Image to Image translation in Pytorch

By Vikrant Dey

Image-to-image translation is a popular topic in the field of image processing and computer vision. The basic idea behind this is to map a source input image to a target output image using a set of image pairs. Some of the applications include object transfiguration, style transfer, and image in-painting.

The earliest methods used for such translations incorporated the use of Convolutional Neural Networks (CNNs). This approach minimized the loss of a pixel value between the images. But it could not produce photo-realistic images. So, recently Generative Adversarial Networks (GANs) have been of much use to the cause. Since GANs utilize adversarial feedback, the quality of image translation has improved quite a lot.

Now, this problem of image translation comes with various constraints as data can be paired as well as unpaired. Paired data have training examples with one to one correspondence, while unpaired data have no such mapping. In this tutorial, we shall see how we can create models for both paired and unpaired data. We shall use a Pix2Pix GAN for paired data and then a CycleGAN for unpaired data.

Now enough of theories; let us jump into the coding part. First, we shall discuss how to create a Pix2Pix GAN model and then a CycleGAN model.

Pix2Pix for Paired Data

The GAN architecture consists of a generator and a discriminator. The generator outputs new synthetic images while the discriminator differentiates between the real and fake (generated) images. So, this betters the quality of the images. The Pix2Pix model discussed here is a type of conditional GAN (also known as cGAN). The output image is generated conditioned on the input image. The discriminator is fed both the input and output images. Then it has to decide if the target is a variated and transformed version of the source. Then, ‘Adversarial losses’ train the generator and the ‘L1 losses’ between the generated and target images update the generator.

Applications of Pix2Pix GAN include conversion of satellite images to maps, black and white photographs to colored ones, sketches to real photos, and so on. In this tutorial, we shall discuss how to convert sketches of shoes to actual photos of shoes.

We are going to use the edges2shoes dataset which can be downloaded from the
link: https://people.eecs.berkeley.edu/~tinghuiz/projects/pix2pix/datasets/edges2shoes.tar.gz

This dataset contains train and test sets of pairs of two figures in each. One is the edged outline of a shoe and the other is the original image of the shoe. Our task is to create a Pix2Pix GAN model from the data so that we can translate the outlines into real pictures of the shoes.

First, we download the dataset. Then we should separate the train and test folders from being in the same folder directory to different folders. For saving the log, we can create a separate folder, though this is optional. After that, we dive into the code.

Importing necessary libraries and modules

```
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import random
import math
import io
from PIL import Image
from copy import deepcopy
from IPython.display import HTML
import torch
import torchvision
import torchvision.transforms as transforms
import torchvision.utils as vutils
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
manual_seed = ...
random.seed(manual_seed)
torch.manual_seed(manual_seed)
```

For working with the train and test data, we need to create data loaders. Also, we enter the necessary transformations and data inputs.

```
log_path = os.path.join("...") #Enter the log saving directory here
data_path_Train = os.path.dirname('...') #Enter the train folder directory
data_path_Test = os.path.dirname('...') #Enter the test folder directory
batch_size = 4
num_workers = 2
transform = transforms.Compose([transforms.Resize((256,512)),
                                transforms.ToTensor(),
                                transforms.Normalize((0.5,),(0.5,)),])
load_Train = torch.utils.data.DataLoader(torchvision.datasets.ImageFolder(root=
             data_path_Train, transform=transform), batch_size=batch_size,
             shuffle=True, num_workers=num_workers)

load_Test = torch.utils.data.DataLoader(torchvision.datasets.ImageFolder(root=
            data_path_Test, transform=transform), batch_size=batch_size,
            shuffle = False, num_workers=num_workers)
```

