GAN Training and Inference

December 11, 2021

[ ]:
```
# Updated GAN CODE
# Data pre-processing pipeline + GAN for the KITTI Dataset.
import torch
from torchvision import transforms
import numpy as np
import cv2
from glob import glob
import os
import matplotlib.pyplot as plt
from skimage import io
import torchvision.models as models
import torch
import numpy as np
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.autograd import Variable
import cv2
import matplotlib.pyplot as plt
from tqdm import tqdm
from glob import glob
import numpy as np
import torch.nn.init
import os
import torch
import numpy as np
```

[ ]:
```
from google.colab import drive
drive.mount(r'/content/drive/', force_remount = True) # mounting the drive.
```

[ ]:
```
basePath = r'/media/athrva/New Volume/cis520/Final Project/Processed_Data'
```

[ ]:
```
# Some Custom Transformations (adapted code)

import torch
import random
```

```
import numpy as np
from PIL import Image

'''Set of tranform random routines that takes list of inputs as arguments,
in order to have random but coherent transformations.'''


class Compose(object):
    def __init__(self, transforms):
        self.transforms = transforms

    def __call__(self, images):
        for t in self.transforms:
            images = t(images)
        return images


class Normalize(object):
    def __init__(self, mean, std):
        self.mean = mean
        self.std = std

    def __call__(self, images):
        for tensor in images:
            for t, m, s in zip(tensor, self.mean, self.std):
                t.sub_(m).div_(s)
        return images


class ArrayToTensor(object):
    def __call__(self, images):
        tensors = []
        for im in images:
            # put it from HWC to CHW format
            im = np.transpose(im, (2, 0, 1))
            tensors.append(torch.from_numpy(im).float()/255)
        return tensors
```

[ ]:
```
### Dataloaders using pytorch frameworks
from torch.utils.data import Dataset, DataLoader
from skimage.transform import resize


def depthByName(path):
    num = path.split('.jpg')[0].split(os.path.sep)[-1]
    return num
```

```
def imageByName(path):
    num = path.split('.jpg')[0].split(os.path.sep)[-1]
    return num

def sortPaths(imgPaths, dmapPaths):
    imgPaths = sorted(imgPaths, key = imageByName);
    dmapPaths = sorted(dmapPaths, key = depthByName);
    return imgPaths, dmapPaths


def validityCheckPairs(img_dmap_pairs): # checks the validity of the incoming␣
 ↪data (checks whether data is pairwise)
    counter = 0

    for pair in img_dmap_pairs:
      img_name = pair[0].split('/')[-1].split('.')[0].split('_')
      dmap_name = pair[1].split('/')[-1].split('.')[0].split('_')

      img_ID = img_name[1] + img_name[3]
      dmap_ID = dmap_name[1] + dmap_name[3]

      assert(img_ID == dmap_ID)

      counter += 1

    print(f'Test Passed : All data is pairwise. Number of Samples tested :␣
 ↪{counter}')
    print('====================================')


def buildPairs(dmaps, imgs):
    pairs = []
    for i in imgs:
        img_name = i.split('/')[-1].split('.')[0].split('_')
        dmap_path = basePath + '/dmaps/' + img_name[0] + '_' + img_name[1] + \
                            '_' + 'dmap' + '_' + img_name[3] + '.jpg'

        if dmap_path in dmaps:
            pairs.append([i, dmap_path])
    return pairs

def getDataPaths(): # Gets the paths were data is stored.
    Directories = glob(basePath + os.path.sep + '*')

    imgPathsDir = glob(Directories[0] + os.path.sep + '*.jpg');
    dmapPathsDir = glob(Directories[1] + os.path.sep + '*.jpg');
    imgPathsDir, dmapPathsDir = sortPaths(imgPathsDir, dmapPathsDir);
```

```
    # Figure out which of the two has fewer entries
    # bottleneck = np.argmin([len(imgPathsDir), len(dmapPathsDir)])
    # For now, we know that images has fewer entries than dmaps
    img_dmap_pairs = buildPairs(imgPathsDir, dmapPathsDir)

    validityCheckPairs(img_dmap_pairs)
    # Check again to be sure
    validityCheckPairs(img_dmap_pairs)

    return img_dmap_pairs

def rgb2gray(rgb):

    r, g, b = rgb[:,:,0], rgb[:,:,1], rgb[:,:,2]
    gray = 0.2989 * r + 0.5870 * g + 0.1140 * b

    return gray

## Image Dataloader
class ImageDataset(Dataset):

    def __init__(self,
                 paths,
                 op,
                 transforms=None):
        """
        Args:
            op (str): "train", "val", or "test" to indicate the split type
            transforms (list or None): Image transformations to apply upon␣
 ↪loading.
        """
        self.paths = paths
        self.transform = transforms
        self.op = op

        self.num_ex = len(paths)

        try:
            if self.op == 'train':
                self.split_paths = self.paths[0:int(0.6*self.num_ex)]
            elif self.op == 'val':
                self.split_paths = self.paths[int(0.6*self.num_ex):int(0.8*self.
 ↪num_ex)]
            elif self.op == 'test':
                self.split_paths = self.paths[int(0.8*self.num_ex):int(self.
 ↪num_ex)]
```

```
        except ValueError:
            print('op is not train, val, or test')

    def __len__(self):
        return len(self.split_paths)

    def __getitem__(self, idx):
        dmap = io.imread(self.split_paths[idx][1])
        img = io.imread(self.split_paths[idx][0])

        img = resize(img, (img.shape[0] // 2, img.shape[1] // 2),
                     anti_aliasing=True)
        dmap = resize(dmap, (dmap.shape[0] // 2, dmap.shape[1] // 2),
                     anti_aliasing=True)
        dmap = np.expand_dims(rgb2gray(dmap),0)
        if self.transform:
          img, dmap = self.img_transform(img, dmap)

        sample = {'img': img, 'dmap': dmap}
        return sample

    def img_transform(self, img, dmap):
        ## Apply Transformations
        img = self.transform(img)
        dmap = torch.from_numpy(dmap).type(torch.float32)
        return img, dmap
```

