I AM SOMETHING OF A PAINTER MYSELF

INDIVIDUAL REPORT

Author: Sanat Lal

DATS6203: Machine Learning II

The George Washington University

December 7, 2020

Table of Contents

INTRODUCTION

I'm Something of a Painter Myself Image-to-image translation has been an increasingly popular topic over the last years. One Sample of such a task is art style transfer. Style transfer algorithms in the context of art try to capture the general style of an artist or an image and apply it to one or many content pictures. As an example, think of the latest holiday picture and try to imagine how Monet or Van Gogh would have painted the scene. So can Data science, in the form of GANs, trick classifiers into believing that the image created is a true Monet

DATASET DETAIL

The dataset contains four directories: monet_tfrec, photo_tfrec, monet_jpg, and photo_jpg. The monet_tfrec and monet_jpg directories contain the same painting images, and the photo_tfrec and photo_jpg directories contain the same photos.

The Monet directories contain Monet paintings.

The photo directories contain photos

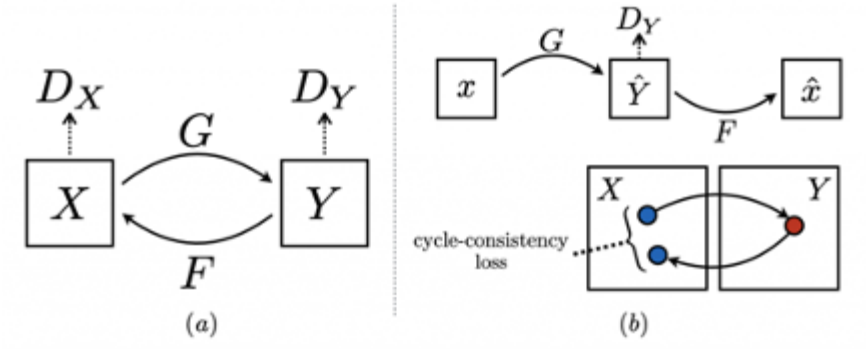
- Monet - 300 Monet paintings sized 256x256 in JPEG format
- monet_tfrec - 300 Monet paintings sized 256x256 in TFRecord format

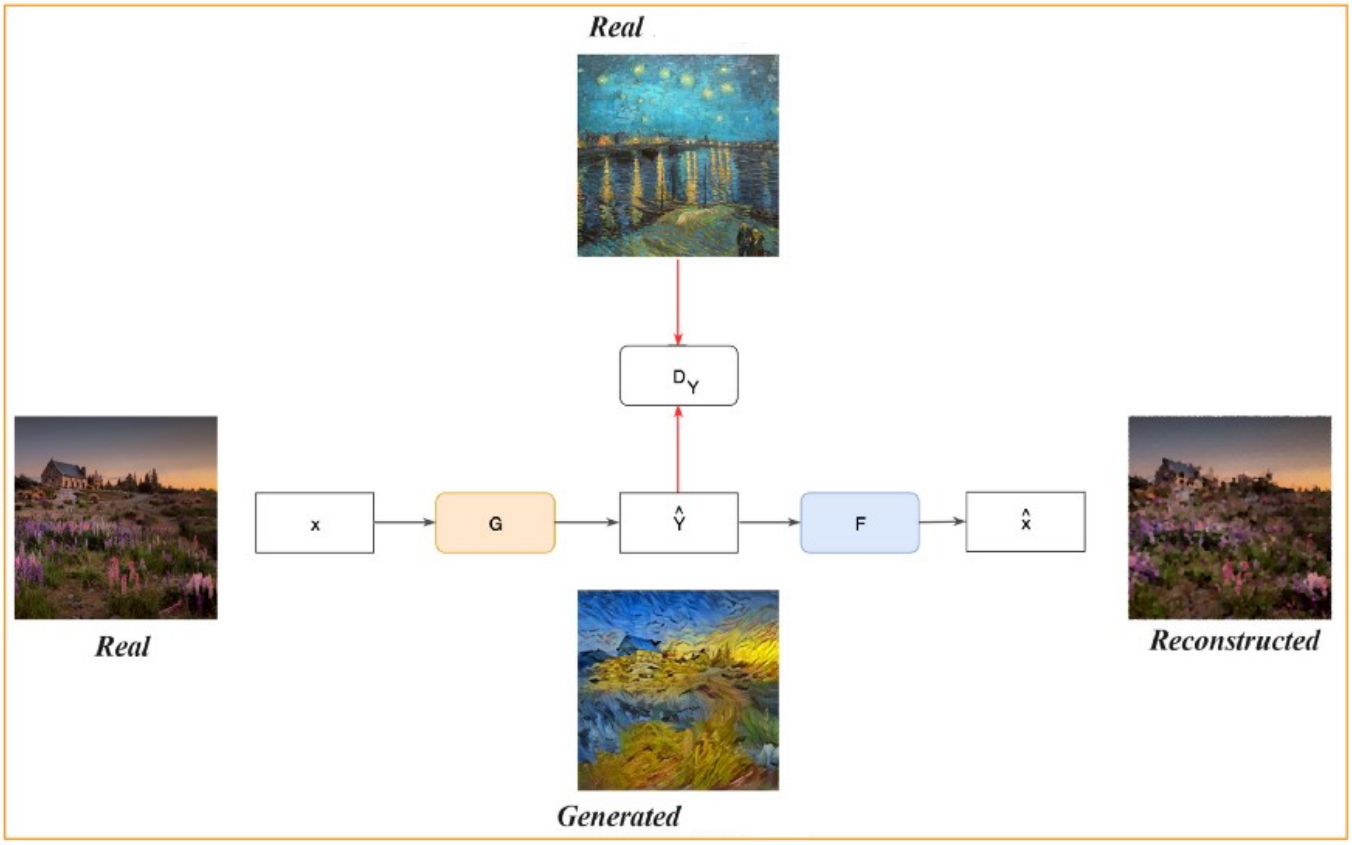
- photo_jpg - 7028 photos sized 256x256 in JPEG format
- photo_tfrec - 7028 photos sized 256x256 in TFRecord format