Now we shall try to view how the images in the batches look like. We have to iterate the objects in the train data loader for viewing one at a time. Then for creating the batches, we have to split the data loader.

```
def show_E2S(batch1, batch2, title1, title2):
    # edges
    plt.figure(figsize=(15,15))
    plt.subplot(1,2,1)
    plt.axis("off")
    plt.title(title1)
    plt.imshow(np.transpose(vutils.make_grid(batch1, nrow=1, padding=5,
    normalize=True).cpu(),(1,2,0)))
    # shoes
    plt.subplot(1,2,2)
    plt.axis("off")
    plt.title(title2)
    plt.imshow(np.transpose(vutils.make_grid(batch2, nrow=1, padding=5,
    normalize=True).cpu(),(1,2,0)))
def split(img):
    return img[:,:,:,:256], img[:,:,:,256:]
r_train, _ = next(iter(load_Train)
X, y = split(r_train.to(device), 256)
show_E2S(X,y,"input X (edges)","ground truth y (shoes)")
```

Output:

Building blocks of architecture

Here comes the main functional part of the code. Convolutional blocks, together with transposed convolutional blocks for upsampling, are defined here. In the later sections, we have to use these extensively.

```
inst_norm = True if batch_size==1 else False  # instance normalization
def conv(in_channels, out_channels, kernel_size, stride=1, padding=0):
    return nn.Conv2d(in_channels, out_channels, kernel_size, stride=stride,
    padding=padding)
def conv_n(in_channels, out_channels, kernel_size, stride=1, padding=0, inst_norm=Fal
    if inst_norm == True:
        return nn.Sequential(nn.Conv2d(in_channels, out_channels, kernel_size,
        stride=stride, padding=padding), nn.InstanceNorm2d(out_channels,
        momentum=0.1, eps=1e-5),)
    else:
        return nn.Sequential(nn.Conv2d(in_channels, out_channels, kernel_size,
        stride=stride, padding=padding), nn.BatchNorm2d(out_channels,
        momentum=0.1, eps=1e-5),)
def tconv(in_channels, out_channels, kernel_size, stride=1, padding=0, output_padding
    return nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride=stride,
    padding=padding, output_padding=output_padding)

def tconv_n(in_channels, out_channels, kernel_size, stride=1, padding=0, output_paddi
    if inst_norm == True:
        return nn.Sequential(nn.ConvTranspose2d(in_channels, out_channels, kernel_siz
        stride=stride, padding=padding, output_padding=output_padding),
        nn.InstanceNorm2d(out_channels, momentum=0.1, eps=1e-5),)
    else:
        return nn.Sequential(nn.ConvTranspose2d(in_channels, out_channels, kernel_siz
        stride=stride, padding=padding, output_padding=output_padding),
        nn.BatchNorm2d(out_channels, momentum=0.1, eps=1e-5),)
```