[ ]:
```
data_paths = getDataPaths()
img_transform = transforms.Compose([
        transforms.ToTensor(),
])

train_dataset = ImageDataset(data_paths, op="train", transforms=img_transform)
val_dataset = ImageDataset(data_paths, op="val", transforms=img_transform)
test_dataset = ImageDataset(data_paths, op="test", transforms=img_transform)

print(train_dataset.__len__())
print(val_dataset.__len__())
print(test_dataset.__len__())

train_batch_size = 1
val_batch_size = 1
test_batch_size = 1

train_dataloader = DataLoader(train_dataset, batch_size=train_batch_size,␣
 ↪shuffle=False)
```

```
validation_dataloader = DataLoader(val_dataset, batch_size=val_batch_size,␣
 ↪shuffle=False)
test_dataloader = DataLoader(test_dataset, batch_size=1, shuffle=False)

train_dataset_notensor = ImageDataset(data_paths, op="train", transforms=None)
train_dataloader_notensor = DataLoader(train_dataset_notensor,␣
 ↪batch_size=train_batch_size, shuffle=False)

# Load some image and view them
for k,data in enumerate(train_dataloader_notensor):
  plt.figure()
  plt.imshow(data['img'][0])
  plt.figure()
  plt.imshow(data['dmap'][0][0])
  plt.show()
  if k >= 3:
        break
```

[ ]:
```
### UNET FROM https://pytorch.org/hub/
 ↪mateuszbuda_brain-segmentation-pytorch_unet/
from collections import OrderedDict
import torch
import torch.nn as nn

### Model definitions.

class pretrainedG():
    def __init__(self,in_channels, out_channels, init_features):
        self.in_channels = in_channels;
        self.out_channels = out_channels;
        self.init_features = init_features;
    def get(self):
        gM = torch.hub.load('mateuszbuda/brain-segmentation-pytorch', 'unet',
            in_channels=self.in_channels, out_channels=self.out_channels,
             init_features=self.init_features, pretrained=True)
        return gM

class Discriminator(nn.Module): # The PatchGAN discriminator
    def __init__(self, in_channels=1):
        super(Discriminator, self).__init__()

        def discriminator_block(in_filters, out_filters, normalization=True):
            """Returns downsampling layers of each discriminator block"""
            layers = [nn.Conv2d(in_filters, out_filters, 4, stride=2,␣
 ↪padding=1)]
            if normalization:
                layers.append(nn.InstanceNorm2d(out_filters))
```

```
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            return layers

        self.gM = nn.Sequential(
            *discriminator_block(in_channels * 2, 64, normalization=False),
            *discriminator_block(64, 128),
            *discriminator_block(128, 256),
            *discriminator_block(256, 512),
            nn.ZeroPad2d((1, 0, 1, 0)),
            nn.Conv2d(512, 1, 4, padding=1, bias=False)
        )

    def forward(self, img_A, img_B):
        # Concatenate image and condition image by channels to produce input
        img_input = torch.cat((img_A, img_B), 1)
        return self.gM(img_input)

class pretrainedD(): # Pretrained ResNet 18 as discriminator (Not Used)
    def __init__(self, in_channels, out_channels, init_features):
        self.in_channels = in_channels;
        self.out_channels = out_channels;
        self.init_features = init_features;
    def get(self):
        model = torch.hub.load('pytorch/vision', 'resnet18',pretrained=True)
        return model
patch = (1, 128 //16, 416 //16)
```

