I AM SOMETHING OF A PAINTER MYSELF

Final Group Report

Group 3

Reported by

Kaiqi Yu

Vishal Pathak

Zichu Chen

Sanat Lal

December 7, 2020

1. Introduction

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? I'm something of a painter of myself is a competition on Kaggle which requires to generates 7,000 to 10,000 Monet-style images from real photos using GAN. In this project, we trained our CycleGAN model with 300 Monet-style paintings and generated 7038 Monet-style images using given photos.

2. Dataset Description

The dataset from Kaggle 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 300 Monet paintings sized 256x256, which can be used to train the model. The photo directories contain 7028 photos sized 256x256. Monet-style is required to be added to these images.

3. Algorithm Description

A generative adversarial network(GAN) is a type of machine learning architecture to generate new photographs which have realistic characteristics. It is comprised 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.

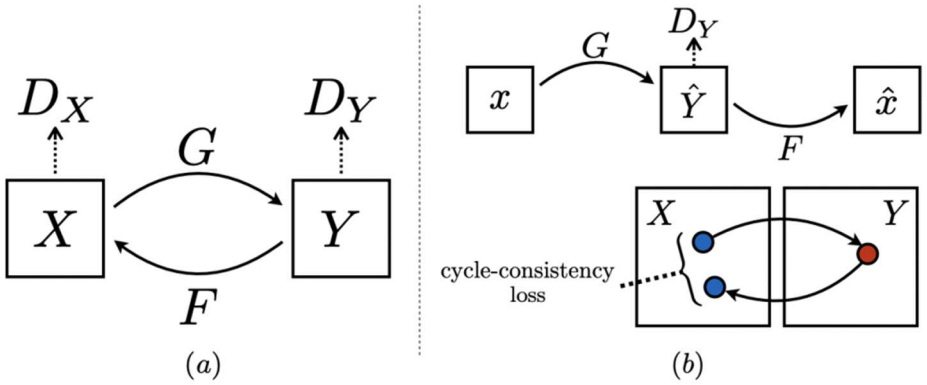
The CycleGAN is an extension of the GAN architecture that contains two generators and two discriminators. In this case, we can summarize the generator and discriminator models as follows:

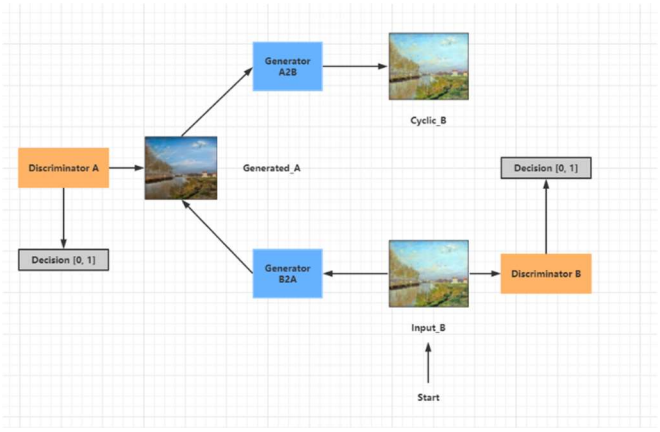
Generator G: Takes real photos as input and generate fake Monet paintings as output.

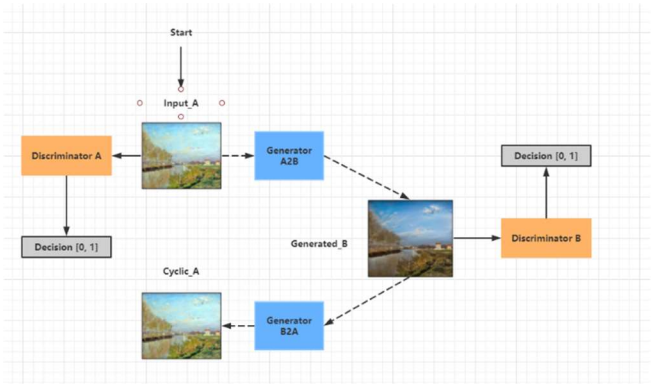
Discriminator DX: Takes real Monet paints and fake Monet paintings generated by G1 as input and output the judgment.

Generator F: Takes real Monet photos as input and generate fake photos as output.

Discriminator DY: Takes real photos and fake photos generated by G2 as input and output the judgement.