NETWORK ARCHITECTURE

- A generative adversarial network(GAN) is a type of machine learning architecture to generate new photographs which have realistic characteristics. It consists of two models: a generator model and a discriminator model. The generator takes a point from a latent space as input and generates new plausible images from the domain and the discriminator takes an image as input and predicts whether it is real or fake. Both models are trained and expected to generate more realistic images.
-
- The CycleGAN is an extension of the GAN architecture that contains two generators and two discriminators. In this case, we can summarize the generator and discriminator models as follows:
- **Generator G: Convert real photos to Monet style paintings**
- **Discriminator DX: Identify real Monet paints or fake Monet paintings generated by G**

- Generator F: Convert real Monet paintings to photos
- Discriminator DY: Identify real photos or fake photos generated by **F**





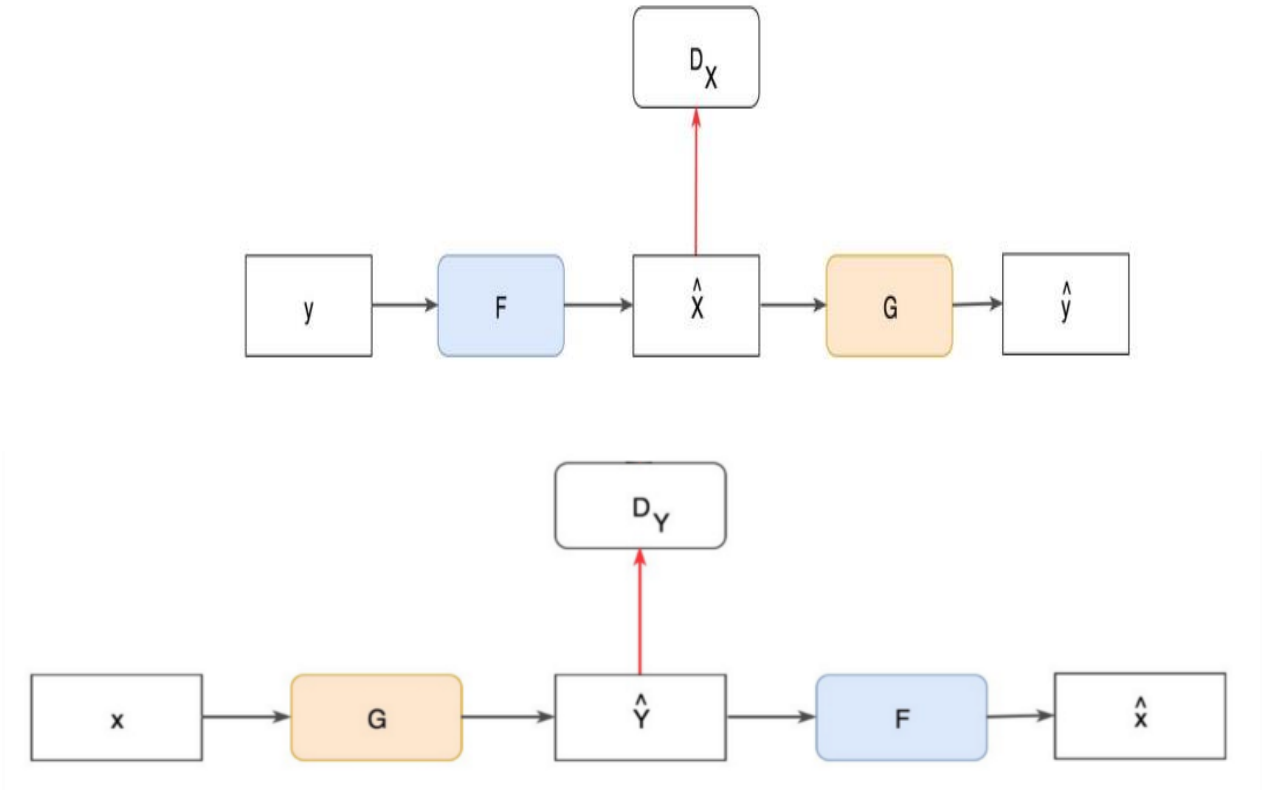


Figure 1. Network architecture

The loss of CycleGAN contains adversarial loss and cycle consistency loss.

Adversarial loss is a standard GAN loss. In this case we have 2 adversarial loss:

Cycle consistency loss compares an input photo to the Cycle GAN to the generated photo and calculates the difference between the two:

The whole loss is: Adversarial Loss & Cyclic Loss

DESCRIPTION OF INDIVIDUAL WORK

DATA PREPROCESSING

