Final Report for Map Generation

Barry Xiong[#1]

[#]*Electrical & Computer Engineering, Stevens Institute of Technology*
Hoboken, New Jersey, United States
[1]bxiong1@stevens.edu

***Abstract*— Image generation is still a hot topic in computer vision, in this paper I will discuss two different algorithms, pix2pix and cycleGAN, that take satellite map images as inputs, and the models will be able to generate the google map like images.**

***Keywords*—Image generation, image-to-image translations, GAN, Deep Learning[1]**

I. Introduction

Deep learning is gaining more and more interests in the Artificial Intelligence area. One of the hot topics is image generation, in which we want to create a model which has the ability to "*generate an image that is indistinguishable from the real image*". Such an idea leads to the creation of Generative Adversarial Networks (GANs) [1], a type of the learning algorithm that is composed of the Generator and Discriminator. This type of algorithm can learn the distribution, i.e. the mean and the variance of an image, and reproduce the similar images from the learnt distributions. However, for image-to-image translations, in which the goal is to "translate" an input image to an output image, using simple GANs is not enough to produce such high quality output images. Thus, in this report, I will be introducing two types of different algorithms, Pix2Pix [2] and CycleGAN [3], and generate the google map like images by giving the satellite images as inputs. Then I will compare and discuss the results from both of the algorithms.

II. Datasets

The dataset is shown as the following images:

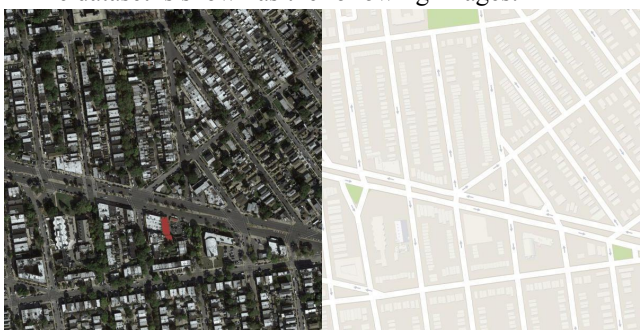


Figure 1.

As Figure 1 shows that this image is composed of two different images, the left hand side is the image from the satellite, known as satellite image, the right hand side is the corresponding google map image that we can always view on the Google Map website. The main task here is we are inputting the left hand side satellite image and we want our generator from the Pix2Pix or CycleGAN algorithm to have the ability to produce the image much like the right hand side. This ability is so-called image-to-image translation.

Under this dataset, it consists of train and validation files. In the train file there are 1096 composed images (just like Figure 1 but with different satellite images and corresponding google map images). Similarly, the validation dataset contains 1098 composed images. The dataset can be found and downloaded with this:
https://drive.google.com/drive/folders/17XsmPkgGLn7pQiSHBSgLHNQa42a_29v2?usp=sharing

III. Background knowledge & methodologies

In this section I will be introducing some of the background knowledge and basic model of GANs and CGANs [4] which are important to understand. Then, I will talk about the pix2pix and cycleGAN that I used to come across with this map generation problem.

1.Generative Adversarial Networks

1.1.Motivations

Generative Adversarial Network is a type of algorithm that is mainly used for image generation problems. In the old days, many researchers used ConvNets for image generations, however, the typical loss function, e.g. MSE loss is only able to generate the general structure of the image, but with detailed features due to the MSE loss, it will produce really blurry results. That is the main motivation which leads to the birth of GAN. GAN is able to learn the distribution from the image and it will reproduce an image based on the distribution that the GAN learnt from the image.GAN model has been proven for decent result on the image generation tasks, which is also the reason that I chose this type of algorithm for map generation in my case. In the

[1] **Source code available at:**
https://github.com/bxiong1/AAI628Project.git

next subsection the discussion on how GAN works will be further explored.

1.2.Approaches and Formula

The typical GAN is composed of two parts, a **Discriminator** and a **Generator**. The structure of a typical GAN shows below:

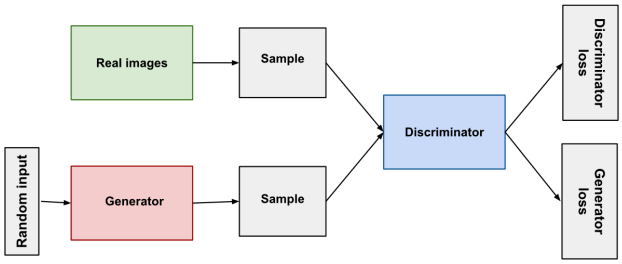


Figure 2. [5]

A **Discriminator** is mainly used for distinguishing the fake image from the real image. A **Generator** on the other hand generates image data, and these data are fed into the Discriminator as negative inputs (fake images). The ultimate goal is that the Discriminator will try its best to mark the Generator's image as negative, and the Generator on the other hand will try to trick the Discriminator such that it cannot interpret the output of the Generator as false input. This concept comes from Game Theory, and the solution for this problem will be called "Nash-Equilibrium".

The loss function of GAN is defined as:

$$min_G max_D V(D,G) = E_x[log(D(x))] + E_x[log(1 - D(G(x)))]$$

The minimax loss function indicated above shows the same underlying implication as the above explanation. We want to maximize the Discriminator's likelihood $D(x)$ to produce the correct labels for both real data and fake data. In other words, we want the Discriminator to be as strong as possible to put "correct" on real data and "incorrect" on the fake data generated from Generator. We also simultaneously train the Generator to minimize the loss of $log(1 - D(G(x)))$ which implies that we want the generator to trick the Discriminator such that the Discriminator will not be able to distinguish the fake image from the real image because both images are look very similar i.e. they both have similar mean μ and variance σ^2

Notice that both Generator and Discriminator are Convolutional Neural Networks, and the input for Generator will be random noise in this model, and the noise will be transform into a fake image, then the fake image will be sent as input along with the real image into the Discriminator, which will produce a true/false probability indicates whether they are fake or not.

1.3.Limitations and Discussions

Even though GAN has shown good results on the image generation task, there is one big limitation on GAN when generating the fake images. GAN is able to generate some real images based on learning the distribution of the training dataset. However, since GAN takes the input as random noise, the generated image from GAN is also random. In other words we cannot have full control on whether the GAN will produce the image that we want in the first place. The model will probably take several times to produce the image until it reaches the image that we want. For instance, if we use the "MINIST" dataset to generate the hand-written numbers, the typical GAN algorithm is only able to generate random handwritten number from 0-9, but if we want to generate 5, GAN does not guarantee to produce a hand-written 5 on the first try of the generation phase. This is not efficient especially in my task where we want to generate a specific place of the map from the generator. Thus, I need to find a GAN type algorithm that is able to generate images in a certain direction instead of generating randomly. This leads to CGAN which will be discussed in the following section.

2.Conditional Generative Adversarial Network

2.1. Motivations

As the problem mentioned in the last section, a typical plain GAN algorithm has limitations, which is critical to my map generation problem. Thus, the new type of GAN model Conditional Generative Adversarial Networks, also known as CGAN, is introduced. Unlike GAN, CGAN has the ability to take target labels as the input of both the Discriminator and Generator. By taking the labels as inputs, CGAN is able to generate images that are based on the labels instead of generating random images.

2.2. Approaches and Formulas

Similar to GAN's structure, the CGAN also has mainly two parts: Discriminator and Generator.

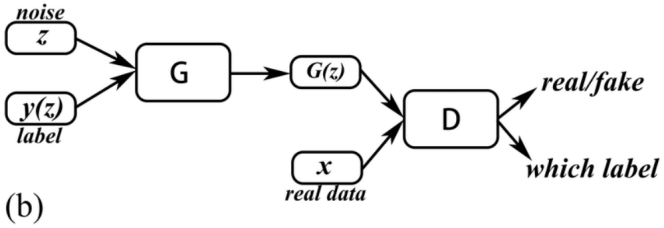


Figure 3. [6]

The above figure shows the general model of the CGAN, as mentioned before, CGAN is a similar model to plain GAN, except that there is a extra input which is called "label" into the Generator, the purpose of this label is help Generator to quickly

identify which image it should generate instead of randomly generating images. Then when the image is
generated, it will pass into the Discriminator along with the real data, which also contains labels. Then the Discriminator will produce two results:

1. Whether the it is real or fake image
2. Whether this generated image belongs to the labels indicated in real data.

Since we modify the structure of the model, we should consider changing the loss of the GAN. The modified version of the loss function can be shown as below:

$$min_G max_D V(D,G) = E_x[log(D(x|y))] + E_x[log(1 - D(G(x|y))$$

The Discriminator's likelihood adds a conditional variable y. The changes of the loss function implies that given the labels, we want to train a Discriminator that can correctly point out the real data from the fake data as well as the Generator will produce data that first of all belongs to this label y and the data can also trick the Discriminator.

