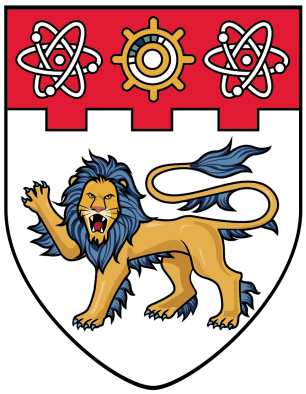

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

AI6121 Project:
Image Translation and UDA

Rao Yixi, G2302775D
Jin Zhixiao, G2303771H
Huang Siwei, G2304149C

School Of Computer Science And Engineering

18/11/2023

Contents

1 Introduction

2 Image-to-Image Translation

The image to image translation model we chose is Cycle Generative Adversarial Network (CycleGAN) developed by Zhu et al [1].

2.1 Introduction

For image-to-image translation task, an original image set S and a target image set T are required. For most of the current I2I models, the two image sets used for training need to be paired, meaning that for every image $s \in S$, there must be a corresponding image $t \in T$. However, this kind of paired image set is scarce because (1) Manual production is time-consuming and laborious (2) The target image set required by some vision and graphics tasks does not exist, such as image translation between painters of different styles. The CycleGAN model proposed by Zhu et al. is designed to solve the unpaired image-to-image translation problem.

The CycleGAN model needs to learn a mapping $G : S \rightarrow T$ between an input image and an output image, which can take the special features of the original image set, and then figure out how to transform these features to the target image set, making it difficult to distinguish between the translated image distribution and the target image set distribution. However, this kind of unpaired unidirectional I2I translation will have an under-constrained problem, that is, the translated image distribution $G(s)$ does match the target image set, but it cannot correspond to the original image s in a meaningful way. This is because there are many mappings G that can generate images with the same distribution. Moreover, this unidirectional training also triggers the well-known problem of mode collapse.

Zhu et al. used the property that translation should be "cycle consistent" to define another mapping $F : T \rightarrow S$ to reproject the translated image onto the original image, ensuring that $F(G(s)) = s$ and $G(F(t)) = t$. The model uses two generators and their corresponding discriminators using adversarial loss and defines a novel cycle consistency loss to train the two generators, and discriminators simultaneously. Figure 1 shows the basic design of CycleGAN and the Loss definition.

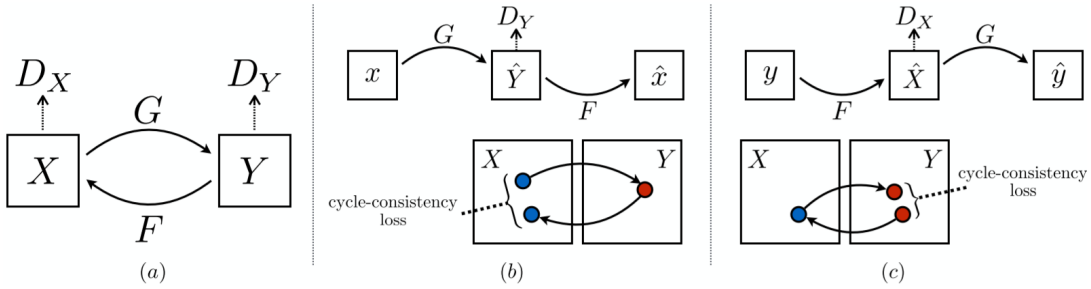

Figure 1: (a) The model contains two mapping functions (generators) and associated adversarial discriminators (b) forward cycle-consistency loss (c) backward cycle-consistency loss.

2.2 Model Architecture

To accomplish I2I, the overall model contains two generators $G : S \rightarrow T$ and $F : T \rightarrow S$ and the corresponding two adversarial discriminators D_S and D_T, which are used to distinguish

between the original image and the translated image. The basic idea is to improve the accuracy of the adversarial discriminators on image sets S or T while improving the image translation ability of the generators. The ideal result is that the discriminators maintain high accuracy on the image sets S or T, but only 50% accuracy in recognizing the fake images translated by the generators.

Adversarial Loss. Adversarial loss is applied to generators and discriminators and is mainly used to ensure that the levels of sets of translations from domain S (T) and domain T (S) are appropriate. Below is the adversarial loss for G and D_T. The other one is similar.

$$\mathscr{L}_{GAN}(G, D_T, S, T) = \mathbb{E}_{t \sim p_{data(t)}}[\log D_T(t)] + \mathbb{E}_{s \sim p_{data(s)}}[\log(1 - D_T(G(s)))] \tag{1}$$

Cycle Consistency Loss. To ensure cycle consistency, a reproject error-like forward cycle consistency loss and backward cycle consistency loss, corresponding to two generators respectively, are used, which are defined as:

$$\mathscr{L}_{cyc}(G, F) = \mathbb{E}_{s \sim p_{data(s)}}[||F(G(s)) - s||_1] + \mathbb{E}_{t \sim p_{data(t)}}[||G(F(t)) - t||_1] \tag{2}$$