[ ]:
```
print(*patch)
```

[ ]:
```
## Initialize Optimizer and Learning Rate Scheduler

EPOCHS = 30
VISUALIZE = True
TRAIN = 1
MODEL_SAVE_DIR = r'/media/athrva/New Volume/cis520/Final Project/TrainedGAN/'
MODEL_SAVE_DIR_RUD = r'/media/athrva/New Volume/cis520/Final Project/GAN_MODELS/
 ↪'
TRAIN_LOG_DIR_RUD = r'/media/athrva/New Volume/cis520/Final Project/'
TRAIN_LOG_DIR = r'/media/athrva/New Volume/cis520/Final Project/'

learning_rate = 1e-4
num_epochs = 1


loss_metric = torch.nn.MSELoss() # For regression task

if torch.cuda.is_available():
```

```
    gpu_boole = True
else:
    gpu_boole = False
```

[ ]:
```
# get logger

use_cuda = torch.cuda.is_available()


trainLogger = open(TRAIN_LOG_DIR + 'train.log', 'a+') # Logger


def sc1(tensor):
    tensor = (tensor[0,0,:,:] + tensor[0,1,:,:] + tensor[0,2,:,:])/3.

    return tensor.unsqueeze(0).unsqueeze(0)

cuda = True if torch.cuda.is_available() else False

Tensor = torch.cuda.FloatTensor if cuda else torch.FloatTensor # Generic␣
 ↪definition of Tensor default type


if TRAIN == 1:
    print('Pre-trained Model Found...Reusing')

    gM = pretrainedG(in_channels=3, out_channels= 1, init_features= 32)
    gM = gM.get()
    dM = Discriminator(in_channels = 1)
    if use_cuda:
        gM.cuda()
        dM.cuda()
    device = torch.device('cuda')
    if os.path.exists(MODEL_SAVE_DIR + '/SegNet.pt'):
        checkpoint = torch.load(MODEL_SAVE_DIR + '/SegNet.pt',␣
 ↪map_location=device)
        gM.load_state_dict(checkpoint['model_state_dict'])
        dM.load_state_dict(checkpoint['dis_state_dict'])
        optimizer_G = torch.optim.Adam(gM.parameters(), lr=learning_rate)
        optimizer_D = torch.optim.Adam(dM.parameters(), lr=learning_rate/2) #␣
 ↪Half learning rate for the discriminator
        optimizer_G.load_state_dict(checkpoint['G_optimizer_state_dict'])
        optimizer_D.load_state_dict(checkpoint['D_optimizer_state_dict'])
        schedulerG = torch.optim.lr_scheduler.StepLR(optimizer_G, step_size=10,␣
 ↪gamma=0.1)
        schedulerD = torch.optim.lr_scheduler.StepLR(optimizer_D, step_size=10,␣
 ↪gamma=0.1)
```

```
        dM.train()
        gM.train()
    else:
        if use_cuda:
            gM.cuda()
            dM.cuda()
        optimizer_G = torch.optim.Adam(gM.parameters(), lr=learning_rate)
        optimizer_D = torch.optim.Adam(dM.parameters(), lr=learning_rate/2)
        schedulerG = torch.optim.lr_scheduler.StepLR(optimizer_G, step_size=10,␣
↪gamma=0.1)
        schedulerD = torch.optim.lr_scheduler.StepLR(optimizer_D, step_size=10,␣
↪gamma=0.1)

        dM.train()
        gM.train()


    # similarity loss definition
    loss_fn = torch.nn.CrossEntropyLoss()
    criterion_GAN = torch.nn.L1Loss()

    # continuity loss definition
    loss_hpy = torch.nn.L1Loss(size_average = True)
    loss_hpy_ = torch.nn.MSELoss()

    a = torch.Tensor([[1, 0, -1],
    [2, 0, -2],
    [1, 0, -1]]) # Kernel for gradient computation

    a = a.view((1,1,3,3)).cuda()

    b = torch.Tensor([[1, 2, 1],
    [0, 0, 0],
    [-1, -2, -1]]) # Kernel for gradient computation

    b = b.view((1,1,3,3)).cuda()


    sig = torch.nn.Sigmoid()

    for epoch in range(EPOCHS):
        for batch_idx, batch in enumerate(train_dataloader):

            x = batch["img"]
            y = batch["dmap"]
            if gpu_boole:
                x = x.cuda().float()
```

```
                y = y.cuda().float()

            epoch_loss = 0

            real_A, real_B = x, y
            real_B = real_B.cuda()
            real_A = real_A.cuda()