One thing should be noted that the "labels" might not only be categories or numbers, it could also be images. This gives much more room for CGAN to do image-to-image translation problems.

2.3. Limitations and Discussions

CGAN overall satisfies the requirements of solving the map generation problems, since CGAN has the ability to take in some useful information inputs for both Discriminator and Generator as a guide. However, there are still minor tweaks that need to be done during the implementation phase in order to better solve the problems and produce high quality images. For instance, the structure of building the Discriminator can use PatchGAN, and the structure of building the Generator can use U-Net. These will be discussed further in the coming sections.

3.Pix2Pix

3.1.Motivations

CGAN is proven to be effective for image-to-image translation problems. Pix2Pix algorithm is built based on the philosophy of CGAN but with some modifications on the implementations of the Discriminator and Generator. Moreover, in the original literature of Pix2Pix [2], the author has used some similar translation datasets of satellite map to google map to prove that Pix2Pix has the ability to generate the overall good quality map images. Thus, I want to implement this algorithm and verify whether the Pix2Pix model has such an ability which is said in the paper.

3.2.Approaches and Formulas

Similar to CGAN and plain GAN model, Pix2Pix model also contains two main parts: Discriminator and Generator. However, the implementation process of the Discriminator and Generator, specifically the Discriminator uses the **PatchGAN** model to implement, and the Generator uses the **U-Net** model.

PatchGAN: The model is still a Covnet, since the Covnet is able to process the image patches independently and identically. The main difference between a PatchGAN and a simple Covnet is that a simple Covnet only has the ability to produce a single scalar as the output of the Discriminator, and use this single scalar to predict whether it is a fake or real image. In other words, the single scalar is an indication probability of the whole input image. Whereas the PatchGAN is able to produce an NxN matrix output A, and each a_ij in the NxN matrix signifies whether the ij patch in the original input image is real or fake [7]. In my case, I take in an input image with a image size of 286x286 and use CNN to reduce the size and output a 30x30 matrix output, and each element in the output will represent a probability of a 70x70 patch/portion of the input image indicating whether this patch/portion is real or fake. The 70x70 image patch/portion is also known as an effective receptive field. The formula for calculating the receptive field is:

$$R_{l-1} = K + S \cdot (R_l - 1)$$

Where,

R_{l-1} is the receptive field from previous convolutional layer

K is the kernel size

S is the stride size

R_l is the receptive field from the current layer

Thus, by applying the formula above five times we can backtrack the single pixel of the output 30x30 matrix back to 70x70 patches of the original input image. In addition, the higher the receptive field is the deeper the network. Thus, in theory we could choose a 286x286 receptive field, but such a high receptive field might not lead to a better image quality; it in fact will decrease the overall performance of the Discriminator. Thus, 70x70 is a good choice for a receptive field. [2][7]

Below is the implementation diagram:

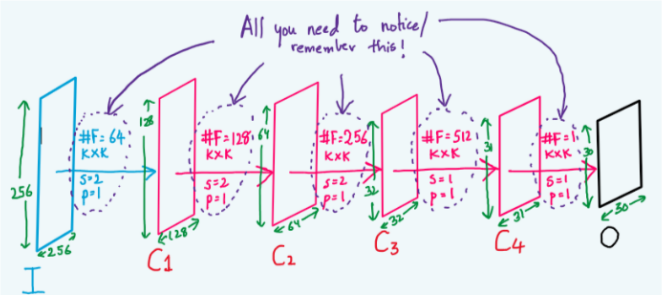


Figure 4. [8]

The general structure and specifications(kernel size, strides, padding) of PatchGAN discriminator model is the followings:

```
Discriminator(
  (initial): Sequential(
    (0): Conv2d(6, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), padding_mode=reflect)
    (1): LeakyReLU(negative_slope=0.2)
  )
  (model): Sequential(
    (0): CNNBlock(
      (conv): Sequential(
        (0): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), bias=False, padding_mode=reflect)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2)
      )
    )
    (1): CNNBlock(
      (conv): Sequential(
        (0): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), bias=False, padding_mode=reflect)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2)
      )
    )
    (2): CNNBlock(
      (conv): Sequential(
        (0): Conv2d(256, 512, kernel_size=(4, 4), stride=(1, 1), bias=False, padding_mode=reflect)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2)
      )
    )
    (3): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), padding=(1, 1), padding_mode=reflect)
  )
)
```

Figure 5.

The implementation of the Discriminator in Pytorch can be viewed in **Appendix A.** One important thing about the Discriminator is that it takes a concatenation of X and Y images. In my case, X is the satellite images, and Y could be the corresponding google map images or it can also be the generated images. These two images concat on channel dimension before sending into the Discriminator for predictions.

U-Net: This model is an improvement on the normal encoder and decoder implementations.

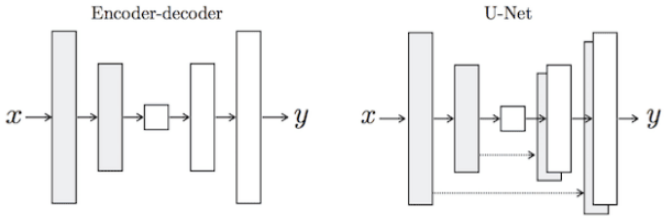


Figure 6. [2][7]

The comparison between the encoder-decoder structure and U-Net structure is mainly skip connections. The encoder-decoder structure looks like an Autoencoder where we downsample the original image to a bottleneck and then we upsample and try to retrieve back the original image. Such an implementation will work for simple images, e.g. hand-written numbers. However, with complicated images, encoder-decoder structure will not perform as well as expected due to the information loss when compressing the images. Thus, U-Net adds skip connections, which allows the model to learn and share more of the low-level features and predict better overall image quality in general. In my implementations, I take the input image size 256x256, and then downsample 6 times with each convolutional kernel size is (4,4); stride is 2; padding is 1 followed by a Batch Normalization and a LeakyReLU activation function. After that we downsample 1 more time with the same kernel size, stride, padding and activation function to reach the bottleneck feature map, which the channels size is 512 and image size is 1x1. Finally, we upsample the bottleneck back to a 256x256 size image (Note: upsampling performs with the same kernel size, stride and padding but with ReLU activation function). However, when we upsample the feature maps, we need to concatenate them with the downsample feature maps, which have the same channel size and image sizes. The implementation details of U-Net can be viewed in **Appendix B**.

The overall training process of Pix2Pix model can be interpreted as the followings:

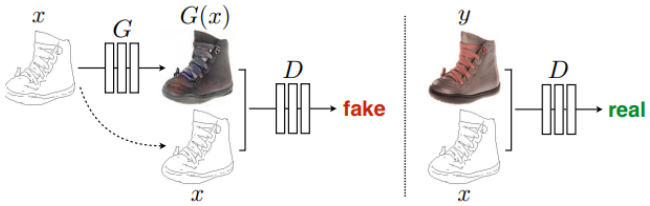


Figure 7.[2]

Similar to GAN and CGAN, we want to train the Discriminator to be able to correctly label the real and fake images given the original input image, whereas the generator will be trained such that it can produce images to trick the Discriminator.

The loss function for training Pix2Pix model contains the general loss from CGAN:

$$min_G max_D V(D,G) = E_x[log(D(x|y))] + E_x[log(1 - D(G(x|y)))]$$

In addition to this general CGAN loss, there is one more L1 loss for Generator, the L1 loss will be minimized during the training process by taking the absolute difference between the generated google map and the original google map corresponding to the input satellite image.

$$L1 = ||fake_y - real_y||_1$$

Note: The L1 loss is chosen over the L2 loss due to the experiment findings that L1 loss produces less blurry results than L2 loss [2].

Thus, the overall training loss is:

$$loss = min_G max_D V(D,G) + \lambda \cdot min_G L1(G)$$

3.3.Results and Discussions

The results generated by this Pix2Pix model are shown in **Appendix C.** As we could see that the Pix2Pix shows some decent results for generating the google map like images. However, most of them are not as good as the ones mentioned in the original papers. Especially when the generated fake images still lack feature details on the map even though the model is able to depict the general structure of maps. Thus, I need to find a different model that could potentially improve the problems that Pix2Pix has, which leads to CycleGAN.

