='enbale MinMaxScaler on visual features')  parser.add\_argument('--standardization', *action*='store\_true', *default*=False)  parser.add\_argument('--validation', *action*='store\_true', *default*=False, *help*='enable cross validation mode')  parser.add\_argument('--workers', *type*=int, *help*='number of data loading workers', *default*=2)  parser.add\_argument('--batch\_size', *type*=int, *default*=2048, *help*='input batch size')  parser.add\_argument('--resSize', *type*=int, *default*=2048, *help*='size of visual features')  parser.add\_argument('--attSize', *type*=int, *default*=1024 , *help*='size of semantic features')  parser.add\_argument('--nz', *type*=int, *default*=1024, *help*='noise for generation')  parser.add\_argument('--embedSize', *type*=int, *default*=2048, *help*='size of embedding h')  parser.add\_argument('--outzSize', *type*=int, *default*=512, *help*='size of non-liner projection z')  *## network architechure*  parser.add\_argument('--ngh', *type*=int, *default*=4096, *help*='size of the hidden units in generator G')  parser.add\_argument('--ndh', *type*=int, *default*=4096, *help*='size of the hidden units in discriminator D')  parser.add\_argument('--nhF', *type*=int, *default*=2048, *help*='size of the hidden units comparator network F')  parser.add\_argument('--ins\_weight', *type*=float, *default*=0.001, *help*='weight of the classification loss when learning G')  parser.add\_argument('--cls\_weight', *type*=float, *default*=0.001, *help*='weight of the score function when learning G')  parser.add\_argument('--ins\_temp', *type*=float, *default*=0.1, *help*='temperature in instance-level supervision')  parser.add\_argument('--cls\_temp', *type*=float, *default*=0.1, *help*='temperature in class-level supervision')  parser.add\_argument('--nepoch', *type*=int, *default*=2000, *help*='number of epochs to train for')  parser.add\_argument('--critic\_iter', *type*=int, *default*=5, *help*='critic iteration, following WGAN-GP')  parser.add\_argument('--lr', *type*=float, *default*=0.0001, *help*='learning rate to training')  parser.add\_argument('--lr\_decay\_epoch', *type*=int, *default*=100, *help*='conduct learning rate decay after every 100 epochs')  parser.add\_argument('--lr\_dec\_rate', *type*=float, *default*=0.99, *help*='learning rate decay rate')  parser.add\_argument('--lambda1', *type*=float, *default*=10, *help*='gradient penalty regularizer, following WGAN-GP')  parser.add\_argument('--classifier\_lr', *type*=float, *default*=0.001, *help*='learning rate to train softmax classifier')  parser.add\_argument('--beta1', *type*=float, *default*=0.5, *help*='beta1 for adam. default=0.5')  parser.add\_argument('--cuda', *action*='store\_true', *default*=True, *help*='enables cuda')  parser.add\_argument('--manualSeed', *type*=int, *default*=3483, *help*='manual seed')  parser.add\_argument('--nclass\_all', *type*=int, *default*=200, *help*='number of all classes')  parser.add\_argument('--nclass\_seen', *type*=int, *default*=150, *help*='number of all classes')  parser.add\_argument('--gpus', *default*='0', *help*='the number of the GPU to use')  opt = parser.parse\_args()  print(opt)  os.environ['CUDA\_VISIBLE\_DEVICES'] = opt.gpus  *if* opt.manualSeed *is* None:      opt.manualSeed = random.randint(1, 10000)  print("Random Seed: ", opt.manualSeed)  random.seed(opt.manualSeed)  torch.manual\_seed(opt.manualSeed)  *if* opt.cuda:      torch.cuda.manual\_seed\_all(opt.manualSeed)  cudnn.benchmark = True  *if* torch.cuda.is\_available() *and* *not* opt.cuda:      print("WARNING: You have a CUDA device, so