The generator model here is basically a U-Net model. It is an encoder-decoder model with skip connections between encoder and decoder layers having same-sized feature maps. For the encoder, we have first the Conv layer, then the Batch_norm layer, and then the Leaky ReLU layer. For the decoder, we have first the Transposed Conv layer, then the Batchnorm layer, and then the (Dropout) and ReLU layers. To merge the layers with skip connections, we use the torch.cat() function.

```
dim_c = 3
dim_g = 64
# Generator
class Gen(nn.Module):
```

```
    def __init__(self, inst_norm=False):
        super(Gen,self).__init__()
        self.n1 = conv(dim_c, dim_g, 4, 2, 1)
        self.n2 = conv_n(dim_g, dim_g*2, 4, 2, 1, inst_norm=inst_norm)
        self.n3 = conv_n(dim_g*2, dim_g*4, 4, 2, 1, inst_norm=inst_norm)
        self.n4 = conv_n(dim_g*4, dim_g*8, 4, 2, 1, inst_norm=inst_norm)
        self.n5 = conv_n(dim_g*8, dim_g*8, 4, 2, 1, inst_norm=inst_norm)
        self.n6 = conv_n(dim_g*8, dim_g*8, 4, 2, 1, inst_norm=inst_norm)
        self.n7 = conv_n(dim_g*8, dim_g*8, 4, 2, 1, inst_norm=inst_norm)
        self.n8 = conv(dim_g*8, dim_g*8, 4, 2, 1)
        self.m1 = tconv_n(dim_g*8, dim_g*8, 4, 2, 1, inst_norm=inst_norm)
        self.m2 = tconv_n(dim_g*8*2, dim_g*8, 4, 2, 1, inst_norm=inst_norm)
        self.m3 = tconv_n(dim_g*8*2, dim_g*8, 4, 2, 1, inst_norm=inst_norm)
        self.m4 = tconv_n(dim_g*8*2, dim_g*8, 4, 2, 1, inst_norm=inst_norm)
        self.m5 = tconv_n(dim_g*8*2, dim_g*4, 4, 2, 1, inst_norm=inst_norm)
        self.m6 = tconv_n(dim_g*4*2, dim_g*2, 4, 2, 1, inst_norm=inst_norm)
        self.m7 = tconv_n(dim_g*2*2, dim_g*1, 4, 2, 1, inst_norm=inst_norm)
        self.m8 = tconv(dim_g*1*2, dim_c, 4, 2, 1)
        self.tanh = nn.Tanh()
    def forward(self,x):
        n1 = self.n1(x)
        n2 = self.n2(F.leaky_relu(n1, 0.2))
        n3 = self.n3(F.leaky_relu(n2, 0.2))
        n4 = self.n4(F.leaky_relu(n3, 0.2))
        n5 = self.n5(F.leaky_relu(n4, 0.2))
        n6 = self.n6(F.leaky_relu(n5, 0.2))
        n7 = self.n7(F.leaky_relu(n6, 0.2))
        n8 = self.n8(F.leaky_relu(n7, 0.2))
        m1 = torch.cat([F.dropout(self.m1(F.relu(n8)), 0.5, training=True), n7], 1)
        m2 = torch.cat([F.dropout(self.m2(F.relu(m1)), 0.5, training=True), n6], 1)
        m3 = torch.cat([F.dropout(self.m3(F.relu(m2)), 0.5, training=True), n5], 1)
        m4 = torch.cat([self.m4(F.relu(m3)), n4], 1)
        m5 = torch.cat([self.m5(F.relu(m4)), n3], 1)
        m6 = torch.cat([self.m6(F.relu(m5)), n2], 1)
        m7 = torch.cat([self.m7(F.relu(m6)), n1], 1)
        m8 = self.m8(F.relu(m7))
        return self.tanh(m8)
```

The discriminator used here is a PatchGAN model. It chops the image into overlapping pixel images or patches. The discriminator works on each patch and averages the result. Then we create a function for initialization of weights.

```
dim_d = 64
# Discriminator
class Disc(nn.Module):
    def __init__(self, inst_norm=False):
        super(Disc,self).__init__()
        self.c1 = conv(dim_c*2, dim_d, 4, 2, 1)
        self.c2 = conv_n(dim_d, dim_d*2, 4, 2, 1, inst_norm=inst_norm)
```

```
        self.c3 = conv_n(dim_d*2, dim_d*4, 4, 2, 1, inst_norm=inst_norm)
        self.c4 = conv_n(dim_d*4, dim_d*8, 4, 1, 1, inst_norm=inst_norm)
        self.c5 = conv(dim_d*8, 1, 4, 1, 1)
        self.sigmoid = nn.Sigmoid()
    def forward(self, x, y):
        xy=torch.cat([x,y],dim=1)
        xy=F.leaky_relu(self.c1(xy), 0.2)
        xy=F.leaky_relu(self.c2(xy), 0.2)
        xy=F.leaky_relu(self.c3(xy), 0.2)
        xy=F.leaky_relu(self.c4(xy), 0.2)
        xy=self.c5(xy)
        return self.sigmoid(xy)
def weights_init(z):
    cls_name =z.__class__.__name__
    if cls_name.find('Conv')!=-1 or cls_name.find('Linear')!=-1:
        nn.init.normal_(z.weight.data, 0.0, 0.02)
        nn.init.constant_(z.bias.data, 0)
    elif cls_name.find('BatchNorm')!=-1:
        nn.init.normal_(z.weight.data, 1.0, 0.02)
        nn.init.constant_(z.bias.data, 0)
```