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:

$$L(G_X, D_Y, X, Y) = E_{y \sim p(y)}[logD_Y(y)] + E_{x \sim p(x)}[1 - logD_Y(G_X(x))]$$

$$L(G_Y, D_X, Y, X) = E_{x \sim p(x)}[logD_X(x)] + E_{y \sim p(y)}[1 - logD_X(G_Y(y))]$$

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

$$L(G_X, G_Y) = E_{x\sim p(x)}[||G_Y(G_X) - x||_1] + E_{y\sim p(y)}[||G_X(G_Y) - y||_1]$$

The whole loss is:

$$L(G_X, G_Y, D_X, D_Y) = L(G_X, D_Y, X, Y) + L(G_Y, D_X, Y, X) + \lambda L(G_X, G_Y)$$

4. Experimental Setup

4.1 Data Preprocessing

For the Pytorch model, as images were already in 3* 256*256 format, all 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 images.

For the Tensorflow model, all the monet paintings and photos were kept 256*256 shape as they were. A data augmentation function with rotation, flips and transposing was written to multiply augment the dataset. We practiced the original data size, which is 300 and twice the size, which is 600. Normalization was also performed before training.

4.2 PyTorch Model

```
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
```

4.1.1 Discriminator Non-Patch Layer

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	OutputNeuron	Kernel size	Stride	Normalization
Conv2d	64	4	2	/

Conv2d	128	4	2	BatchNormal
Conv2d	256	4	2	
Conv2d	512	4	2	
Conv2d	512	4	2	
Conv2d	512	4	2	
Conv2d	512	2	2	/
Linear	1	/	/	/

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.

```
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)
```

4.1.2 Discriminator

For Discriminator we have used 5 convolution layers . The Inner layers are followed by batch normalization. The Exit layer is a linear layer with 1 output neuron. In the forward function, we have used the same Leaky ReLU activation function.

Layer	Output Neuron	Kernel size	Stride	Normalization
Conv2d	64	4	2	/
Conv2d	128	4	2	Batch Normalization
Conv2d	256	4	2	
Conv2d	512	4	2	
Conv2d	512	4	2	/
Linear	1	/	/	/

```
class disc(nn.Module):
    def __init__(self):
        super(disc, 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=2, stride=1)

        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.conv5(x), negative_slope=0.2)

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

        return torch.sigmoid(x)
```

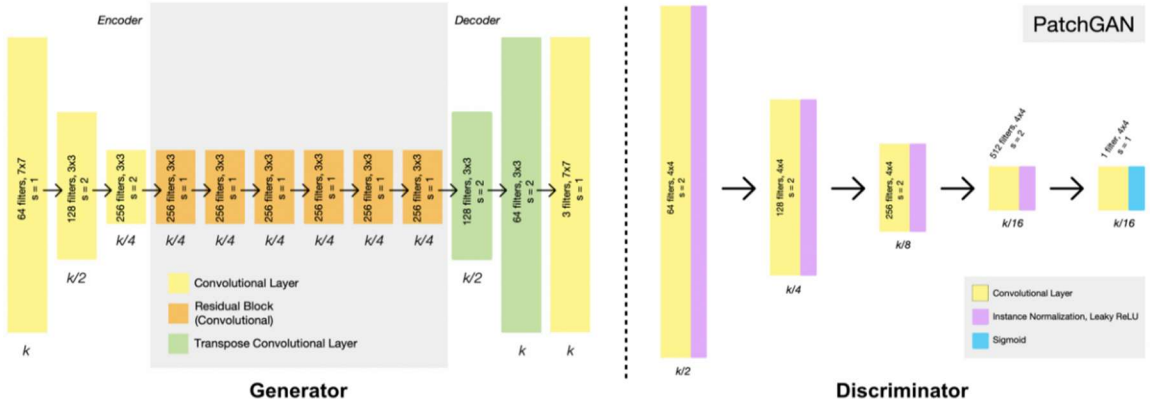
4.1.3 Generator

For the Generator, 18 layers were added in which most of the layers are 2D convolutional. In the last of the middle layers we have used the 2D-Transposed Convolutional layer to generate an output feature map which has a spatial dimension greater than that of the input. In most of our layers we also added Reflection pad to add boundaries of certain length. This is useful as it ensures that the output will transition smoothly into the padding.

