Cycle-Adversarial Networks for Image-to-Image Translation and Feature Extraction

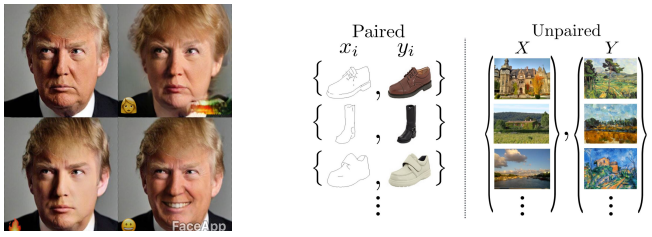
Srinidhi Goud
Department of Computer Science
New York University
sgm400@nyu.edu

Abstract

Transferring characteristics of images from one domain to another requires us to learn function mapping that maps an input image from one domain to another. This is made possible by training a model with paired images creating a one-to-one mapping. But this project relaxes this requirement of one-to-one mapping, greatly increasing the trainable data set. Our goal is to learn a mapping G: X $\rightarrow$ Y where, G(X) will form a domain that is a sub-part of the domain Y and will be in-distinguishable. Moreover, this project introduces a cycle consistency by creating an inverse mapping such that F(G(X)) $\approx$ X. This project further explored feature visualization, effects of training with smaller data set and drawing some light on why this model fails on few image domains

1 Introduction

Just the prospect of transferring an image from one domain to another is very exciting. For example, faceapp on iphoneX uses it to chnage the expression of your face. We can convert any image to a domain of style of Van Gogh starry night. More interesting, results are to be able to transfer one set of objects to another domain such as mapping from male to female or horse to zebra or oranges to apples.



(a) (b)

Figure 1: a) Style transfer in Faceapp, b)Paired images required for training Style transfer networks

Traditionally, we need a paired data set to train such a model to transfer images of one domain to another through paired images only. That is, we need to train the model with an image in domain A mapping to an image in domain B. But doing this restricts us highly to a limited data sets. Or sometimes, creating such specific data sets might not be possible. This paper by the authors, achieves a relaxation that makes such transfer possible without paired image training. That is, by reasoning the stylistic differences between two images domains, we can transfer one image's style to the another domain, predicting how it would be. We are given one set of images in domain X and a different set in

domain Y . We may train a mapping G : X→Y such that the output $\hat{y}$= G(x), x ∈X, is indistinguishable from images y∈Y by an adversary trained to classify $\hat{y}$ apart from y.

Quoting the authors of the paper, "Adversarial Training can, in theory, learn mappings G and F that produce outputs identically distributed as target domains Y and X respectively. However, with large enough capacity, a network can map the same set of input images to any random permutation of images in the target domain, where any of the learned mappings can induce an output distribution that matches the target distribution. Thus, an adversarial loss alone cannot guarantee that the learned function can map an individual input xi to a desired output yi"

To regularize the model, the authors introduce the constraint of cycle-consistency - if we transform from source distribution to target and then back again to source distribution, we should get samples from our source distribution.

2 Objective

To optimize using following losses:

2.1 Adversarial Loss

$$L_{GAN}(G, D_Y, X, Y) = E_{y \sim p_{data}}[logD_Y(y)] + E_{x \sim p_{data}}[log(1 - D_Y(G(x)))] \tag{1}$$

2.2 Cycle-Consistency Loss

$$L_{CYC}(G, F) = E_{y \sim p_{data}}[\|G(F(y)) - y\|] + E_{x \sim p_{data}}[\|F(G(x)) - x\|] \tag{2}$$

2.3 Full Objective

$$L(G, F, D_X, D_Y) = L_{GAN}(G, D_X, X, Y) + L_{GAN}(F, D_X, Y, X) + \lambda L_{CYC}(G, F) \tag{3}$$

where λ controls the weights of the two objectives. Finally, the project aims to solve:

$$G^*, F^* = arg \min_{G,F} \max(D_x, D_y) L(G, F, D_X, D_Y) \tag{4}$$

3 Network Architecture

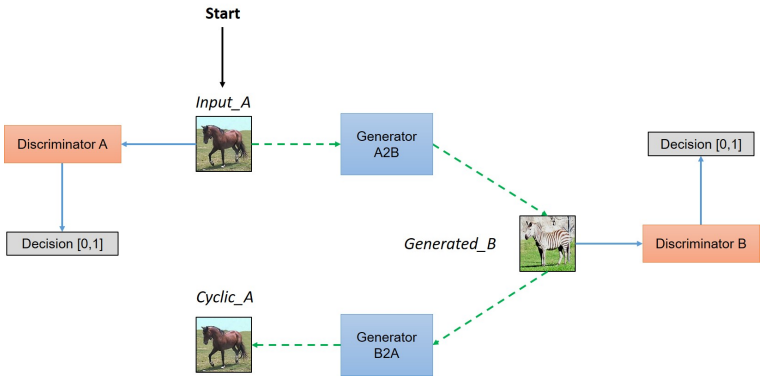


Figure 2: complete network blocks

We feed testA data set as realA to Generator netGAtoB to map images from domain D_A to domain D_B. The generated image is now called fakeB data that is fed to the discriminator netDB that Discriminates the fakeA dat from a realB data. Now the fakeB data is fed to another Generator netGBtoA that create fakeA data. This data is now fed to the Discriminator netDA that discriminates this data from the original data realA. This forms a whole cycle, in the adversarial network.

4 Approach and results

Note: In the zip file all the models are not there but have some screen shots of the models that are saved in the checkpoints folder. All the approaches have the same model as shown here in this figure:

```
ResnetGenerator (
  (model): Sequential (
    (0): ReflectionPad2d (3, 3, 3, 3)
    (1): Conv2d(3, 64, kernel_size=(7, 7), stride=(1, 1))
    (2): InstanceNorm2d(64, eps=1e-05, momentum=0.1, affine=False)
    (3): ReLU (inplace)
    (4): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (5): InstanceNorm2d(128, eps=1e-05, momentum=0.1, affine=False)
    (6): ReLU (inplace)
    (7): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (8): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False)
    (9): ReLU (inplace)
    (10): ResnetBlock X 9 such blocks (
      (conv_block): Sequential (
        (0): ReflectionPad2d (1, 1, 1, 1)
        (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False)
        (3): ReLU (inplace)
        (4): ReflectionPad2d (1, 1, 1, 1)
        (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False)
      )       )
    (19): ConvTranspose2d(256, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
output_padding=(1, 1))
    (20): InstanceNorm2d(128, eps=1e-05, momentum=0.1, affine=False)
    (21): ReLU (inplace)
    (22): ConvTranspose2d(128, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output_pa
1))
    (23): InstanceNorm2d(64, eps=1e-05, momentum=0.1, affine=False)
    (24): ReLU (inplace)
    (25): ReflectionPad2d (3, 3, 3, 3)
    (26): Conv2d(64, 3, kernel_size=(7, 7), stride=(1, 1))
    (27): Tanh ()
  )
)
```

Figure 3: Layers in Generators

```
NLayerDiscriminator (
  (model): Sequential (
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (1): LeakyReLU (0.2, inplace)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (3): InstanceNorm2d(128, eps=1e-05, momentum=0.1, affine=False)
    (4): LeakyReLU (0.2, inplace)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False)
    (7): LeakyReLU (0.2, inplace)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(1, 1), padding=(1, 1))
    (9): InstanceNorm2d(512, eps=1e-05, momentum=0.1, affine=False)
    (10): LeakyReLU (0.2, inplace)
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), padding=(1, 1))
  )
)
```

Figure 4: Layers in Discriminator

4.1 Lua implementation

(a) a (b) b

Figure 5: Real image A and its transformation for two different images

Additionally, in the beginning the authors had the notion that lua implementation, on which the project paper's results are based on is significantly performing better than pyTorch model. This the generated image where the generator is trying to generate zebras out of horses. The bottom show the best results out of both pre-trained and trained lua model.

As we can see in fig 5 a) is transformed image while b) shows negative drawback of adversarial networks not being able to distinguish multiple objects. That leads to image distortion. Here is the result of orange to apple model fig 6.