you should probably run with --cuda")  *# load data*  data = util.DATA\_LOADER(opt)  print("# of training samples: ", data.ntrain)  netG = model.MLP\_G(opt)  netMap = model.Embedding\_Net(opt)  netD = model.MLP\_CRITIC(opt)  F\_ha = model.Dis\_Embed\_Att(opt)  model\_path = './models/' + opt.dataset  *if* *not* os.path.exists(model\_path):      os.makedirs(model\_path)  *if* len(opt.gpus.split(','))>1:      netG=nn.DataParallel(netG)      netD = nn.DataParallel(netD)      netMap = nn.DataParallel(netMap)      F\_ha = nn.DataParallel(F\_ha)  contras\_criterion = losses.SupConLoss\_clear(opt.ins\_temp)  input\_res = torch.FloatTensor(opt.batch\_size, opt.resSize)  input\_att = torch.FloatTensor(opt.batch\_size, opt.attSize)  noise\_gen = torch.FloatTensor(opt.batch\_size, opt.nz)  input\_label = torch.LongTensor(opt.batch\_size)  *if* opt.cuda:      netG.cuda()      netD.cuda()      netMap.cuda()      F\_ha.cuda()      input\_res = input\_res.cuda()      noise\_gen, input\_att = noise\_gen.cuda(), input\_att.cuda()      input\_label = input\_label.cuda()  def sample():      batch\_feature, batch\_label, batch\_att = data.next\_batch(opt.batch\_size)      input\_res.copy\_(batch\_feature)      input\_att.copy\_(batch\_att)      input\_label.copy\_(util.map\_label(batch\_label, data.seenclasses))  def generate\_syn\_feature(*netG*, *classes*, *attribute*, *num*):      nclass = *classes*.size(0)      syn\_feature = torch.FloatTensor(nclass \* *num*, opt.resSize)      syn\_label = torch.LongTensor(nclass \* *num*)      syn\_att = torch.FloatTensor(*num*, opt.attSize)      syn\_noise = torch.FloatTensor(*num*, opt.nz)  *if* opt.cuda:          syn\_att = syn\_att.cuda()          syn\_noise = syn\_noise.cuda()  *for* i *in* range(nclass):          iclass = *classes*[i]          iclass\_att = *attribute*[iclass]          syn\_att.copy\_(iclass\_att.repeat(*num*, 1))          syn\_noise.normal\_(0, 1)  *with* torch.no\_grad():              output = *netG*(syn\_noise, syn\_att)          syn\_feature.narrow(0, i \* *num*, *num*).copy\_(output.data.cpu())          syn\_label.narrow(0, i \* *num*, *num*).fill\_(iclass)  *return* syn\_feature, syn\_label  *# setup optimizer*  *import* itertools  optimizerD = optim.Adam(itertools.chain(netD.parameters(), netMap.parameters(), F\_ha.parameters()), *lr*=opt.lr,  *betas*=(opt.beta1, 0.999))  optimizerG = optim.Adam(netG.parameters(), *lr*=opt.lr, *betas*=(opt.beta1, 0.999))  def calc\_gradient\_penalty(*netD*, *real\_data*, *fake\_data*, *input\_att*):  *# print real\_data.size()*      alpha = torch.rand(opt.batch\_size, 1)      alpha = alpha.expand(*real\_data*.size())  *if* opt.cuda:          alpha = alpha.cuda()      interpolates = alpha \* *real\_data* + ((1 - alpha) \* *fake\_data*)  *if* opt.cuda:          interpolates = interpolates.cuda()      interpolates = Variable(interpolates, *requires\_grad*=True)      disc\_interpolates = *netD*(interpolates, *input\_att*)      ones = torch.ones(disc\_interpolates.size())  *if* opt.cuda:          ones = ones.cuda()      gradients = autograd.grad(*outputs*=disc\_interpolates, *inputs*=interpolates,  *grad\_outputs*=ones,  *create\_graph*=True, *retain\_graph*=True, *only\_inputs*=True)[0]      gradient\_penalty = ((gradients.norm(2, *dim*=1) - 1) \*\* 2).mean() \* opt.lambda1  *return* gradient\_penalty  *# use the for-loop to save the GPU-memory*  def class\_scores\_for\_loop(*embed*, *input\_label*, *relation\_net*):      all\_scores=torch.FloatTensor(*embed*.shape[0],opt.nclass\_seen).cuda()  *for* i, i\_embed *in* enumerate(*embed*):          expand\_embed = i\_embed.repeat(opt.nclass\_seen, 1)*#.reshape(embed.shape[0] \* opt.nclass\_seen, -1)*          all\_scores[i]=(torch.div(*relation\_net*(torch.cat((expand\_embed, data.attribute\_seen.cuda()), *dim*=1)),opt.cls\_temp).squeeze())      score\_max, \_ = torch.max(all\_scores, *dim*=1, *keepdim*=True)  *# normalize the scores for stable training*      scores\_norm = all\_scores - score\_max.detach()      mask = F.one\_hot(*input\_label*, *num\_classes*=opt.nclass\_seen).float().cuda()      exp\_scores = torch.exp(scores\_norm)      log\_scores = scores\_norm - torch.log(exp\_scores.sum(1, *keepdim*=True))      cls\_loss = -((mask \* log\_scores).sum(1) / mask.sum(1)).mean()  *return* cls\_loss  *# It is much faster to use the matrix, but it cost much GPU memory.*  def class\_scores\_in\_matrix(*embed*, *input\_label*, *relation\_net*):      expand\_embed = *embed*.unsqueeze(*dim*=1).repeat(1, opt.nclass\_seen, 1).reshape(*embed*.shape[0] \* opt.nclass\_seen, -1)      expand\_att = data.attribute\_seen.unsqueeze(*dim*=0).repeat(*embed*.shape[0], 1, 1).reshape(  *embed*.shape[0] \* opt.nclass\_seen, -1).cuda()      all\_scores = torch.div(*relation\_net*(torch.cat((expand\_embed, expand\_att), *dim*=1)),opt.cls\_temp).reshape(*embed*.shape[0],                                                                                                      opt.nclass\_seen)      score\_max, \_ = torch.max(all\_scores, *dim*=1, *keepdim*=True)      scores\_norm = all\_scores - score\_max.detach()      mask = F.one\_hot(*input\_label*, *num\_classes*=opt.nclass\_seen).float().cuda()      exp\_scores = torch.exp(scores\_norm)      log\_scores = scores\_norm - torch.log(exp\_scores.sum(1, *keepdim*=True))      cls\_loss = -((mask \* log\_scores).sum(1) / mask.sum(1)).mean()  *return* cls\_loss  *for* epoch *in* range(opt.nepoch):      FP = 0      mean\_lossD = 0      mean\_lossG = 0  *for* i *in* range(0, data.ntrain, opt.batch\_size):  *############################*  *# (1) Update D network: optimize WGAN-GP objective, Equation (2)*  *###########################*  *for* p *in* netD.parameters():  *# reset requires\_grad*              p.requires\_grad = True  *# they are set to False below in netG update*  *for* p *in* netMap.parameters():  *# reset requires\_grad*              p.requires\_grad = True  *for* p *in* F\_ha.parameters():  *# reset requires\_grad*              p.requires\_grad = True  *for* iter\_d *in* range(opt.critic\_iter):              sample()              netD.zero\_grad()              netMap.zero\_grad()  *#*  *# train with realG*  *# sample a mini-batch*              sparse\_real = opt.resSize - input\_res[1].gt(0).sum()              embed\_real, outz\_real = netMap(input\_res)              criticD\_real = netD(input\_res, input\_att)              criticD\_real = criticD\_real.mean()  *# CONTRASITVE LOSS*              real\_ins\_contras\_loss = contras\_criterion(outz\_real, input\_label)  *# train with fakeG*              noise\_gen.normal\_(0, 1)              fake = netG(noise\_gen, input\_att)              fake\_norm = fake.data[0].norm()              sparse\_fake = fake.data[0].eq(0).sum()              criticD\_fake = netD(fake.detach(), input\_att)              criticD\_fake = criticD\_fake.mean()  *# gradient penalty*              gradient\_penalty = calc\_gradient\_penalty(netD, input\_res, fake.data, input\_att)              Wasserstein\_D = criticD\_real - criticD\_fake              cls\_loss\_real = class\_scores\_for\_loop(embed\_real, input\_label, F\_ha)              D\_cost = criticD\_fake - criticD\_real + gradient\_penalty + real\_ins\_contras\_loss + cls\_loss\_real              D\_cost.backward()              optimizerD.step()  *############################*  *# (2) Update G network: optimize WGAN-GP objective, Equation (2)*  *###########################*  *for* p *in* netD.parameters():  *# reset requires\_grad*              p.requires\_grad = False  *# avoid computation*  *for* p *in* netMap.parameters():  *# reset requires\_grad*              p.requires\_grad = False  *for* p *in* F\_ha.parameters():  *# reset requires\_grad*              p.requires\_grad = False          netG.zero\_grad()          noise\_gen.normal\_(0, 1)          fake = netG(noise\_gen, input\_att)          embed\_fake, outz\_fake = netMap(fake)          criticG\_fake = netD(fake, input\_att)          criticG\_fake = criticG\_fake.mean()          G\_cost = -criticG\_fake          embed\_real, outz\_real = netMap(input\_res)          all\_outz = torch.cat((outz\_fake, outz\_real.detach()), *dim*=0)          fake\_ins\_contras\_loss = contras\_criterion(all\_outz, torch.cat((input\_label, input\_label), *dim*=0))          cls\_loss\_fake = class\_scores\_for\_loop(embed\_fake, input\_label, F\_ha)          errG = G\_cost + opt.ins\_weight \* fake\_ins\_contras\_loss + opt.cls\_weight \* cls\_loss\_fake  *# + opt.ins\_weight \* c\_errG*          errG.backward()          optimizerG.step()      F\_ha.zero\_grad()  *if* (epoch + 1) % opt.lr\_decay\_epoch == 0:  *for* param\_group *in* optimizerD.param\_groups:              param\_group['lr'] = param\_group['lr'] \* opt.lr\_dec\_rate  *for* param\_group *in* optimizerG.param\_groups:              param\_group['lr'] = param\_group['lr'] \* opt.lr\_dec\_rate      mean\_lossG /= data.ntrain / opt.batch\_size      mean\_lossD /= data.ntrain / opt.batch\_size      print(          '[%d/%d] Loss\_D: %.4f Loss\_G: %.4f, Wasserstein\_dist: %.4f, real\_ins\_contras\_loss:%.4f, fake\_ins\_contras\_loss:%.4f, cls\_loss\_real: %.4f, cls\_loss\_fake: %.4f'          % (epoch, opt.nepoch, D\_cost, G\_cost, Wasserstein\_D, real\_ins\_contras\_loss, fake\_ins\_contras\_loss, cls\_loss\_real, cls\_loss\_fake))  *# evaluate the model, set G to evaluation mode*      netG.eval()  *for* p *in* netMap.parameters():  *# reset requires\_grad*          p.requires\_grad = False  *if* opt.gzsl: *# Generalized zero-shot learning*          syn\_feature, syn\_label = generate\_syn\_feature(netG, data.unseenclasses, data.attribute, opt.syn\_num)          train\_X = torch.cat((data.train\_feature, syn\_feature), 0)          train\_Y = torch.cat((data.train\_label, syn\_label), 0)          nclass = opt.nclass\_all          cls = classifier\_embed\_contras.CLASSIFIER(train\_X, train\_Y, netMap, opt.embedSize, data, nclass, opt.cuda,                                                    opt.classifier\_lr, 0.5, 25, opt.syn\_num,                                                    True)          print('unseen=%.4f, seen=%.4f, h=%.4f' % (cls.acc\_unseen, cls.acc\_seen, cls.H))  *else*:  *# conventional zero-shot learning*          syn\_feature, syn\_label = generate\_syn\_feature(netG, data.unseenclasses, data.attribute, opt.syn\_num)          cls = classifier\_embed\_contras.CLASSIFIER(syn\_feature, util.map\_label(syn\_label, data.unseenclasses), netMap,                                                    opt.embedSize, data,                                                    data.unseenclasses.size(0), opt.cuda, opt.classifier\_lr, 0.5, 100,                                                    opt.syn\_num,                                                    False)          acc = cls.acc          print('unseen class accuracy=%.4f '%acc)  *# reset G to training mode*      netG.train()  *for* p *in* netMap.parameters():  *# reset requires\_grad*          p.requires\_grad = True |