The model is a binary classification model since it predicts only two results: real or fake. So we use BCE loss. We also need to calculate L1 losses to find the deviation between the expected and translated images. Then we use Adam optimizer for both the generator and discriminator.

```
BCE = nn.BCELoss() #binary cross-entropy
L1 = nn.L1Loss()
#instance normalization
Gen = Gen(inst_norm).to(device)
Disc = Disc(inst_norm).to(device)
#optimizers
Gen_optim = optim.Adam(Gen.parameters(), lr=2e-4, betas=(0.5, 0.999))
Disc_optim = optim.Adam(Disc.parameters(), lr=2e-4, betas=(0.5, 0.999))
```

Now we shall view one instance of the input and target images along with the predicted image before training our model.

```
fix_con, _ = next(iter(load_Test)
fix_con = fix_con.to(device)
fix_X, fix_y = split(fix_con)
def compare_batches(batch1, batch2, title1, title2, batch3=None, title3):
    # batch1
    plt.figure(figsize=(15,15))
    plt.subplot(1,3,1)
    plt.axis("off")
    plt.title(title1)
    plt.imshow(np.transpose(vutils.make_grid(batch1, nrow=1, padding=5,
    normalize=True).cpu(), (1,2,0)))
    # batch2
    plt.subplot(1,3,2)
    plt.axis("off")
    plt.title(title2)
    plt.imshow(np.transpose(vutils.make_grid(batch2, nrow=1, padding=5,
    normalize=True).cpu(), (1,2,0)))
    # third batch
    if batch3 is not None:
        plt.subplot(1,3,3)
        plt.axis("off")
        plt.title(title3)
        plt.imshow(np.transpose(vutils.make_grid(batch3, nrow=1, padding=5,
        normalize=True).cpu(), (1,2,0)))
with torch.no_grad():
    fk = Gen(fix_X)
compare_batches(fix_X, fk, "input image", "prediction", fix_y, "ground truth")
```

Output:

Training the model

After the generator generates an output, the discriminator first works on the input image and the generated image. Then it works on the input image and the output image. After that, we calculate the generator and the discriminator losses. The L1 loss is a regularizing term and a hyperparameter known as *'lambda'* weighs it. Then we add the looses together.

```
loss = adversarial_loss + lambda * L1_loss
```

```
img_list = []
Disc_losses = Gen_losses = Gen_GAN_losses = Gen_L1_losses = []
iter_per_plot = 500
epochs = 5
L1_lambda = 100.0
for ep in range(epochs):
    for i, (data, _) in enumerate(load_Train):
        size = data.shape[0]
        x, y = split(data.to(device), 256)
        r_masks = torch.ones(size,1,30,30).to(device)
        f_masks = torch.zeros(size,1,30,30).to(device)
        # disc
        Disc.zero_grad()
        #real_patch
        r_patch=Disc(y,x)
        r_gan_loss=BCE(r_patch,r_masks)
        fake=Gen(x)
        #fake_patch
        f_patch = Disc(fake.detach(),x)
        f_gan_loss=BCE(f_patch,f_masks)
        Disc_loss = r_gan_loss + f_gan_loss
        Disc_loss.backward()
        Disc_optim.step()
        # gen
        Gen.zero_grad()
        f_patch = Disc(fake,x)
        f_gan_loss=BCE(f_patch,r_masks)
        L1_loss = L1(fake,y)
        Gen_loss = f_gan_loss + L1_lambda*L1_loss
        Gen_loss.backward()

        Gen_optim.step()
        if (i+1)%iter_per_plot == 0 :

            print('Epoch [{}/{}], Step [{}/{}], disc_loss: {:.4f}, gen_loss: {:.4f},D

            Gen_losses.append(Gen_loss.item())
            Disc_losses.append(Disc_loss.item())
            Gen_GAN_losses.append(f_gan_loss.item())
            Gen_L1_losses.append(L1_loss.item())
            with torch.no_grad():
                Gen.eval()
```

```
            fake = Gen(fix_X).detach().cpu()
            Gen.train()
        figs=plt.figure(figsize=(10,10))
        plt.subplot(1,3,1)
        plt.axis("off")
        plt.title("input image")
        plt.imshow(np.transpose(vutils.make_grid(fix_X, nrow=1, padding=5,
        normalize=True).cpu(), (1,2,0)))
        plt.subplot(1,3,2)
        plt.axis("off")
        plt.title("generated image")
        plt.imshow(np.transpose(vutils.make_grid(fake, nrow=1, padding=5,
        normalize=True).cpu(), (1,2,0)))

        plt.subplot(1,3,3)
        plt.axis("off")
        plt.title("ground truth")
        plt.imshow(np.transpose(vutils.make_grid(fix_y, nrow=1, padding=5,
        normalize=True).cpu(), (1,2,0)))

        plt.savefig(os.path.join(log_PATH,modelName+"-"+str(ep) +".png"))
        plt.close()
        img_list.append(figs)
```