Figure 6: Apples to Orange

4.2 PyTorch implementation

Note: real-A is input image in domain D_A fake-A is output image in domain D_A generated by netG-B-to-A for real-B input real-B is input image in domain D_B fake-B is output image in domain D_B generated by netG-A-to-B for real-A input rec-A is output image of generator B-to-A in domain D_A for input fake-B rec-B is output image of generator A-to-B in domain D_B for input fake-A

To avoid mode collapse, we pool the real A and real B to a batch size of 50 images while training.

However, I have felt that pyTorch gave far better results for same sized data and epoch counts. Partly beacuse, the data set used for pyTorch had higher resolution and unique images, than lua's case. This is limited to horse2zebra data set. Whereas, I could not test for other cases such as orange to apples due lack of time/resources. However, I have trained the model for maps and horse-to-zebra generation.

This is the result for 200 epochs, and full data size of around 1334 images:

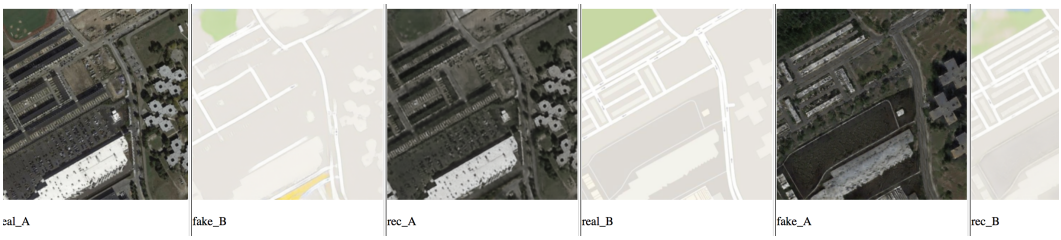


Figure 7: Results for maps

Figure 8: Results for Horse to Zebra

This is the result with half the data size as input:

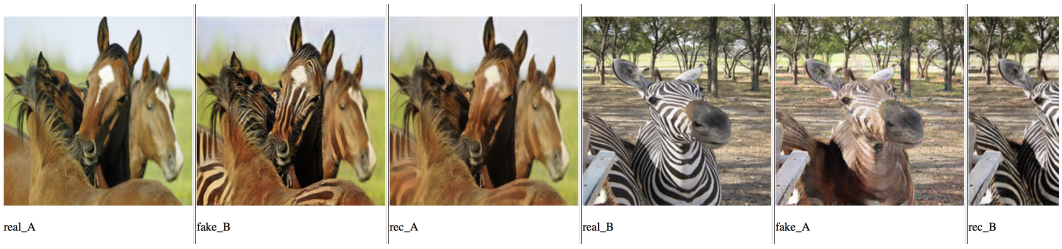


Figure 9: Results for Horse to Zebra with half the size of data, training

This result, took just 667 images to train for 200 epochs and still did a brilliant job to transform the input image. I did not see any significant differences with full data size or half data size. Now the results for the image transformation at few epoch for horse to zebra transformation:

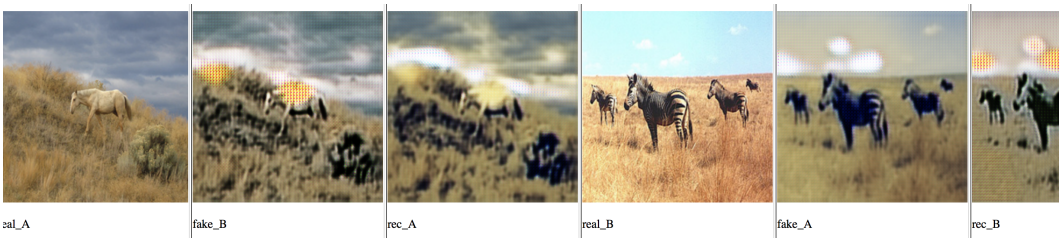


Figure 10: Results after epoch 1. We are just getting some image distortion after epoch 1

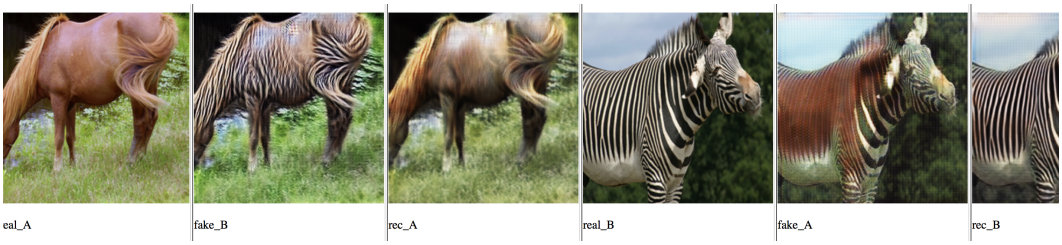


Figure 11: Results after epoch 24, we first begin to notice some transformation into domain B