4.CycleGAN

4.1. Motivations

CycleGAN is also one of the popular models that is designed for image-to-image translations. However, its model is differently designed than the Pix2Pix model. The main difference in CycleGAN is that it is not a one-way translation model, it is designed to translate the image to another image back and forth. CycleGAN has applied to several different tasks, e.g. convert zebra to horse or horse to zebral; convert summer to winter or winter to summer; and painting to photo or photo to painting. These tasks turn out to have good image results. I have closely investigated these tasks, and I found that the translation of these images requires a model to learn much of the detailed features. For the Map Generation tasks, in order to let the generator generate high quality images, it needs to learn those detailed features, such as houses. Moreover, CycleGAN is trained under a more complicated loss than Pix2Pix, thus the expectation for CycleGAN to generate higher quality google map images is doable.

4.2. Approaches

As always, CycleGAN is composed of Discriminators and Generators. However, CycleGAN uses two separate pairs of Discriminator and Generator whereas Pix2Pix only uses one pair of Discriminator and Generator.

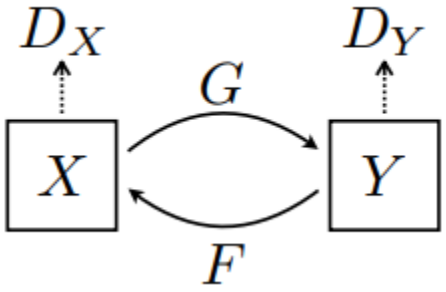


Figure 8.[3]

As mentioned above, CycleGAN needs two sets of Discriminators and Generators. In Figure 8. The dataset X is trained with Discriminator D(X) and dataset Y is trained with Discriminator D(Y), the interesting part is the Generator G is responsible to generate image that has the same distribution as Y taking X as input, and Generator F is responsible to do the reverse. In my Map generation task, X is the satellite image and Y is the google map images. Thus, we train D(X) to only distinguish whether it is a satellite image, and train D(Y) to only determine whether it is a google image. Then Generator G will generate google map images by taking the satellite image as input, whereas Generator F will take google map images and generate satellite images. Then G is responsible to trick D(Y) and F will be responsible to trick D(X).

The Discriminator model is also a PatchGAN, but instead of taking 286x286 image size as inputs, this PatchGAN takes 256x256 as inputs. Moreover, the Discriminator of CycleGAN takes only one input: either original satellite image, corresponding google map images or generated images. The implementation details are shown in **Appendix D.**

The Generator of CycleGAN on the other hand is different from Pix2Pix. It mainly looks like an encoder-decoder block; however, when it reaches the bottleneck, it contains a total number of 9 ResNet Blocks.

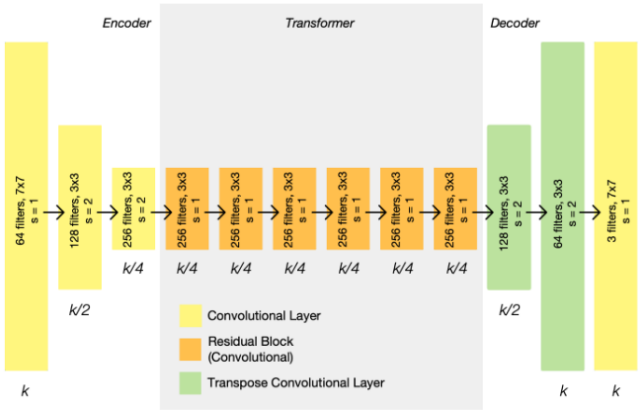


Figure 9.[9]

Note that instead of having 6 ResNet Blocks, my implementation has 9. The rest of the implementation is the same as Figure 9. And the implementation code is in **Appendix E.**

The main difference between CycleGAN and Pix2Pix is the training loss functions. The training loss is composed of two parts: general adversarial loss and cycle loss [3]. The general adversarial loss is the same with the other GAN type loss

$$min_G max_{D_Y} V(D_Y, G) = E_y[log(D_Y(y))] + E_x[log(1 - D_Y(G(x)$$

$$min_F max_{D_X} V(D_X, F) = E_x[log(D_X(x))] + E_y[log(1 - D_X(F(y)$$

Here since we have two Discriminators and Generators, we will have two adversarial losses to train Discriminators (D_Y and D_X) and Generators($G(x)$ and $F(y)$). The main goal for this training is still the same for other Discriminators, in which it trains each Discriminator to correctly recognize whether the input is real or fake for X and Y respectively. The Generators will have the duty to generate Y given X to trick the Discriminator D_Y and vice versa.

However such a general adversarial loss is not enough, it can only bring the generated data to the same domain of data. It cannot guarantee to generate the output that is the same as the given input. Thus, cycle loss is introduced to handle such issues.

$$L_{cyc}(G, F) = E_x[||F(G(x)) - x||_1] + E_y[||G(F(y)) - y||_1]$$

The formula implies that if the transformation is made on dataset X to Y (G(x)), when we reverse the transformation back from Y to X (F(G(x))) should be exactly the same as original input data X. In my case, if I input a satellite image X, and I generated a google image Y, when I convert the generated google image back to satellite image, it should be identical to the satellite image I put in at the beginning. Same idea applies from dataset Y to X. Thus, the overall loss function will be:

$$L(G, F, D_X, D_Y) = min_G max_D V(D_Y, G) + min_F max_D V(D_X, F) + \lambda \cdot min_G min_F L_{cyc}(G, F)$$

4.3.Results and Discussions

The results from CycleGAN are shown in **Appendix F.** It is obvious to say that the generated google map images, even though it is not perfect for every image, have better feature details depicted than the generated results from the Pix2Pix model.

IV. Implementations and Metrics

In the following sections I will be discussing and showing the implementations of data processing, the train function for Pix2Pix and CycleGAN respectively and the metrics, known as Frechet Inception Distance (FID), used to measure the quality of the generated images compare with the original images.

1. Data Processing

As mentioned in the DATASETS sections, each image dataset is composed of two different images. The left hand side of the image in Figure 1 is the satellite map image, and the right hand side image in Figure 1 is the corresponding google map images. For both Pix2Pix and CycleGAN we want to produce the output images that look like the google map images by inputting the satellite map images. To do so, we need to first split the Figure 1 image into two parts, in which the satellite map image is indicated as X and corresponding google map image is indicated as Y. The midpoint of the Figure 1 image is the 600th pixel, this indicates that we can split the width of this image into two parts from 0 to 599 pixels is known as X data and from 600 to the end of Figure 1 will be the Y data.

After splitting the Figure 1 into two separate data, I also apply different image augmentations to the data. The reason for data augmentations will help to train a more robust model for both Pix2Pix and CycleGAN. There are basically four simple augmentation techniques I applied: **Resize, Horizontal Flip, Color Jitter, and Normalization**.

Resize: this augmentation helps to resize the images to 256x256 in case the image is not the size that is suitable to input into Discriminator and Generator.
Horizontal Flip:the augmentation helps to flip the image on the vertical axis with a certain probability (p=0.5 in my case).
Color Jitter: Randomly changes the image brightness, saturation and contrast with a certain probability (p=0.2 in my case).
Normalization: normalize the image pixels according to the predefined mean and variance (mean=[0.5, 0.5, 0.5], variance=[0.5,0.5,0.5] the values are gotten from ImageNet normalization).

For this dataset, I use **Resize, Horizontal Flip** on both X (satellite map image) and Y (google map image). In addition, I applied additional **Horizontal Flip, Color Jitter, and Normalization** on X and only applied **Normalization** on Y. In the end I transform them both to tensor since it is required to process under the Pytorch environment.

I repeat the data processing procedure, mentioned above, for the entire train dataset and validation dataset to obtain the augmentation of the satellite map images and google map images for training. The implementation of the data processing is shown in **Appendix G.**

2.Training Implementations

2.1.Training for Pix2Pix

After obtaining the processed data, the Discriminator and the Generator can be trained. The main idea of training is that I first load in the Discriminator model and Generator model, and input satellite map image X to obtain a generated google map Fake_Y. Applying the Binary Cross Entropy loss (BCE) to train the Discriminator.

```
y_fake = gen(x)
D_real = disc(x, y)
D_real_loss = bce(D_real, torch.ones_like(D_real))
D_fake = disc(x, y_fake.detach())
D_fake_loss = bce(D_fake, torch.zeros_like(D_fake))
D_loss = (D_real_loss + D_fake_loss)/2
```

Figure 10.

Note that the Discriminator is trained such that to label the real Y as the correctly real image and Fake_Y will be incorrect fake images given the X input. That is the reason we put *torch.ones_like* in the D_real_loss and *torch.zeros_like* in the D_fake_loss. Moreover, we add up the loss and divide by 2 to produce the total loss function in the Discriminator.