Reflection PAD 2D	Layer	Output Neuron	Kernel size	Stride	Padding	Normalization
TRUE	Conv2d	32	7	1	0	Batch Normalization
FALSE	Conv2d	64	2	1	1	
FALSE	Conv2d	128	3	2	1	
TRUE	Conv2d	128	3	1	0	
TRUE	Conv2d	128	3	1	0	
TRUE	Conv2d	128	3	1	0	
TRUE	Conv2d	128	3	1	0	
TRUE	Conv2d	128	3	1	0	
TRUE	Conv2d	128	3	1	0	
TRUE	Conv2d	128	3	1	0	
TRUE	Conv2d	128	3	1	0	
TRUE	Conv2d	128	3	1	0	
TRUE	Conv2d	128	3	1	0	
TRUE	Conv2d	128	3	1	0	
TRUE	Conv2d	128	3	1	0	
TRUE	ConvTrans 2d	64	3	2	1	
TRUE	ConvTrans 2d	32	3	2	1	
TRUE	Conv2d	3	7	1	0	/

4.3 Improved Cycle GAN

In this case, we use Generator and Discriminator architecture as below:

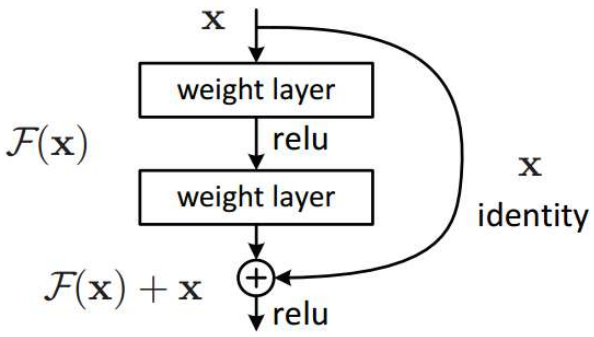


A generator in CycleGAN normally consists of Encoder and Decoder. In this improved model, a transformer with 6 Resnet blocks were added to avoid the possible gradient vanishing resulted from adding the depth of the network.

Having batch size as 1, GAN uses instance normalization rather than batch normalization like other models. Leaky ReLU is normally used in GAN architecture, while in this case, it is only used in decoder block in generator. The encoder of GAN down sample input images and extract features from all the images. In our model, it has architecture below:

Layer	Filters	Kernel Size	Stride	Normalization	Activation
Conv2D	64	7*7	1	Instance Normalization	ReLU
Conv2D	128	3*3	2		
Conv2D	256	3*3	2		

Complex model sometimes comes with degradation problem, which is, it fails to learn simple functions. Resnet is used to solve the problem. In this case, six same Resnet block is connected to Encoder and each Resnet Block has architecture and parameters below:



Layer	Filters	Kernel Size	Stride	Normalization	Activation
Conv2D	256	3*3	1	Instance Normalization	ReLU
Conv2D	256	3*3	1		/

The decoder in GAN uses several transposed convolutional layers to up sample the feature matrix. Leaky ReLU is also used in this block. To solve gradient vanishing problem, skip connection is also set between encoder and decoder layers with same size. The detailed parameters are shown below:

Layer	Filters	Kernel Size	Stride	Normalization	Activation
Conv2DTranspose	256	3*3	2	Instance Normalization	LeakyReLU
Conv2DTranspose	128	3*3	2		
Conv2DTranspose	65	7*7	1		

Discriminator in GAN works as a classifier and identify the whether the input is real images or images generated by generator. Convolutional layers with stride are used instead of convolutional layers with pooling layers. The architecture is shown below:

Layer	Filters	Kernel Size	Stride	Normalization	Activation
Conv2D	64	4*4	2	/	ReLU
Conv2D	128	4*4	2	Instance Normalization	
Conv2D	256	4*4	2		
Conv2D	512	4*4	1		