# testing result

|  |
| --- |
| [375/2000] Loss\_D: 1.0740 Loss\_G: 3.1810, Wasserstein\_dist: 0.7987, real\_ins\_contras\_loss:1.8419, fake\_ins\_contras\_loss:3.3917, cls\_loss\_real: 0.0006, cls\_loss\_fake: 0.2597  unseen=0.6432, seen=0.6575, h=0.6503  [376/2000] Loss\_D: 1.1288 Loss\_G: 2.8772, Wasserstein\_dist: 0.7708, real\_ins\_contras\_loss:1.8697, fake\_ins\_contras\_loss:3.4364, cls\_loss\_real: 0.0005, cls\_loss\_fake: 0.2908  unseen=0.6335, seen=0.6516, h=0.6424  [377/2000] Loss\_D: 1.1302 Loss\_G: 2.9798, Wasserstein\_dist: 0.7613, real\_ins\_contras\_loss:1.8569, fake\_ins\_contras\_loss:3.4037, cls\_loss\_real: 0.0006, cls\_loss\_fake: 0.2960  unseen=0.6176, seen=0.6713, h=0.6434  [378/2000] Loss\_D: 1.1049 Loss\_G: 2.9658, Wasserstein\_dist: 0.7679, real\_ins\_contras\_loss:1.8403, fake\_ins\_contras\_loss:3.3991, cls\_loss\_real: 0.0005, cls\_loss\_fake: 0.2262  unseen=0.6465, seen=0.6474, h=0.6469  [379/2000] Loss\_D: 1.1267 Loss\_G: 2.9132, Wasserstein\_dist: 0.7659, real\_ins\_contras\_loss:1.8609, fake\_ins\_contras\_loss:3.3711, cls\_loss\_real: 0.0005, cls\_loss\_fake: 0.2390  unseen=0.6144, seen=0.6645, h=0.6385  [380/2000] Loss\_D: 1.1256 Loss\_G: 2.8771, Wasserstein\_dist: 0.7600, real\_ins\_contras\_loss:1.8550, fake\_ins\_contras\_loss:3.3810, cls\_loss\_real: 0.0005, cls\_loss\_fake: 0.2538  unseen=0.6271, seen=0.6540, h=0.6403  [381/2000] Loss\_D: 1.0960 Loss\_G: 3.1327, Wasserstein\_dist: 0.7859, real\_ins\_contras\_loss:1.8470, fake\_ins\_contras\_loss:3.3915, cls\_loss\_real: 0.0005, cls\_loss\_fake: 0.2275  unseen=0.6412, seen=0.6581, h=0.6496  [382/2000] Loss\_D: 1.0863 Loss\_G: 3.1254, Wasserstein\_dist: 0.7947, real\_ins\_contras\_loss:1.8434, fake\_ins\_contras\_loss:3.3735, cls\_loss\_real: 0.0004, cls\_loss\_fake: 0.2525  unseen=0.6488, seen=0.6633, h=0.6560  [383/2000] Loss\_D: 1.0970 Loss\_G: 3.0476, Wasserstein\_dist: 0.7740, real\_ins\_contras\_loss:1.8401, fake\_ins\_contras\_loss:3.3943, cls\_loss\_real: 0.0004, cls\_loss\_fake: 0.2985  unseen=0.6190, seen=0.6807, h=0.6484  [384/2000] Loss\_D: 1.1231 Loss\_G: 3.0560, Wasserstein\_dist: 0.7725, real\_ins\_contras\_loss:1.8575, fake\_ins\_contras\_loss:3.3495, cls\_loss\_real: 0.0004, cls\_loss\_fake: 0.2609  unseen=0.6691, seen=0.6278, h=0.6478  [385/2000] Loss\_D: 1.0989 Loss\_G: 3.0212, Wasserstein\_dist: 0.7798, real\_ins\_contras\_loss:1.8405, fake\_ins\_contras\_loss:3.3737, cls\_loss\_real: 0.0004, cls\_loss\_fake: 0.3067  unseen=0.6371, seen=0.6520, h=0.6445  [386/2000] Loss\_D: 1.0629 Loss\_G: 3.0410, Wasserstein\_dist: 0.7999, real\_ins\_contras\_loss:1.8340, fake\_ins\_contras\_loss:3.3487, cls\_loss\_real: 0.0004, cls\_loss\_fake: 0.2354  unseen=0.6462, seen=0.6530, h=0.6496  [387/2000] Loss\_D: 1.0829 Loss\_G: 3.1909, Wasserstein\_dist: 0.7980, real\_ins\_contras\_loss:1.8490, fake\_ins\_contras\_loss:3.3546, cls\_loss\_real: 0.0004, cls\_loss\_fake: 0.2954  unseen=0.6408, seen=0.6524, h=0.6465  [388/2000] Loss\_D: 1.1141 Loss\_G: 2.9498, Wasserstein\_dist: 0.7733, real\_ins\_contras\_loss:1.8529, fake\_ins\_contras\_loss:3.4305, cls\_loss\_real: 0.0004, cls\_loss\_fake: 0.2928  unseen=0.6637, seen=0.6260, h=0.6443  [389/2000] Loss\_D: 1.1211 Loss\_G: 3.1020, Wasserstein\_dist: 0.7749, real\_ins\_contras\_loss:1.8589, fake\_ins\_contras\_loss:3.3886, cls\_loss\_real: 0.0004, cls\_loss\_fake: 0.2805  unseen=0.6413, seen=0.6485, h=0.6449 |