Therefore, the full objective function can be defined as:

$$G^*, F^* = \arg\min_{G,F} \max_{D_S, D_T} \mathscr{L}(G, F, D_T, D_S) \tag{3}$$

$$\mathscr{L}(G, F, D_T, D_S) = \mathscr{L}_{GAN}(G, D_T, S, T) + \mathscr{L}_{GAN}(G, D_S, T, S) + \lambda \mathscr{L}_{cyc}(G, F) \tag{4}$$

2.3 Implementation

We have organized the implementation of CycleGAN into four steps. The first step is to define the dataset and pre-processing, the second step is to define the discriminator model, the third step is to define the generator model, and the fourth step is to define the training for CycleGAN. We use Pytorch to implement CycleGAN and refer to the source code of Zhu et al [1] and Aladdin's code reproduction [2]. Here we only present the core codes, please check the source code for detailed implementation.

2.3.1 Dataset & Pre-processing

We use the GTA5 video game street view from [3] as the original dataset S and the real cityscape dataset from [4] as the T dataset, so our goal is to convert the GTA5 video game street view image into a real street view style image. The image augmentation we use is, to resize the image to 180×360, then use horizontal flip on the image with $p = 0.5$, and finally normalize all the image channels to have a mean of 0.5 and standard deviation of 0.5. Finally, convert it to tensor. The detailed implementation is shown below.

```
A.Compose([A.Resize(width=360, height=180),
                    A.HorizontalFlip(p=0.5),
                    A.Normalize(mean=[0.5, 0.5, 0.5],
                                std=[0.5, 0.5, 0.5],
                                max_pixel_value=255),
                    ToTensorV2()])
```

2.3.2 Discriminator Model

The implementation of discriminator architectures follows the paper description of Zhu et al., who use 70 $\times$ 70 PatchGAN as the discriminator model. Define `Ck` as 4 $\times$ 4 Convolution-InstanceNorm-LeakyReLU layer with k filters and stride 2. We use Pytorch's `nn.Module` to define `Ck` block class:

```
self.conv = nn.Sequential(
    nn.Conv2d(
        in_channels,
        out_channels,
        4,
        stride,
        1,
        bias=True,
        padding_mode="reflect"
    ),
    nn.InstanceNorm2d(out_channels),
    nn.LeakyReLU(0.2, inplace=True)
)
```

We then connect four `Ck` blocks in the order of C64-C128-C256-C512 as the discriminator architecture, where the C64 block does not use the InstanceNorm layer. Finally we use a convolution to produce a 1-dimensional output and add the sigmoid function.

2.3.3 Generator Model

The Generator architecture used by Zhu et al. consists of a convolution block and a residual block, where the convolution block is divided into downsampling and upsampling blocks. In the following, we use `nn.Module` define convolution block as:

```
# Downsampling layers or upsampling layers
self.conv = nn.Sequential(
    nn.Conv2d(in_channels,
              out_channels,
              padding_mode="reflect",
              **kwargs)
              if down
              else
              nn.ConvTranspose2d(in_channels,
                                 out_channels,
                                 **kwargs),
    nn.InstanceNorm2d(out_channels),
    nn.ReLU(inplace=True) if use_act else nn.Identity()
)
```

Based on this convolution block class, we can define three types of convolution blocks. The first is the `c7s1-k` block, which uses a 7 $\times$ 7 Convolution-InstanceNorm-ReLU layer with k filters and stride 1. The second is the downsampling block `dk`, which uses a 3 $\times$ 3 Convolution-InstanceNorm-ReLU layer with k filters and stride 2. The third is the upsampling block `uk`, which is a 3 $\times$ 3 fractional-strided-Convolution-InstanceNorm-ReLU layer with k filters and stride $\frac{1}{2}$.

For residual block, it is two 3×3 convolutional layers with the same number of filters on both layer. We defined it as the residual block (below) class using `nn.Module`. The `forward` function is `return x +self.block(x)`.

```
self.block = nn.Sequential(
    ConvBlock(channels,
              channels,
              kernel_size=3,
              padding=1),
    ConvBlock(channels,
              channels,
              use_act=False,
              kernel_size=3,
              padding=1)
)
```

We build the generator network as: "c7s1-64,d128,d256,R256,R256,R256,R256,R256,R256, R256, R256, R256, R256, R256, R256, R256, R256, u128 u64,c7s1-3", and followed by a tanh activation function.

```
# c7s1-64
x = self.initial(x)
# d128,d256
for layer in self.down_blocks:
    x = layer(x)
# R256,R256,R256,R256,R256,R256,R256,R256,R256
x = self.res_blocks(x)
# u128 u64
for layer in self.up_blocks:
    x = layer(x)
# c7s1-3
return torch.tanh(self.last(x))
```

2.3.4 Training

When implementing training, we adopted Zhu et al.'s training details and their recommended settings, such as training parameter settings:

- Use a batch size of 2 to prevent out of memory.
- A total of 200 epochs are trained, of which the learning rate is kept constant for 100 epochs, and the decay learning rate strategy is used for the other 100 epochs. Here we use the cosine annealing learning rate strategy.
- Set the initial learning rate to 0.0002
- Set cycle consistent control factor $\lambda = 10$
- Use Adam solver