            real_B = real_B

            optimizer_G.zero_grad()

            fake_B = gM(real_A)
            fake_B = sig(fake_B) # pass the model output through a sigmoid␣
↪(optional)

            # Adversarial ground truths
            valid = Variable(Tensor(np.ones((real_A.size(0), *patch))),␣
↪requires_grad=False)
            fake = Variable(Tensor(np.zeros((real_A.size(0), *patch))),␣
↪requires_grad=False)



            pred_fake = dM(fake_B, sc1(real_A))

            """Code for the novel gradient loss"""

            G_x = F.conv2d(real_B, a)
            G_y = F.conv2d(real_B, b)
            F_x = F.conv2d(fake_B, a)
            F_y = F.conv2d(fake_B, b)
            G = sig((torch.pow(G_x,2)/torch.max(torch.pow(G_x,2)))+ (torch.
↪pow(G_y,2)/torch.max(torch.pow(G_y,2))))
            F_ = sig((torch.pow(F_x,2)/torch.max(torch.pow(F_x,2)))+ (torch.
↪pow(F_y,2)/torch.max(torch.pow(F_y,2))))


            lhpy = loss_hpy(fake_B,real_B)
            lhpy_ = loss_hpy_(fake_B,real_B)

            # Total loss : It has a few components
            loss_G = 0.5*torch.sum(torch.abs(torch.abs(G) - torch.abs(F_))) +␣
↪1*(lhpy + lhpy_) + 0.5*loss_fn(pred_fake, valid) +␣
↪3*criterion_GAN(pred_fake, valid)

            loss_G.backward()
```

```
            epoch_loss += loss_G.item()
            optimizer_G.step()

            if batch_idx % 5 == 0: # Update the discriminator every 5 batches␣
↪instead of 1.
                optimizer_D.zero_grad()
                #pass
                # Real loss
                pred_real = dM(real_B, sc1(real_A))
                loss_real = criterion_GAN(pred_real,valid)

                # Fake loss
                pred_fake = dM(fake_B.detach(), sc1(real_A))
                loss_fake = criterion_GAN(pred_fake, fake)

                # Total loss
                loss_D = (loss_real + loss_fake)
                loss_D.backward()
                optimizer_D.step()

            trainLogger.write(f'{loss_G.item()}'+','+f'{loss_D.item()}'+'\n')
            with open(TRAIN_LOG_DIR_RUD + 'training_info.txt','a+') as file: #␣
↪redundant logger
                file.write(f'G Loss : {loss_G.item()}, D Loss : {loss_D.
↪item()}')

            if VISUALIZE:
                im_target = fake_B.detach().cpu().numpy()[0]
                im_target = im_target/abs(im_target.max())
                im_targetgt = real_B.detach().cpu().numpy()[0]
                im_target_rgb = im_target
                im_real = x[0].detach().cpu().numpy()
                im_target_rgb = np.vstack([rgb2gray(np.
↪moveaxis(im_real,0,-1)),im_targetgt[0],
                                           im_target_rgb[0]])
                cv2.imshow('Output : ', im_target_rgb)
                cv2.waitKey(5)


        print('Current Epoch Loss : ', epoch_loss)

        print (batch_idx, '/', EPOCHS, '|',' | loss G :', loss_G.item(), '|',␣
↪'| loss D :', loss_D.item())

        torch.save({'model_state_dict' : gM.state_dict(),
                    'G_optimizer_state_dict' : optimizer_G.state_dict(),
                    'D_optimizer_state_dict' : optimizer_D.state_dict(),
```

```
                        'dis_state_dict' : dM.state_dict()}, MODEL_SAVE_DIR_RUD␣
↪+ f'SegNet_17{epoch}.pt');
        torch.save({'model_state_dict' : gM.state_dict(),
                        'G_optimizer_state_dict' : optimizer_G.state_dict(),
                        'D_optimizer_state_dict' : optimizer_D.state_dict(),
                        'dis_state_dict' : dM.state_dict()}, MODEL_SAVE_DIR +␣
↪'SegNet.pt');
```