- As images were already in 3* 256*256 format so the image was resized to 128* 128 , furthermore it was scaled as well as transposed . These methods were applied to both sets of images i.e. Monet & photos. The images were resized and standardized in the training set while retaining color for the images. This process only cuts out the noise in the outer part of the painted image. Moreover, borders were also added to the image using ANTIALIAS

```
def training(monet_addr, photos_addr):
    X_train, Y_train = np.zeros((300, 3, 256, 256), dtype=np.float32), np.zeros((7038, 3, 256, 256), dtype=np.float32)

    for i in range(len(monet_addr)):
        temp_np = np.asarray(
            Image.open(monet_addr[i]).resize((256, 256), Image.ANTIALIAS))  # resizing the image to 128x128
        X_train[i] = temp_np.transpose(2, 0, 1)
        X_train[i] /= 255
        X_train[i] = X_train[i] * 2 - 1

    for i in range(len(photos_addr)):
        temp_np = np.asarray(Image.open(photos_addr[i]).resize((256, 256), Image.ANTIALIAS))
        Y_train[i] = temp_np.transpose(2, 0, 1)
        Y_train[i] /= 255
        Y_train[i] = Y_train[i] * 2 - 1

    return X_train, Y_train
```

DISCRIMINATOR NON-PATCH

- The flow of algorithm implementation in building the network layers for training can be seen in figure 2. For our discriminator we have 7 convolutional 2d layers

Layer	Output Neuron	Kernel size	Stride	Batch Normal
Conv2d	64	4	2	FALSE
Conv2d	128	4	2	TRUE
Conv2d	256	4	2	TRUE
Conv2d	512	4	2	TRUE
Conv2d	512	4	2	TRUE
Conv2d	512	4	2	TRUE
Conv2d	512	2	2	FALSE
Linear	1	NA	NA	NA