The Generator on the other hand needs to trick the Discriminator such that the Discriminator will give a real label on the Fake_Y.

```
D_fake=disc(x, y_fake)
G_fake_loss = bce(D_fake, torch.ones_like(D_fake))
L1 = l1_loss(y_fake, y)*L1_LAMBDA
G_loss = G_fake_loss + L1
```

Figure 11.

This is the reason we put *torch.ones_like* in the G_fake_loss. In addition the L1 loss mentioned in the paper helps the Pix2Pix model generate an image that will look much the same as the Y. Then we add the losses together to form a Generator Loss.

In the Training process of Pix2Pix model, I used learning rate is 2×10^{-4} and number of epochs as 500, the $\lambda = 100$ for L1 loss. The full implementation is shown in **Appendix H.**

2.2.Training for CycleGAN

Training CycleGAN is a bit different from Pix2Pix, since the model has two Discriminators and two Generators. Instead of using BCE loss here, the CycleGAN is trained with MSE loss. I first used Generator G and input satellite map image X to generate the Fake_Y (fake google map image). Then the MSE loss is applied to train Discriminator D_Y to correctly label real google map image Y as "real" and fake generated google map image Fake_Y as "fake". Similar approach for training Discriminator D_X except the input for D_X will be satellite image X and the Fake_X (fake satellite image generated by using Generator F given Y). After that I add two Discriminator losses together and divide by 2.

```
fake_y = gen_Y(x)

D_Y_real = dis_Y(y)
D_Y_fake = dis_Y(fake_y.detach())

D_Y_real_loss = MSE_LOSS(D_Y_real, torch.ones_like(D_Y_real))
D_Y_fake_loss = MSE_LOSS(D_Y_fake, torch.zeros_like(D_Y_fake))

D_Y_loss = D_Y_real_loss + D_Y_fake_loss

fake_x = gen_X(y)

D_X_real = dis_X(x)
D_X_fake = dis_X(fake_x.detach())

D_X_real_loss = MSE_LOSS(D_X_real, torch.ones_like(D_X_real))
D_X_fake_loss = MSE_LOSS(D_X_fake, torch.zeros_like(D_X_fake))

D_X_loss = D_X_real_loss + D_X_fake_loss

D_loss = (D_Y_loss + D_X_loss)/2
```

Figure 11.

Training the Generator requires two types of losses. As the first part is similar to Pix2Pix except I use two MSE losses each with a Generator (G and F) instead of using one BCE loss. Another loss is known as cycle loss. The implementation is simply applying two L1 losses, one is responsible to minimize the distance between G(X) and Y another is responsible to minimize the distance between F(Y) and X.

```
D_Y_fake = dis_Y(fake_y)
D_X_fake = dis_X(fake_x)

loss_G_X = MSE_LOSS(D_X_fake, torch.ones_like(D_X_fake))
loss_G_Y = MSE_LOSS(D_Y_fake, torch.ones_like(D_Y_fake))


#cycle loss
cycle_x = gen_X(fake_y)
cycle_y = gen_Y(fake_x)

CYCLE_X_loss = L1_LOSS(x, cycle_x)
CYCLE_Y_loss = L1_LOSS(y, cycle_y)
```

Figure 12.

Similar to Pix2Pix, I used learning rate is 2×10^{-4} and number of epochs as 500, the $\lambda = 100$ for L1 loss. In addition there is one more coefficient $\lambda_{cyc} = 10$ for cycle loss. The purpose of using the same setting is to test the performance of two different models. The full implementation is shown in **Appendix I.**

3.Frechet Inception Distance

The metric Frechet Inception Distance is selected as a measurement of the quality of the generated images. The FID is originally based on Inception Score (IS). Inception Score uses the pretrained Inception-V3 model and take in the generated images (i.e. Fake_Y google map image in my case) and then classify the generated image to one of those 1000 categories [10]. However, such an approach is not enough to compare to a real image (the real google map image Y). Thus, the FID is introduced. FID also uses the pretrained Inception-V3 model but FID does not use the classification layer of the Inception-V3. It only uses upto the layer pooling layer prior to the classification layers to generate the feature maps as the output. In my case, the Inception-V3 will take in the fake google map images (Fake_Y) and generate a fake feature map. Then, the Inception-V3 will take in the real google map image (Y) to generate a real feature map. After that FID calculate the mean and the covariance for both fake and real feature maps, and obtain the distance by applying the formula:

$$d((m_g, C_g), (m_r, C_r)) = ||m_g - m_r||_2^2 + Tr(C_g + C_r - 2(C_g C_r)^{0.5})$$

Where,

m_g and m_r are the mean of the feature maps from generated (Fake_Y) and real google map image (Y) respectively,

C_g and C_r are the covariance of the feature maps from generated (Fake_Y) and real google map image (Y) respectively,

$Tr(\cdot)$ is known as trace linear algebra function, which takes only the main diagonal elements and sums them up [11].

In general, if the FID is lower the better the quality of the generated images are. Thus, by comparing FID numerical values, I can tell whether the CycleGAN has outbeaten the Pix2Pix on Map Generation task.

The way that I generate FID for Pix2Pix and CycleGAN is that I trained the model and I passed in some of the input satellite images X and produced the generated google map images Fake_Y. Since each input X has a corresponding real google map image Y, I can generate FID on the generated google map and real google map. The implementation of FID is done in Keras, and it is learnt from [10] with some minor modifications shown in **Appendix J.**

V. Conclusions & Future Improvements

After applying two methods, these are the summary of the results.

Method	Score FID
Pix2Pix	32.748
CycleGAN	22.973

Table 1.

As we can see from the numerical comparison of the FID scores, we can conclude that CycleGAN can generate much better maps than Pix2Pix, since the FID for CycleGAN is lower than Pix2Pix's FID. This is also true when I am making the visual comparisons between the generated images from CycleGAN and Pix2Pix (in **Appendix F** and **Appendix C**). The generated image from CycleGAN has much more feature details (e.g. houses on the map) than Pix2Pix. Moreover, the CycleGAN has a much clearer structure than the generated images from Pix2Pix.

However, in terms of time that it takes to train the CycleGAN is about 4 hours, whereas the time that it takes to train Pix2Pix is around 3 hours. This is reasonable since Pix2Pix is a simpler model, it only contains a Discriminator and a Generator. The CycleGAN, on the other hand, has two Discriminators and Generators. In terms of loss functions, Pix2Pix has only one general adversarial loss and L1 loss, but CycleGAN has two adversarial losses and two L1 losses in total. Thus, CycleGAN should take longer to train than Pix2Pix.

Therefore, in general I can conclude that for the image-to-image map generation task, CycleGAN has much better accuracy and is able to generate better quality google map images than Pix2Pix given a fixed set of hyperparameters(e.g. Epochs, learning rate), even though CycleGAN takes a bit longer to train.

One of the improvements that I could make in the future. First of all, even though the Pix2Pix model is less accurate than CycleGAN, but it does not mean that Pix2Pix cannot produce the same quality image like CycleGAN does. In this project I only train 500 epochs since I want to make a comparison between two models. I could extend the epochs to 1000 and test out the results. The longer it trains there is potential to obtain better results, but we have to keep the learning rate large so that the convergence will not fall into local minias. Moreover, the longer it trains, the overfitting problem could be crucial. Thus, I have to carefully select the hyperparameters so that it also avoids overfitting.

Another improvement is that in CycleGAN, the Generator uses the encoder-decoder structure with some ResNet Blocks in between. I could also try to implement U-Net as the Generator to test whether the generated results from CycleGAN are better. U-Net, as discussed above, has better performance than encoder-decoder on exchanging the lower level information due to the skip connections. A good quality Google Map Generation requires lots of lower level information. Thus, U-Net could be a good structure to choose as Generators.

Acknowledgment

I want to acknowledge Professor Rensheng Wang, for providing the basic content of background knowledge of algorithms. Moreover, I want to acknowledge Aladdin Persson for providing the basic implementation of the Pix2Pix model and CycleGAN models that help me to develop on this Map Generation project [12].

Reference:

[1] "Generative adversarial nets - arxiv.org." [Online]. Available: https://arxiv.org/pdf/1406.2661.pdf. [Accessed: 06-May-2022].

[2]P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, "Image-to-image translation with conditional adversarial networks," *arXiv.org*, 26-Nov-2018. [Online]. Available: https://arxiv.org/abs/1611.07004. [Accessed: 06-May-2022].

[3] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired image-to-image translation using cycle-consistent adversarial networks," *arXiv.org*, 24-Aug-2020. [Online]. Available: https://arxiv.org/abs/1703.10593. [Accessed: 06-May-2022].