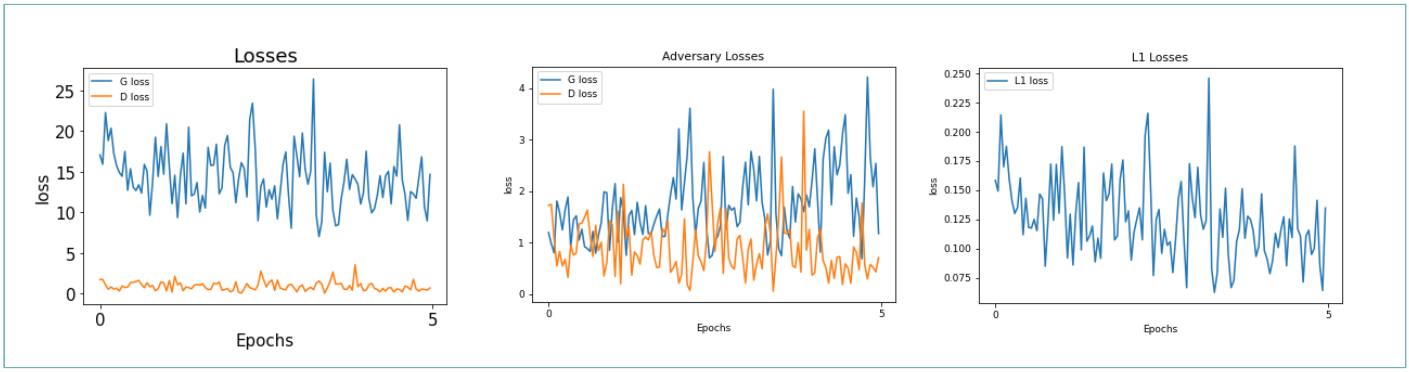
An image list ‘img_list’ is created. So, if you want to create a GIF to illustrate the training procedure, you can do it by making use of the list. Moving on to the last section, we shall now view our predictions.

```
t_batch, _ =  next(iter(load_Test))
t_x, t_y = batch_data_split(t_batch, 256)
with torch.no_grad():
    Gen.eval()
    fk_batch=G(t_x.to(device))
compare_batches(t_x, fk_batch, "input images", "predicted images", t_y, "ground truth
```

Output:

The number of epochs used here is only 5. Hence the predictions are a lot less realistic than expected. If you increase the number of epochs to 30 or more, the results will be astonishing. But it takes a lot of time to accomplish that.

The losses for this training are illustrated here:



You can easily create the plots from the expressions given above. But, if you face any difficulty in plotting the data, you should look up this tutorial: https://www.codespeedy.com/plotting-mathematical-expression-using-matplotlib-in-python/