5. Results

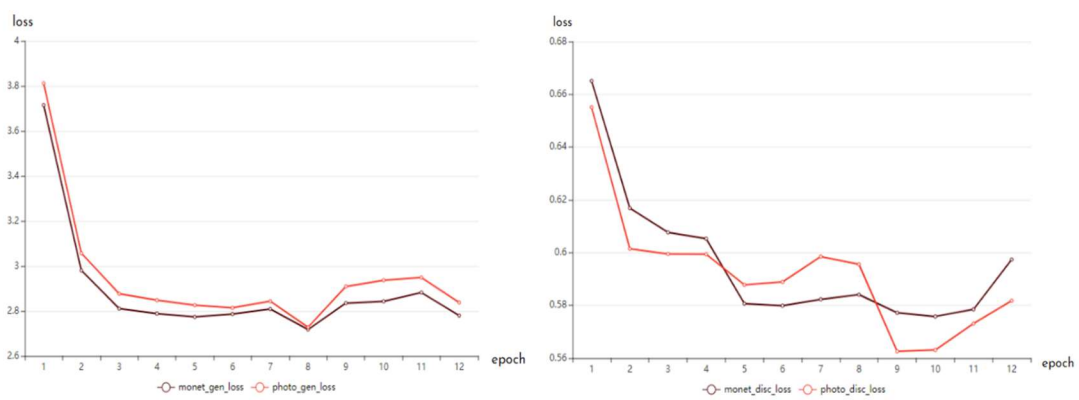
5.1 Loss

We used two frameworks to build the model and compare the loss value of the best models:

	moent_gen_loss	**photo_gen_loss**	**monet_disc_loss**	**photo_disc_loss**
Keras	3.9370	4.0263	0.4280	0.4367
PyTorch	7.3176	5.7076	0.4807	0.4925

As shown in the figure above, the trained Keras model is better than the PyTorch model. It may be because the Keras model has data augmentation in preprocessing and has more training epochs.

The loss curve of the Keras model over time is shown below:



As shown in the figure above, the generator's loss value reaches the lowest at the 8th epoch, and then it starts to fluctuate.

The discriminator has fluctuations at the beginning and reaches the lowest in the 9th-10th epoch. When the generator learned how to generate more realistic models, the discriminator loss began to increase at 11th epoch. Because of the time limit, we chose 10 epochs model as our final output.

5.2 Cycle Evaluation

The picture generated by the generator cycle is shown below:

The figure above shows Monet's paintings generated by monet generator with input photo. The image on the right are photos generated by photo generator taking fake Monet’s paintings as input

We can see that generators have better performance on some pictures like that in the first and second set, and the last set of pictures show much noise because of large color blocks.

The figure above shows photos generated by photo generator with Monet paintings data as input. Then take photos as input and generate Monet paintings through a monet generator.

It can be seen from the figure that the photo -> Monet's paintings -> photo cycle works well.

5.3 Monet Generation

The main purpose of this project is to generate Monet paintings from photos. Some manually picked images generated by monet generator are shown as follows:

As shown in the figure, some output images of the network are not very realistic. We can see some pictures with Monet's style, but others having noise. There is still a long way to go to.

5.4 Running time comparison

We have tried to use several platforms and different hardware to train models. The time comparison results are as follows:

Accelerator	GPU on AWS	GPU on Kaggle	TPU on Kaggle
Time	600s/epoch	233s/epoch	113s/epoch

As shown in the table, using the TPU on Kaggle to train the model has a large advantage than that on GPU. However, there is a time limit for using GPU or TPU on Kaggle, so if you need long time and multiple training, GPU instance on AWS should be chosen.

6. Summary and conclusions

6.1 Conclusion

In the comparison of the two models, the improved Tensorflow & Keras model with more preprocessing has a better performance.

Under the trial of different platforms and hardware, the TPU on Kaggle got the best results and the shortest time.

When training the Tensorflow & Keras model, it takes long to train on more data and more epochs, which means we could not get satisfactory results with more suitable epoch or learning rate. But the method is feasible, and more adjustment is available.

Our submission on Kaggle got a score of 64.0 and ranked at 134^{th}.

6.2 Future work

In future work, we will simultaneously improve the models implemented with two frameworks. We will add data augmentation part to the model implemented by PyTorch and train more epochs. We will also use the learning rate scheduler in the model implemented with Keras to control the model and choose the best learning rates.

In addition, we will continue to upload the code to Kaggle and get better scores by constantly improving the model.

7. References

[1] https://machinelearningmastery.com/what-is-cyclegan/

[2] https://www.lyrn.ai/2019/01/07/instagan-instance-aware-image-to-image-translation/

[3] https://www.kaggle.com/amyjang/monet-cyclegan-tutorial

[4] https://www.kaggle.com/dimitreoliveira/introduction-to-cyclegan-monet-paintings

[5] https://www.kaggle.com/dimitreoliveira/improving-cyclegan-monet-paintings

[6] https://www.kaggle.com/doanquanvietnamca/the-beauty-of-cyclegan

[7] https://www.tensorflow.org/tutorials/load_data/tfrecord