[4] M. Mirza and S. Osindero, "Conditional generative adversarial nets," *arXiv.org*, 06-Nov-2014. [Online]. Available: https://arxiv.org/abs/1411.1784. [Accessed: 06-May-2022].

[5]"Overview of gan structure | generative adversarial networks | google developers," *Google*. [Online]. Available: https://developers.google.com/machine-learning/gan/gan_structure. [Accessed: 06-May-2022].

[6] "Architecture of (a) gan and (b) CGAN. - researchgate.net." [Online]. Available: https://researchgate.net/figure/Architecture-of-a-GAN-and-b-CGAN_fig4_336887651. [Accessed: 06-May-2022].

[7] J. Brownlee, "A gentle introduction to pix2pix generative adversarial network," *Machine Learning Mastery*, 05-Dec-2019. [Online]. Available:

https://machinelearningmastery.com/a-gentle-introduction-to-pix2pix-generative-adversarial-network/#:~:text=Pix2Pix%20is%20a%20Generative%20Adversarial,presented%20at%20CVPR%20in%202017. [Accessed: 06-May-2022].

[8] S. -, "Understanding Patchgan," *Medium*, 28-May-2020. [Online]. Available: https://sahiltinky94.medium.com/understanding-patchgan-9f3c8380c207. [Accessed: 06-May-2022].

[9]S. Wolf, "Cyclegan: Learning to translate images (without paired training data)," *Medium*, 20-Nov-2018. [Online]. Available: https://towardsdatascience.com/cyclegan-learning-to-translate-images-without-paired-training-data-5b4e93862c8d#:~:text=Generator%20Architecture,composed%20of%20three%20convolution%20layers. [Accessed: 06-May-2022].

[10]J. Brownlee, "How to implement the Frechet Inception Distance (FID) for evaluating Gans," *Machine Learning Mastery*, 10-Oct-2019. [Online]. Available: https://machinelearningmastery.com/how-to-implement-the-frechet-inception-distance-fid-from-scratch/#:~:text=for%20Real%20Images-,What%20Is%20the%20Frechet%20Inception%20Distance%3F,by%20Martin%20Heusel%2C%20et%20al. [Accessed: 06-May-2022].

[11]"Trace (linear algebra)," *Wikipedia*, 21-Apr-2022. [Online]. Available: https://en.wikipedia.org/wiki/Trace_(linear_algebra). [Accessed: 06-May-2022].

[12] Aladdinpersson, "Aladdinpersson/machine-learning-collection: A resource for learning about ML, DL, pytorch and tensorflow. feedback always appreciated :)," *GitHub*. [Online]. Available: https://github.com/aladdinpersson/Machine-Learning-Collection. [Accessed: 06-May-2022].

Appendix A

```
from torch.nn.modules.activation import LeakyReLU
class CNNBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride):
        super().__init__()
        self.conv= nn.Sequential(
                nn.Conv2d(in_channels, out_channels, 4, stride, bias=False, padding_mode="reflect"),
                nn.BatchNorm2d(out_channels),
                nn.LeakyReLU(0.2),
        )
    def forward(self,x):
        return self.conv(x)
#x,y <- concatenate these along the channels
class Discriminator(nn.Module):
    def __init__(self, in_channels=3, features=[64, 128, 256, 512]): #256->30x30
            super().__init__()
            self.initial = nn.Sequential(
                    nn.Conv2d(in_channels*2, features[0], kernel_size=4, stride=2, padding=1, padding_mode="reflect"),
                    nn.LeakyReLU(0.2),
            )
            layers = []
            in_channels=features[0]
            for feature in features[1:]:
                layers.append(
                        CNNBlock(in_channels, feature, stride=1 if feature==features[-1] else 2),
                )
                in_channels = feature
            layers.append(
                    nn.Conv2d(in_channels,1,kernel_size=4,stride=1, padding=1, padding_mode="reflect")
            )
            self.model = nn.Sequential(*layers)
    def forward(self, x, y):
        x = torch.cat([x,y], dim=1)
        x = self.initial(x)
        return self.model(x)
```

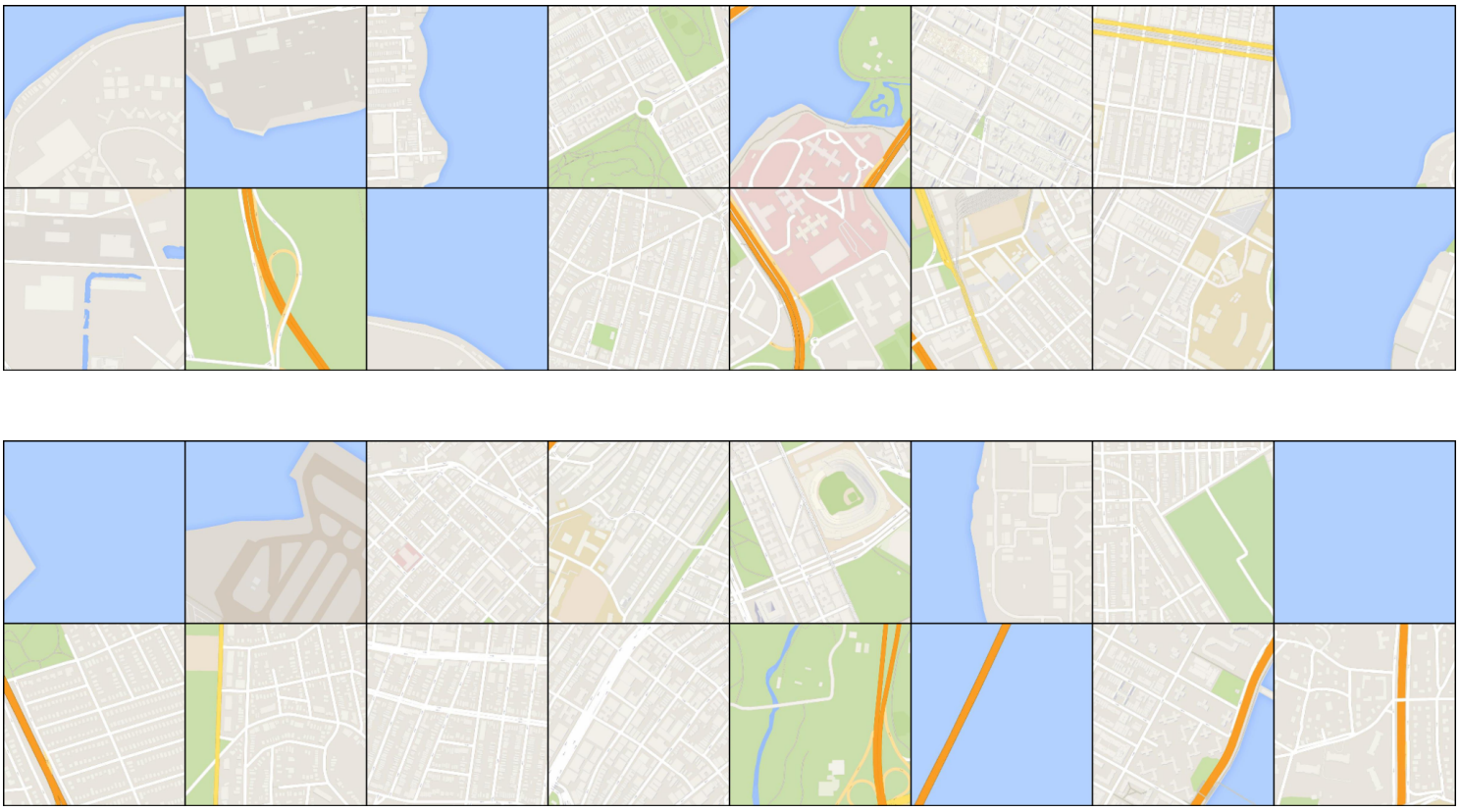
APPENDIX B

```
from torch.nn.modules.batchnorm import BatchNorm2d
class Block(nn.Module):
    def __init__(self, in_channels, out_channels, down=True, act="relu", use_dropout=False):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 4, 2, 1, bias=False, padding_mode="reflect")
            if down
            else nn.ConvTranspose2d(in_channels, out_channels, 4, 2, 1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU() if act=="relu" else nn.LeakyReLU(0.2),
        )
        self.use_dropout = use_dropout
        self.dropout = nn.Dropout(0.5)
    def forward(self, x):
        x = self.conv(x)
        return self.dropout(x) if self.use_dropout else x

class Generator(nn.Module):
    def __init__(self, in_channels=3, features=64):
        super().__init__()
        self.initial_down = nn.Sequential(
            nn.Conv2d(in_channels, features, 4, 2, 1, padding_mode="reflect"),
            nn.LeakyReLU(0.2),
        )#128
        self.down1 = Block(features, features*2, down=True, use_dropout=False)#64
        self.down2 = Block(features*2, features*4, down=True, use_dropout=False)#32
        self.down3 = Block(features*4, features*8, down=True, use_dropout=False)#16
        self.down4 = Block(features*8, features*8, down=True, use_dropout=False)#8
        self.down5 = Block(features*8, features*8, down=True, use_dropout=False)#4
        self.down6 = Block(features*8, features*8, down=True, use_dropout=False)#2
        self.bottleneck = nn.Sequential(
            nn.Conv2d(features*8, features*8, 4, 2, 1, padding_mode="reflect"),
            nn.ReLU(),
        )#1
        self.up1 = Block(features*8, features*8, down=False, act="relu", use_dropout=True)
        self.up2 = Block(features*8*2, features*8, down=False, act="relu", use_dropout=True)
        self.up3 = Block(features*8*2, features*8, down=False, act="relu", use_dropout=True)
        self.up4 = Block(features*8*2, features*8, down=False, act="relu", use_dropout=False)
        self.up5 = Block(features*8*2, features*4, down=False, act="relu", use_dropout=False)
        self.up6 = Block(features*4*2, features*2, down=False, act="relu", use_dropout=False)
        self.up7 = Block(features*2*2, features, down=False, act="relu", use_dropout=False)
        self.final_up = nn.Sequential(
            nn.ConvTranspose2d(features*2, in_channels, 4, 2, 1),
            nn.Tanh(),
        )
    def forward(self,x):
        d1 = self.initial_down(x)
        d2 = self.down1(d1)
        d3 = self.down2(d2)
        d4 = self.down3(d3)
        d5 = self.down4(d4)
        d6 = self.down5(d5)
        d7 = self.down6(d6)
        d8 = self.bottleneck(d7)

        u1 = self.up1(d8)
        u2 = self.up2(torch.cat([u1,d7], dim=1))
        u3 = self.up3(torch.cat([u2,d6], dim=1))
        u4 = self.up4(torch.cat([u3,d5], dim=1))
        u5 = self.up5(torch.cat([u4,d4], dim=1))
        u6 = self.up6(torch.cat([u5,d3], dim=1))
        u7 = self.up7(torch.cat([u6,d2], dim=1))
        return self.final_up(torch.cat([u7,d1], dim=1))
```

