Cardiff University

Computer Science an Informatics

CMT400 – Dissertation

Image Manipulation with Deep Generative Adversarial Networks

Author: Rongjian Huang
Supervisor: Dr. Yukun Lai
Moderator: Dr. Xianfang Sun

Abstract

Digital images are commonly used in every field nowadays. However, sometimes we want to make some changes to it, such as deleting or moving the object inside the image, adding a new object into the image, or sometimes simply wanting a new image with similar contents. Motivated by this requirement, I implement an unconditional generative model trained on a single natural image based on the paper of SinGAN. After a multi-stage training, this model is able to capture the inner distribution of the training image. As a result, it can generate realistic and highly diverse random images. This model can also be applied to many image manipulation tasks, including image editing, image harmonization, and paint to image translation. Qualitative analysis has been done in order to find out the best hyperparameters. And I also discuss the limitations and future extensions of this model in this paper.

Table of Contents

Table of Figures

1. Introduction

Deep Neural Network (DNN) has been highly conspicuous since 2010, although this idea was first held in the last century. Equipped with nowadays more powerful GPU and advanced distributed computing algorithms, DNN broke through the limitation of the lack of computational power and resources. In the field of Computer Vision (CV), Convolutional Neural Network (CNN) [1] has achieved great success in many tasks, including image classification [2], auto-driving [3], cancer detection [4], and semantic segmentation [5]. However, these models mentioned above are mostly discriminative models rather than generative models. Take cancer detection. Considering the scenario of cancer detection, given an x-ray photo from the patient, the model needs to determine whether the patient has cancer, and if so, what is the type of cancer. This is a classification problem. Why have discriminative models achieve great success, but there is seldom progress in generative models? Generally speaking, it is easier to identify Monet's painting rather than imitate Monet's painting. This idea is also applicable in neural network models. The generative model that generates data is doing a more difficult job than the discriminative model that handles data.

This situation continues until Ian Goodfellow proposed the Generative Adversarial Network (GAN) [6] in 2014, which ignited the flame of research on generative models. Before GAN, many generative models were lack of expressiveness in generating images, such as Variational Auto Encoder (VAE) [7]. However, GAN solved the problem. It can generate images with sufficient details. GAN is a composition of two individual networks, a discriminator and a generator. The discriminator network is like an ordinary CNN binary classifier that determines whether the image is a real or generated image. If the image is a real image, the discriminator will give it a higher score and vice versa. And generator is to generate some realistic images from a random vector (in most cases, Gaussian or uniform noise) and try to make the distribution of the generated images consistent with the real images' distribution. In other words, the generator needs to learn the features from the real images and generate images with such features that can fool the discriminator. In GAN's architecture, the generator network is more important. The discriminator network's role is to be an adaptive loss function to guide the training of the generator [8]. GAN can learn and use more accurate loss function through adversarial learning, which encourages the network to produce high-quality images. In the past few years, we have witnessed that the research on GANs has made great progress in generating high-quality and realistic images. This achievement is also widely applied in various fields, such as image

manipulation and image synthesis. For example, image to image translation [9], style transfer [39], super-resolution [11], image harmonization [12], and image coloring [13]. However, these models usually use a dataset of class-specific images to train, bringing it two disadvantages. First, sometimes it is hard to look for this kind of dataset. For example, only a few datasets can be used for semantic segmentation task [14]. Second, a large amount of training data needs to consume a lot of computational power and resources.

On the contrary, GAN trained on a single image can perform well in image manipulation tasks. Many small patches in the image tend to be similar or even duplicate so that the model can learn and capture them easily [15]. Furthermore, each image has some unique features like little object or texture in the image, and only the model trained on this image can capture them [16]. The model trained through a large amount of training data can capture the general features, but it may not capture these unique features in the image. So I think that single image training is an excellent direction for image manipulation.

Based on the work from the previous single-image generation models (INGAN [15], SINGAN [17], CoSinGAN [18], ZSSR [16]), I build an GAN model that is trained on single natural images and generated realistic images that can semantically resemble the training images. And my model can be applied to many tasks, including generating random images, image harmonization, paint to image translation, and image editing through single image training. Then I test the results through qualitative analysis.

2. Related Work

2.1. Generative model

In general, the generative model's goal is, given a set of training data and the generative model can learn the distribution of training data and generate a sample with the same distribution. We assume that training data came from distribution A and generated data comes from distribution B. The job of the generative model is to minimize the distance between this two distribution. There have been many promising generative models in the past ten years such as the autoregressive model [21,22], Variational Auto-Encoder (VAE) [7], flow-based model [20], and also Generative Adversarial Network (GAN) [6], each of which has its advantages and disadvantages.

For autoregressive models like pixel CNN [21] and pixel RNN [22], they are widely used in other fields like predicting economics, informatics, and natural phenomena. Applied in the generative field, they can produce high-quality images. However, they can be time-consuming because they are hard to cooperate with the distributed paradigm.

A better model is Variational Auto-Encoder (VAE). VAE is a generative model proposed by Diederik P. Kingma and Max Welling in 2013. It is now a primary tool for inference in many complex scenarios. It is an updated version of auto-encoder (AE) [23], so they have a similar structure - a composition of encoder and decoder. The encoder is responsible for mapping the image to a latent vector in the latent space, while the decoder is responsible for mapping the latent vector back to the image and limit the restored image as similar as possible to the original image. However, what made VAE different from AE is that VAE has a restriction on the encoder that forces the generated latent vector to roughly respect the standard normal distribution. So while we want to generate a new image, we only need to input a latent vector that respects the standard normal distribution. VAE can generate a more realistic image than AE and also is way faster than autoregressive models. While the weakness of VAE is that VAE needs to use MSE to tell the difference between the generated image and training image to guide the training, which leads to generating blurry images. Lots of work have been done to improve the VAE's ability. Currently, the SOTA is VQ-VAE-2 [24], which can generate realistic images comparable to BigGAN's results.