At the same time, we adopt their recommended improvement techniques, such as using least-squares loss to replace the negative log likelihood objective in $\mathscr{L}_{GAN}$. This loss is more stable during training and generates higher quality results. Therefore, the new goal will be:

- When training G or F: minimize

$$\mathbb{E}_{s\sim p_{data(s)}}[(D_T(G(s))-1)^2]+$$

$$\mathbb{E}_{t\sim p_{data(t)}}[(D_S(F(t))-1)^2]+$$

$$\mathscr{L}_{cyc}(G,F)$$

- When training D_S or D_T: minimise

$$\mathbb{E}_{s\sim p_{data(s)}}[(D_S(s)-1)^2]+\mathbb{E}_{t\sim p_{data(t)}}[D_S(F(t))^2]+$$

$$\mathbb{E}_{t\sim p_{data(t)}}[(D_T(t)-1)^2]+\mathbb{E}_{s\sim p_{data(s)}}[D_T(G(s))^2]$$

Following the above training setting, for each batch images (source, target), we first train the discriminators D_T and D_S. We can obtain two discriminators' result by using corresponding generators.

```
# Patch: DT(T) and DT(GT(S))
DT_real = disc_T(target)
DT_fake = disc_T(gen_T(source))

# Patch: DS(S) and DS(GS(T))
DS_real = disc_S(source)
DS_fake = disc_S(gen_S(target))
```

And then we can calculate the $\mathscr{L}_{GAN}$ losses with respect to discriminators D_T and D_S.

```
# L_GAN(GT,DT,S,T) = minimize Et[(DT(t) - 1)^2] + Es[DT(GT(s))^2]
DT_real_loss = mse(DT_real, torch.ones_like(DT_real))
DT_fake_loss = mse(DT_fake, torch.zeros_like(DT_fake))
# L_GAN(GT,DT,S,T)
DT_loss = DT_real_loss + DT_fake_loss

# L_GAN(GS,DS,S,T) = minimize Es[(DS(s) - 1)^2] + Et[DS(GS(t))^2]
DS_real_loss = mse(DS_real, torch.ones_like(DS_real))
DS_fake_loss = mse(DS_fake, torch.zeros_like(DS_fake))
# L_GAN(GS,DS,S,T)
DS_loss = DS_real_loss + DS_fake_loss
```

Finally, we put it together to get the total adversarial loss, and then train the model with the defined optimizer to opitmize discriminators.

```
# put it togethor: L_GAN(GT,DT,S,T) + L_GAN(GS,DS,S,T)
D_loss = (DT_loss + DS_loss) / 2
```

Next, we need to train the generators G_t and G_S. In the full objective function, the two generator models appear both in adversarial loss and cycle consistency loss. To compute the adversarial loss in terms of generators, the results of the discriminators are first computed for the forward and backward translated images.

```
# Patch: DT(GT(S)) and DS(GS(T))
DT_fake = disc_T(gen_T(source))
DS_fake = disc_S(gen_S(target))
```

Then we need to compute the adversarial loss with respect to generators.

```
# minimize Et[(DT(t) - 1)^2] and Es[(DS(s) - 1)^2]
loss_GT = mse(DT_fake, torch.ones_like(DT_fake))
loss_GS = mse(DS_fake, torch.ones_like(DS_fake))
```

To compute the cycle consistency loss, which only concludes the generators, we first need to get the reprojection of the translated images.

```
# GS(GT(S)) -> S'
cycle_source = gen_S(gen_T(source))
# GT(GS(T)) -> T'
cycle_target = gen_T(gen_S(target))
```

And then calculate the cycle consistency loss according to the definition shown above. Finally, minimising the cycle consistency loss by using the defined optimizer to opitmize the generators model.

```
# Lcyc(GS,GT) = Es[|GS(GT(s)) - s|] + Et[|GT(GS(t)) - t|]
cycle_source_loss = l1(source, cycle_source)
cycle_target_loss = l1(target, cycle_target)

# add all togethor: L(GS,GT,DS,DT) = LGAN(GT,DT,X,Y) + LGAN(GS,DS,X,Y) +
                                     Lcyc(GS,GT)
G_loss = loss_GT + loss_GS +
         lambda * (cycle_source_loss + cycle_target_loss)
```

2.4 Result Discussion

pro cons

3 Unsupervised Domain Adaptation via I2I Translation

References

[1] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired image-to-image translation using cycle-consistent adversarial networks," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2223–2232.

[2] A. Persson, "A clean, simple and readable implementation of cyclegan in pytorch," https://github.com/aladdinpersson/Machine-Learning-Collection/tree/master/ML/Pytorch/GANs/CycleGAN, accessed: 2023-11-10.

[3] S. R. Richter, V. Vineet, S. Roth, and V. Koltun, "Playing for data: Ground truth from computer games," 2016.

[4] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The cityscapes dataset for semantic urban scene understanding," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 3213–3223.