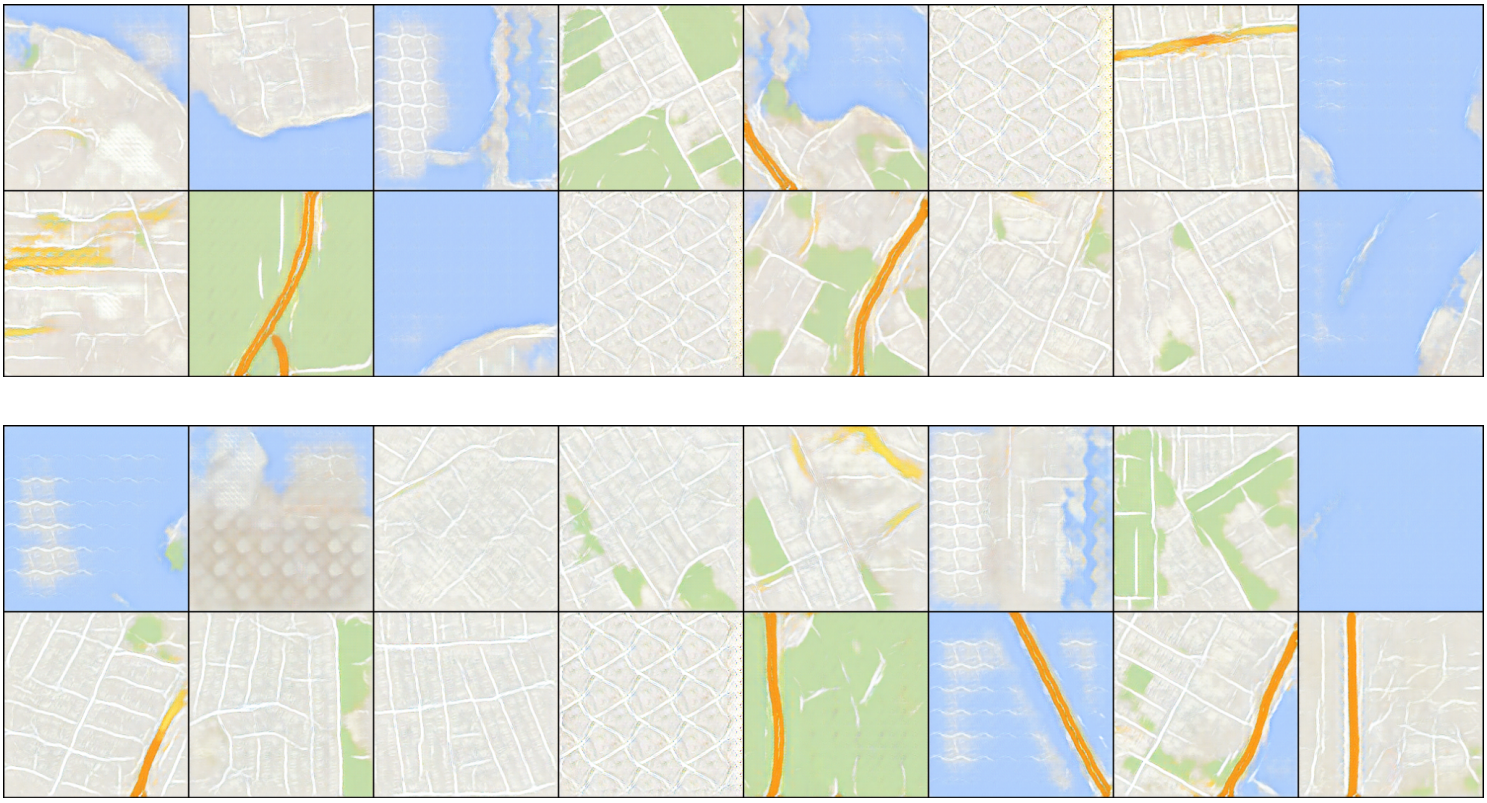
APPENDIX C

INPUT SATELLITE IMAGES (X):

CORRESPONDING OUTPUT GOOGLE IMAGES (Y):



GENERATED GOOGLE IMAGES (FAKE Y):



APPENDIX D

```python
import torch
import torch.nn as nn

class Block(nn.Module):
    def __init__(self, in_channels, out_channels, stride):
        super().__init__()
        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),
        )
    def forward(self, x):
        return self.conv(x)
class Discriminator(nn.Module):
    def __init__(self, in_channels=3, features=[64, 128, 256, 512]):
        super().__init__()
        self.init = nn.Sequential(
                nn.Conv2d(in_channels, features[0], kernel_size=4, stride=2, padding=1, padding_mode="reflect"),
                nn.LeakyReLU(features[0], inplace=True),
        )
        in_channels=features[0]
        layers=[]
        for i in features[1:]:
            layers.append(Block(in_channels, i, stride=1 if i == features[-1] else 2))
            in_channels = i

        layers.append(nn.Conv2d(in_channels, 1, kernel_size=4, stride=1, padding=1, padding_mode="reflect")) #BatchSize x 30 x 30
        self.model = nn.Sequential(*layers)

    def forward(self, x):
        x = self.init(x)
        x = self.model(x)
        return torch.sigmoid(x)
```