Discriminator Non-Patch Layer

For Forward Propagation, we have used Leaky relu as it was giving us better results as compared to ReLU. If the input of a neuron with ReLU activation goes below zero for all combinations somehow (e.g. because it developed a large enough negative bias or by other, more unlikely, accidents), its gradient becomes zero forever and it will never become alive again. Leaky ReLU always has a bit of gradient left, so it can shift back to life over time.

```python
class disc_np(nn.Module):
    def __init__(self):
        super(disc_np, self).__init__()

        self.conv1 = nn.Conv2d(3, 64, kernel_size=4, stride=2)

        self.conv2 = nn.Conv2d(64, 128, kernel_size=4, stride=2)
        self.bn2 = nn.BatchNorm2d(128)

        self.conv3 = nn.Conv2d(128, 256, kernel_size=4, stride=2)
        self.bn3 = nn.BatchNorm2d(256)

        self.conv4 = nn.Conv2d(256, 512, kernel_size=4, stride=2)
        self.bn4 = nn.BatchNorm2d(512)

        self.conv5 = nn.Conv2d(512, 512, kernel_size=4, stride=2)
        self.bn5 = nn.BatchNorm2d(512)

        self.conv6 = nn.Conv2d(512, 512, kernel_size=4, stride=2)
        self.bn6 = nn.BatchNorm2d(512)

        self.conv7 = nn.Conv2d(512, 512, kernel_size=2, stride=2)

        self.head = nn.Linear(512, 1)

    def forward(self, input):
        x = Fn.leaky_relu(self.conv1(input), negative_slope=0.2)
        x = Fn.leaky_relu(self.bn2(self.conv2(x)), negative_slope=0.2)
        x = Fn.leaky_relu(self.bn3(self.conv3(x)), negative_slope=0.2)
        x = Fn.leaky_relu(self.bn4(self.conv4(x)), negative_slope=0.2)
        x = Fn.leaky_relu(self.bn5(self.conv5(x)), negative_slope=0.2)
        x = Fn.leaky_relu(self.bn6(self.conv6(x)), negative_slope=0.2)
        x = Fn.leaky_relu(self.conv7(x), negative_slope=0.2)

        x = x.view(x.size(0), -1)
        x = self.head(x)

        return torch.sigmoid(x)
```

MODEL COMPILATION

- For training purpose, we use epochs of 120 and Batch size 16. We first trained our real photos and tried to convert it into Monet style paintings . secondly, we will use discriminator to discriminate the real images as real and fake images as fake.
- Similarly, we will use the input original Monet paintings and will try to generate real scenic pictures using the same cyclic GANs. After each cycle loss will be calculated and will update the Weights and bias in the form of back propagation.
- With every epoch we will keep on getting our features and our task is not to let go the actual aesthetics of original image. The errors form cyclic loss and adversarial loss will be added together to form total loss of the model
- I started with 4 epochs and 40 batch size but while using 40 batch size we started getting cuda error.
- Error which we removed it by reducing batch size. Also, in low epochs we were getting blank images and after increasing epochs we get better results

```
k = 0

for epoch in range(epochs):
    print('Epoch number: {0}'.format(epoch))

    for batch in range(X_torch.size(0) // batch_size):
        if batch % 100 == 0:
            print('*Batch number: {0}*'.format(batch))

        monet_real = X_torch[batch * batch_size: (batch + 1) * batch_size]
        if k != 7038:
            photo_real = Y_torch[k % 7038: (k + 1) % 7038]
        else:...
        k += 1

        monet_real = Variable(monet_real).type(dtype)
        photo_real = Variable(photo_real).type(dtype)


        photo_fake = G(monet_real)

        scores_real = ptd(Dg, photo_real)
        scores_real_np = Dgnp(photo_real)
        scores_fake = ptd(Dg, photo_fake)
        scores_fake_np = Dgnp(photo_fake)

        label_fake = Variable(torch.zeros(batch_size)).type(dtype)
        label_real = Variable(torch.ones(batch_size)).type(dtype)
```

CONCLUSION

- Very High Batch size may lead to may troubles if processing capacity is low, CUDA Error is one of them
- If we take low epochs and try to train our model , we might end up losing many features and very high epoch will perform better but it will take lot of time .In our case very high epoch, lead to a better loss but not in a better quality of the output images
- Resizing the image to 256 * 256 may not lead to best results we need to preprocess(normalization) after every convolution

REFERENCES

- https://hardikbansal.github.io/CycleGANBlog/
- https://poloclub.github.io/ganlab/
- https://machinelearningmastery.com/tour-of-generative-adversarial-network-models/

- https://www.youtube.com/watch?v=5RYETbFFQ7s
- https://machinelearningmastery.com/what-is-cyclegan/
- https://www.kaggle.com/c/gan-getting-started/notebooks
- www.stackoverflow.com
- https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html