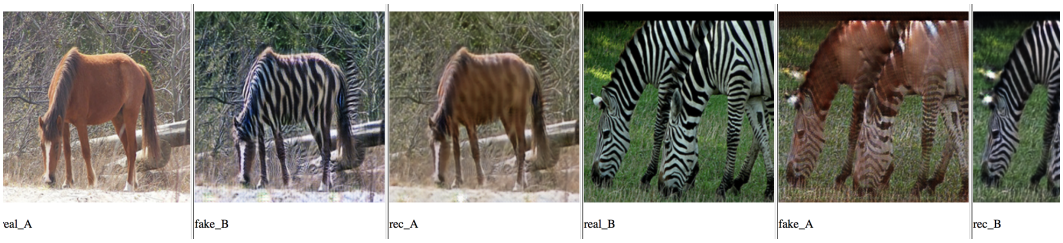


Figure 12: Results after epoch 78, we start getting some satisfiable result

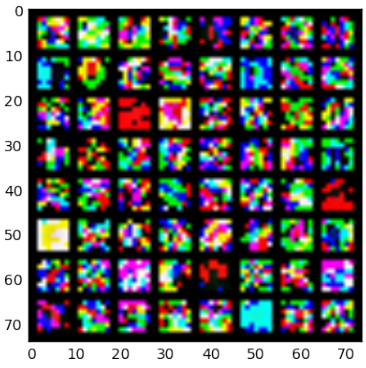
With this, we will get a general idea, on how soon our model is converging. If you look at epoch-wise transformation, we get a zebra at an epoch count as low as 25, but then, when the model sees a new type of image, it fails to transform it. This is similar to mode collapse, and Generator converging to a small point in Domain B. I think this leaves us at a lower threshold on the least number of epochs that are definitely required for such models, to create a good mapping. And also, it is necessary that input data is as diverse as possible. The slurm files, also show the time stamps, giving us an idea on how soon this training is executed for every epoch. Also, the log files are also added in the zip file. As my project doesnot really relate to trying to decreasing the Discriminator losses I did not add the loss table. But if there is a way other than batch normalization, pooling and others, it is important to work to make the discriminator as tighter upper bound for generator as possible. Interesting way would be to train Discriminator more number of times than Generator, by maintaining a deeper model for Generator.

But since, our model is fed only horse images as input, we can say, that, in general, if we feed multiple objects it cannot map the image properly and we have seen this in the above results fig 5 b. This model might not be good if we try to change the shape of the object interpreting from fig 5 b.. This necessitates us to study how the features are mapped, so that we can see what is really important for the model to learn a right mapping. Even though this might be subjective to the domain of similar objects (horse and zebra physically have same structure) this might be a worthy attempt to understand the models.

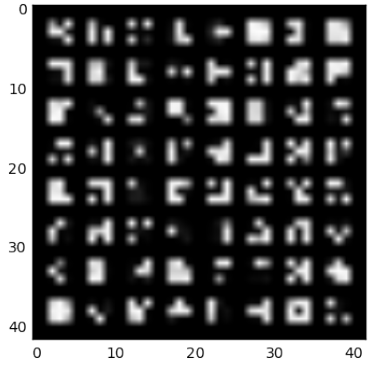
4.2.1 Feature Extraction

The feature extraction on the present project had been extremely difficult. Most of my time had been spent on extracting them as layers were not named and iterating them was difficult in pyTorch due to lack on prior of information. However, a simple feature extraction did not provide any significantly conceivable results. Here, I will try to explain my approach.

Unlike the popular way of projecting the output from higher layers to lower layers, I have simple feed-forwarded the input and extracting the output at that particular layer. Additionally, there was no optimization applied to find the right inputs to activate the neurons of interest in the network. Otherwise, I have also projected the weight vectors at various layers to see if there is any layer in generator model, that tries to classify the image based on structure.