APPENDIX E

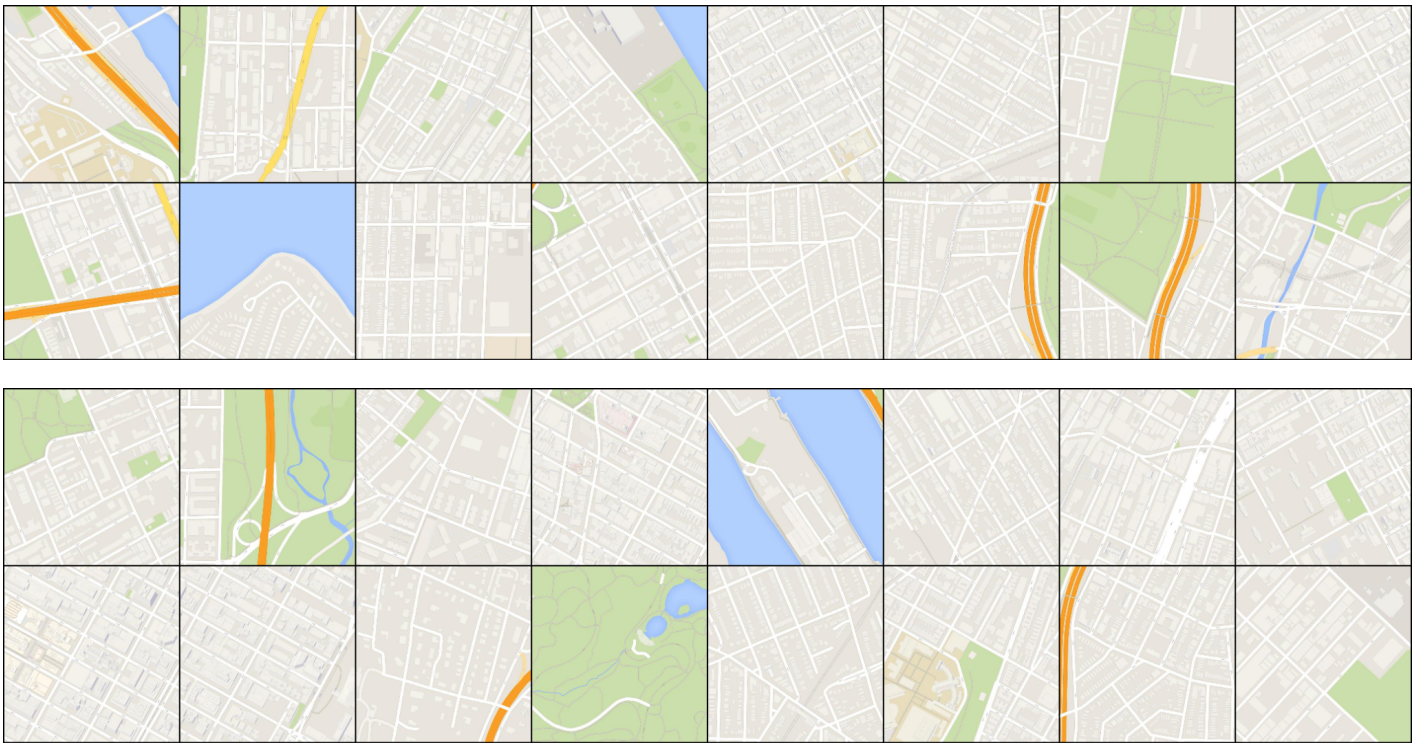
```python
class CNNBlock(nn.Module):
    def __init__(self, in_channels, out_channels, down=True, act=True, **kwargs):
        super().__init__()
        self.conv = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, **kwargs) if down
                else nn.ConvTranspose2d(in_channels, out_channels, **kwargs),
                nn.InstanceNorm2d(out_channels),
                nn.ReLU(inplace=True) if act else nn.Identity(),
        )

    def forward(self, x):
        return self.conv(x)

class Residual(nn.Module):
    def __init__(self, channel):
            super().__init__()
            self.res = nn.Sequential(
                    CNNBlock(channel, channel, kernel_size=3, stride=1, padding=1),
                    CNNBlock(channel, channel, act=False, kernel_size=3, stride=1, padding=1, padding_mode="reflect"),
            )
    def forward(self, x):
        x = x + self.res(x)
        return x
```

```python
class Generator(nn.Module):
    def __init__(self, img_channels, num_res=9):
            super().__init__()
            self.init = nn.Sequential(
                    nn.Conv2d(img_channels, 64, kernel_size=7, stride=1, padding=3, padding_mode="reflect"),
                    nn.InstanceNorm2d(64),
                    nn.ReLU(inplace=True),
            )

            self.down = nn.ModuleList(
                    [
                      CNNBlock(64, 128, kernel_size=3, stride=2, padding=1, padding_mode="reflect"),
                      CNNBlock(128, 256, kernel_size=3, stride=2, padding=1, padding_mode="reflect"),
                    ]
            )

            self.resi = nn.Sequential(
                    *[Residual(256) for _ in range(num_res)]
            )

            self.up = nn.ModuleList(
                    [
                      CNNBlock(256, 128, down=False, kernel_size=3, stride=2, padding=1, output_padding=1),
                      CNNBlock(128, 64, down=False, kernel_size=3, stride=2, padding=1, output_padding=1),
                    ]
            )

            self.last = nn.Conv2d(64, img_channels,kernel_size=7, stride=1, padding=3, padding_mode="reflect")

    def forward(self, x):
        x = self.init(x)
        for down_layer in self.down:
            x = down_layer(x)
        x = self.resi(x)
        for up_layer in self.up:
            x = up_layer(x)
        x = self.last(x)
        return torch.tanh(x)
```

APPENDIX F

INPUT SATELLITE IMAGES (X):

CORRESPONDING OUTPUT GOOGLE IMAGES (Y):



GENERATED GOOGLE IMAGES (FAKE Y):

APPENDIX G

DATA AUGMENTATIONS:

```
both_transform = A.Compose(
        [A.Resize(width=256, height=256),
        A.HorizontalFlip(p=0.5),],
        additional_targets={"image0":"image"},)
transform_only_input = A.Compose(
        [
                A.HorizontalFlip(p=0.5),
                A.ColorJitter(p=0.2),
                A.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5], max_pixel_value=255.0,),
                ToTensorV2(),
        ]
)

transform_only_mask = A.Compose(
        [
                A.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5], max_pixel_value=255.0,),
                ToTensorV2(),
        ]
)
```

OVERALL DATA PROCESSING:

```
from PIL import Image
import numpy as np
import os
from torch.utils.data import Dataset, DataLoader
from torchvision.utils import save_image
import sys
class MapDataset():
    def __init__(self, root_dir):
        self.root_dir = root_dir
        self.list_files =os.listdir(self.root_dir)
        print(self.list_files)
    def __len__(self):
        return(len(self.list_files))
    def __getitem__(self, index):
        img_file = self.list_files[index]
        img_path = os.path.join(self.root_dir, img_file)
        image = np.array(Image.open(img_path))
        input_image = image[:,:600,:] #channel, width, height
        target_image = image[:,600:,:]
        augmentations = both_transform(image=input_image, image0=target_image)
        input_image, target_image=augmentations["image"], augmentations["image0"]
        input_image = transform_only_input(image=input_image)["image"]
        target_image = transform_only_mask(image=target_image)["image"]
        return input_image, target_image
```

APPENDIX H

```
LEARNING_RATE = 2e-4
BATCH_SIZE = 16
NUM_WORKERS = 1
IMAGE_SIZE = 256
CHANNELS_IMG = 3
L1_LAMBDA = 100
NUM_EPOCHS = 500
LOAD_MODEL = False
SAVE_MODEL = True
CHECKPOINT_DISC = "disc.pth.tar"
CHECKPOINT_GEN = "gen.pth.tar"
```

```
import torch.optim as optim
from tqdm import tqdm

def train_fn(disc, gen, loader, opt_disc, opt_gen, l1_loss, bce, g_scaler, d_scaler):
    loop = tqdm(loader, leave=True)

    for idx, (x, y) in enumerate(loop):
        x = x.to(DEVICE)
        y = y.to(DEVICE)

        #Train Discriminator
        with torch.cuda.amp.autocast():
            y_fake = gen(x)
            D_real = disc(x, y)
            D_real_loss = bce(D_real, torch.ones_like(D_real))
            D_fake = disc(x, y_fake.detach())
            D_fake_loss = bce(D_fake, torch.zeros_like(D_fake))
            D_loss = (D_real_loss + D_fake_loss)/2
        disc.zero_grad()
        d_scaler.scale(D_loss).backward()
        d_scaler.step(opt_disc)
        d_scaler.update()

        #Train Generator
        with torch.cuda.amp.autocast():
            D_fake=disc(x, y_fake)
            G_fake_loss = bce(D_fake, torch.ones_like(D_fake))
            L1 = l1_loss(y_fake, y)*L1_LAMBDA
            G_loss = G_fake_loss + L1
        gen.zero_grad()
        g_scaler.scale(G_loss).backward()
        g_scaler.step(opt_gen)
        g_scaler.update()
```

```python
def train():
    disc = Discriminator(in_channels=3).to(DEVICE)
    gen = Generator(in_channels=3, features=64).to(DEVICE)
    opt_disc = optim.Adam(disc.parameters(), lr=LEARNING_RATE, betas=(0.5,0.999),)
    opt_gen = optim.Adam(gen.parameters(), lr=LEARNING_RATE, betas=(0.5,0.999))
    BCE = nn.BCEWithLogitsLoss()
    L1_LOSS = nn.L1Loss()

    if LOAD_MODEL:
        load_checkpoint(
                CHECKPOINT_GEN, gen, opt_gen, LEARNING_RATE
        )

        load_checkpoint(
                CHECKPOINT_DISC, disc, opt_disc, LEARNING_RATE
        )
    train_dataset = MapDataset(root_dir=TRAIN_DIR)
    train_loader = DataLoader(
            train_dataset,
            batch_size=BATCH_SIZE,
            shuffle=True,
            num_workers=NUM_WORKERS,
    )
    g_scaler = torch.cuda.amp.GradScaler()
    d_scaler = torch.cuda.amp.GradScaler()
    val_dataset = MapDataset(root_dir=VAL_DIR)
    val_loader = DataLoader(val_dataset, batch_size=1, shuffle=False)
```

```python
    for epoch in range(NUM_EPOCHS):
        train_fn(
                disc, gen, train_loader, opt_disc, opt_gen, L1_LOSS, BCE, g_scaler, d_scaler,
        )
        if SAVE_MODEL and epoch % 5 == 0:
            save_checkpoint(gen, opt_gen, filename=CHECKPOINT_GEN)
            save_checkpoint(disc, opt_disc, filename=CHECKPOINT_DISC)

        # save_some_examples(gen, val_loader, epoch, folder="../content/drive/MyDrive/AI/maps/evaluation")


if __name__=="__main__":
    train()
```