[ ]:
```
## PREDICTION PIPELINE.
# This is a simple script to visualize the model's performance on the data␣
↪images.

use_cuda = torch.cuda.is_available()
PREDICT = 1

once = True

def sc1(tensor):
    tensor = (tensor[0,0,:,:] + tensor[0,1,:,:] + tensor[0,2,:,:])/3.
    return tensor.unsqueeze(0).unsqueeze(0)

cuda = True if torch.cuda.is_available() else False

Tensor = torch.cuda.FloatTensor if cuda else torch.FloatTensor


if PREDICT == 1:
    print('Pre-trained Model Found...Reusing')

    gM = pretrainedG(in_channels=3, out_channels= 1, init_features= 32)
    gM = gM.get()
    dM = Discriminator(in_channels = 1)
    if use_cuda:
        gM.cuda()
        dM.cuda()
    device = torch.device('cuda')
    if os.path.exists(MODEL_SAVE_DIR + '/SegNet.pt'):
        checkpoint = torch.load(MODEL_SAVE_DIR + '/SegNet.pt',␣
↪map_location=device)
        gM.load_state_dict(checkpoint['model_state_dict'])
        dM.load_state_dict(checkpoint['dis_state_dict'])
        optimizer_G = torch.optim.Adam(gM.parameters(), lr=learning_rate)
        optimizer_D = torch.optim.Adam(dM.parameters(), lr=learning_rate/2)
        optimizer_G.load_state_dict(checkpoint['G_optimizer_state_dict'])
        optimizer_D.load_state_dict(checkpoint['D_optimizer_state_dict'])
        schedulerG = torch.optim.lr_scheduler.StepLR(optimizer_G, step_size=10,␣
↪gamma=0.1)
```

```
        schedulerD = torch.optim.lr_scheduler.StepLR(optimizer_D, step_size=10,␣
↪gamma=0.1)
        dM.train()
        gM.train()
    else:
        print('No Model Found...')
        dM.train()
        gM.train()


    sig = torch.nn.Sigmoid()

    for epoch in range(1):
        for batch_idx, batch in enumerate(validation_dataloader): # could be␣
↪test_dataloader.


            x = batch["img"]
            y = batch["dmap"]
            if gpu_boole:
                x = x.cuda().float()
                y = y.cuda().float()

            epoch_loss = 0

            real_A, real_B = x, y
            real_B = real_B.cuda()
            real_A = real_A.cuda()
            real_B = real_B

            optimizer_G.zero_grad()

            fake_B = gM(real_A)
            im_target = fake_B.detach().cpu().numpy()[0]
            im_target = im_target/abs(im_target.max())
            im_targetgt = real_B.detach().cpu().numpy()[0]
            im_target_rgb = im_target
            im_real = x[0].detach().cpu().numpy()
            im_target_rgb = np.vstack([im_targetgt[0],
                                       im_target_rgb[0]])
            if once == True:
                plt.imsave(basePath + os.path.sep + 'output.png', im_target_rgb)
                once = False
            cv2.imshow('Output : ', im_target_rgb)
            cv2.waitKey(5)
            plt.imsave(f'/media/athrva/New Volume/cis520/Final Project/
↪Progression/out_{batch_idx}.png', im_target_rgb)
```