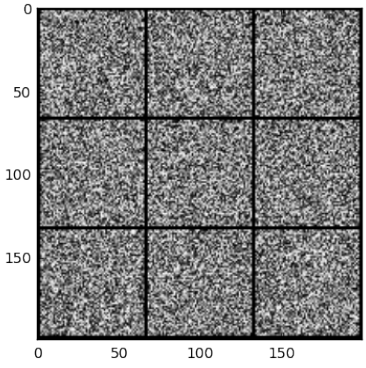


(a) Weights of last conv layer projected from 64 layers to 3 layers of images

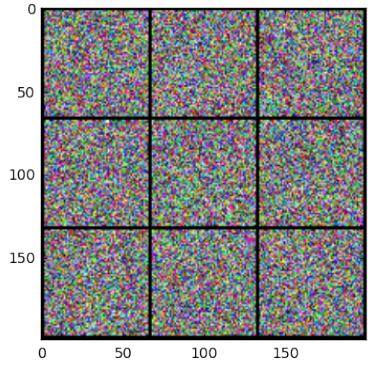


(b) Weights of last conv layer projected from 1 input to 64 layers of images

Figure 13

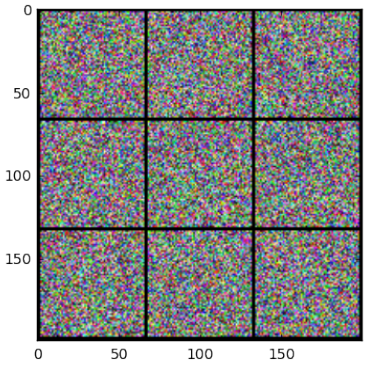


(a) First 9 layers of weights of a Discriminator are projected

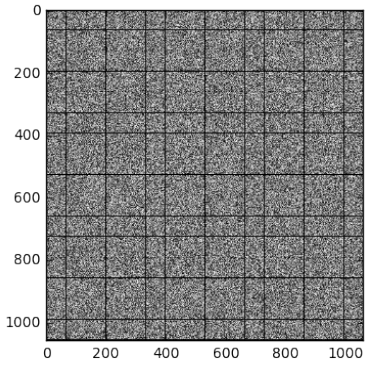


(b) Color image is fed with each of R, G and B layers individually, concatenated and projected. When one color is fed, the rest of the colours are zero'd. Finally all three outputs are concatenated to form 3 values for RGB colored output. Output at 3rd Conv layer from beginning.

Figure 14



(a) Color image is fed with each of R, G and B layers individually, concatenated and projected. When one color is fed, the rest of the colours are cloned to the same value. Repeated for all three colors respectively and concatenated to form a colored output. Output at 3rd Conv layer from beginning.



(b) Output at 3rd Conv layer when the entire RGB is fed. Consequently a black/white image is projected.

Figure 15

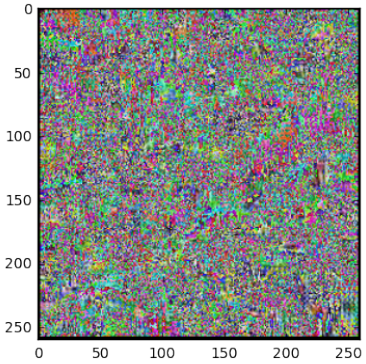


Figure 16: Output at the last conv layer

All of these layers are not normalized before they are projected. The better way to extract features is to selectively project input images, to collect the activations of the neurons at various layers. This will give us a better idea what each layer will do. Ofcourse, this is very domain specific, so for every domain D_A to D_B transformation will require us to extract features manually.

Acknowledgments

This project has been possible with help of the project built in GitHub by Jun-Yan Zhu and Taesung Park for both LUA and PyTorch models. I would like to thank them.

References

[1] Ian J. Goodfellow, Jonathon Shlens & Christian Szegedy *Explaining and Harnessing Adversarial Examples* ICLR 2015

[2] Jun-Yan Zhu, Taesung Park, Phillip Isola & Alexei A. Efros *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*ICCV 2017

[3] Matt Zeiler & Rob Fergus *JVisualizing and Understanding Convolutional Networks* ECCV 2014