APPENDIX I

```
BATCH_SIZE = 16
LEARNING_RATE = 2e-4
LAMBDA_IDENTITY = 0.0
LAMBDA_CYCLE = 10
NUM_WORKERS = 4

L1_LAMBDA = 100
NUM_EPOCHS = 500
LOAD_MODEL = False
SAVE_MODEL = True
LAMBDA_IDENTITY = 0.0
LAMBDA_CYCLE = 10
CHECKPOINT_DISC = "disc.pth.tar"
```

```
import torch.optim as optim
from tqdm import tqdm
import torch
torch.cuda.empty_cache()

def train_fn(dis_X, dis_Y, gen_X, gen_Y, opt_dis, opt_gen, L1_LOSS, MSE_LOSS, train_loader, g_scaler, d_scaler):
    loop = tqdm(train_loader, leave=True)

    for idx, (x, y) in enumerate(loop):
        x = x.to(DEVICE)
        y = y.to(DEVICE)


        #train discriminator of x and y

        with torch.cuda.amp.autocast():
            fake_y = gen_Y(x)

            D_Y_real = dis_Y(y)
            D_Y_fake = dis_Y(fake_y.detach())

            D_Y_real_loss = MSE_LOSS(D_Y_real, torch.ones_like(D_Y_real))
            D_Y_fake_loss = MSE_LOSS(D_Y_fake, torch.zeros_like(D_Y_fake))

            D_Y_loss = D_Y_real_loss + D_Y_fake_loss

            fake_x = gen_X(y)

            D_X_real = dis_X(x)
            D_X_fake = dis_X(fake_x.detach())

            D_X_real_loss = MSE_LOSS(D_X_real, torch.ones_like(D_X_real))
            D_X_fake_loss = MSE_LOSS(D_X_fake, torch.zeros_like(D_X_fake))

            D_X_loss = D_X_real_loss + D_X_fake_loss

            D_loss = (D_Y_loss + D_X_loss)/2

        opt_dis.zero_grad()
        d_scaler.scale(D_loss).backward()
        d_scaler.step(opt_dis)
        d_scaler.update()
```

```
        #Generator Loss
        with torch.cuda.amp.autocast():
            D_Y_fake = dis_Y(fake_y)
            D_X_fake = dis_X(fake_x)

            loss_G_X = MSE_LOSS(D_X_fake, torch.ones_like(D_X_fake))
            loss_G_Y = MSE_LOSS(D_Y_fake, torch.ones_like(D_Y_fake))


            #cycle loss
            cycle_x = gen_X(fake_y)
            cycle_y = gen_Y(fake_x)

            CYCLE_X_loss = L1_LOSS(x, cycle_x)
            CYCLE_Y_loss = L1_LOSS(y, cycle_y)



            G_loss = (loss_G_X + loss_G_Y + CYCLE_X_loss * LAMBDA_CYCLE + CYCLE_Y_loss * LAMBDA_CYCLE)

        opt_gen.zero_grad()
        g_scaler.scale(G_loss).backward()
        g_scaler.step(opt_gen)
        g_scaler.update()

        if idx % 50 == 0:
            save_image(fake_y*0.5+0.5, f"../content/drive/MyDrive/AI/maps/cycleganeval/fake_y_{idx}.png")
            save_image(y*0.5+0.5, f"../content/drive/MyDrive/AI/maps/cycleganeval/real_y_{idx}.png")
            save_image(x*0.5+0.5, f"../content/drive/MyDrive/AI/maps/cycleganeval/x_{idx}.png")
```

```
def train():
    dis_X = Discriminator(in_channels=3).to(DEVICE)
    dis_Y = Discriminator(in_channels=3).to(DEVICE)

    gen_X = Generator(img_channels=3, num_res=9).to(DEVICE)
    gen_Y = Generator(img_channels=3, num_res=9).to(DEVICE)

    opt_dis = optim.Adam(
              list(dis_X.parameters()) + list(dis_Y.parameters()),
              lr=LEARNING_RATE,
              betas=(0.5, 0.999),
        )

    opt_gen = optim.Adam(
              list(gen_X.parameters()) + list(gen_Y.parameters()),
              lr=LEARNING_RATE,
              betas=(0.5, 0.999),
        )
    L1_LOSS = nn.L1Loss()
    MSE_LOSS = nn.MSELoss()

    train_dataset = MapDataset(root_dir=TRAIN_DIR)
    train_loader = DataLoader(
           train_dataset,
           batch_size=BATCH_SIZE,
           shuffle=True,
           num_workers=NUM_WORKERS,
    )

    g_scaler = torch.cuda.amp.GradScaler()
    d_scaler = torch.cuda.amp.GradScaler()
    val_dataset = MapDataset(root_dir=VAL_DIR)
    val_loader = DataLoader(val_dataset, batch_size=1, shuffle=False)


    for epoch in range(NUM_EPOCHS):
        train_fn(dis_X, dis_Y, gen_X, gen_Y, opt_dis, opt_gen, L1_LOSS, MSE_LOSS, train_loader, g_scaler, d_scaler)

if __name__ == "__main__":
    train()
```

APPENDIX J

```
import numpy
from numpy import cov
from numpy import trace
from numpy import iscomplexobj
from numpy import asarray
from numpy.random import shuffle
from scipy.linalg import sqrtm
from keras.applications.inception_v3 import InceptionV3
from keras.applications.inception_v3 import preprocess_input
from skimage.transform import resize
def scale_images(images, new_shape):
    images_list = list()
    for image in images:
        # resize with nearest neighbor interpolation
        new_image = resize(image, new_shape, 0)
        # store
        images_list.append(new_image)
    return asarray(images_list)

# calculate frechet inception distance
def calculate_fid(model, images1, images2):
    # calculate activations
    act1 = model.predict(images1)
    act2 = model.predict(images2)
    # calculate mean and covariance statistics
    mu1, sigma1 = act1.mean(axis=0), cov(act1, rowvar=False)
    mu2, sigma2 = act2.mean(axis=0), cov(act2, rowvar=False)
    # calculate sum squared difference between means
    ssdiff = numpy.sum((mu1 - mu2)**2.0)
    # calculate sqrt of product between cov
    covmean = sqrtm(sigma1.dot(sigma2))
    # check and correct imaginary numbers from sqrt
    if iscomplexobj(covmean):
        covmean = covmean.real
    # calculate score
    fid = ssdiff + trace(sigma1 + sigma2 - 2.0 * covmean)
    return fid
path_fake_y = "../content/drive/MyDrive/AI/maps/cycleganeval/fake_y_0.png"
path_real_y = "../content/drive/MyDrive/AI/maps/cycleganeval/real_y_0.png"
# prepare the inception v3 model
model = InceptionV3(include_top=False, pooling='avg', input_shape=(299,299,3))
images1 = np.array(Image.open(path_fake_y)).astype('float32')
images2 = np.array(Image.open(path_real_y)).astype('float32')
# resize images
images1 = scale_images(images1, (299,299,3))
images2 = scale_images(images2, (299,299,3))
# pre-process images
images1 = preprocess_input(images1)
images2 = preprocess_input(images2)
# calculate fid
fid = calculate_fid(model, images1, images2)

print('FID: %.3f' % fid)
```