There are other models that can be comparable to BigGAN [25]. Open AI proposed GLOW [26] in 2018, and it generated impressive results. While training on a human face dataset, it can generate high-resolution human faces that can hardly be distinguished by a human. Moreover, it can manipulate the human faces' attribute, changing the generated

image from happy to sad, from long hair to short hair, female to male, etc. GLOW model is a flow-based model. The flow-based model's main idea is to look for a reversible encoder that maps original data to a latent vector in the latent space. Furthermore, because it is reversible, so once the encoder is trained, we can immediately get the decoder by calculating the inverse of the encoder. However, training a flow-based model is costly. GLOW took a week and 40 GPUs to train to generate 256x256 images. So currently, it is not suitable to use as a general solution to image manipulation tasks.

2.2. Generative Adversarial Network

Ian Goodfellow proposed GAN in 2014. It is a training framework composed of two individual networks, discriminator and generator. Here is a classic example from Ian Goodfellow [6]: The generator can be regarded as a criminal group who wants to make counterfeit money, while the discriminator can be regarded as the policemen team who wants to identify counterfeit money. Both the criminal group and the policemen team train their skills by the feedbacks from the opponent. These competitions end until the criminal group is able to generate counterfeit money that is indistinguishable from policemen.

Training GAN can be represented by this formula:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

A generator tries to learn the data distribution of training data and generates random samples. A discriminator is a binary classifier that classifies whether the given data is from the real distribution or generator. If it is a real image, the discriminator will give it a higher score, and vice versa. Discriminator plays the role of an adaptive loss function in the GAN framework.

GAN can learn data distribution and generate sharp images. However, training GAN can sometimes be challenging. First of all, GAN is known to be very unstable to train. Because under the optimal discriminator, minimizing the generator loss is equivalent to minimizing the Jensen-Shannon divergence (JS-divergence) between training data distribution and generator distribution. However, the generator distribution is supported by low dimensional manifolds. It is unlikely that two distribution overlap at some point [27]. So the JS-divergence will be approximately log2, and the gradient of the generator is always approximately 0, which is gradient vanishing. Also, mode collapse frequently

happens in GAN. The generator tends to generate the repeated but correct samples rather than generate highly diverse but may incorrect samples.

2.3. Improvement of GAN's structure

To address GAN's problem, researchers have done a lot of brilliant jobs to improve GAN's performance. These researches are divided into several categories. Some researchers improve the model structure of GAN.

For example, Alec Radford & Luke Metz proposed DCGAN [28] in 2016. They integrate CNN into GAN and use CNN's powerful feature extraction capabilities to strengthen GAN's learning ability. They replaced the pooling layer with a stridden convolutional layer to keep the spatial information in images, use Batch Normalization to stabilize training, replace a fully connected layer with a convolutional layer to make it a fully convolutional network. In general, they increase the stability of the model and the quality of the generated image. However, the problem of mode collapse still cannot be solved.

What is more, Progressive GAN (PGGAN) [8] was proposed by NVIDIA LAB in 2018. In this paper, the authors put forward a coarse-to-fine structure. Here is the idea. If a model wants to generate a high-resolution image (e.g., 1024x1024), it is challenging to transfer a low-dimensional random noise to a high-dimensional data. So we start generation from a low-resolution image and gradually add layers to handle a higher resolution image. The benefit of this structure is that the model learns to generate a general framework of the images in the lower layer, and the model learns to generate some finer details during the process of adding layers on the model. This structure guarantees a stabler and more efficient way to generate images.

Similarly, Lapgan [29], proposed by Emily Denton in 2015, also used the coarse-to-fine structure. What is different is Lagan did not adopt the method of gradually adding layers to the network. Instead, the authors create and train a Laplacian pyramid structure of discriminator and generator. The generator at the bottom of the pyramid model generates a low-resolution image from random noise and delivers the upsampled generated image to the higher lever generator as a prior. Then the higher level generator uses this prior and another noise to generate a higher resolution image.

2.4. Improvement on loss function

Other researchers have proposed some improvement on loss functions. For instance, in image-to-image translation task, it require model to learn a mapping function G : X → Y which map image from one domain to another domain. In Cycle GAN [14], there are two mapping, G : X→Y and F: Y→X.

Junyan Zhu introduced two cycle consistency losses, forward cycle-consistency loss: x → G(x) → F (G(x)) ≈ x and backward cycle-consistency loss: y → F (y) → G(F (y)) ≈ y, which force the model to learn not only the mapping from one domain to the other, but also the mapping back to the original domain. As a result, It can produce impressive Van Gogh's style image by inputing any natural image in an unsupervised setting.

The proposal of WGAN [27] and WGANGP [30] is a milestone in improving the loss function of GAN. The author of WGAN, Martin Arjovsky, suggested the EM distance (Earth-Mover distance) as a substitution of JS-divergence in original GAN.

$$W(\mathbb{P}_r, \mathbb{P}_g) = \inf_{\gamma \in \Pi(\mathbb{P}_r, \mathbb{P}_g)} \mathbb{E}_{(x,y)\sim\gamma}\big[\, \|x - y\| \,\big]$$

The concept of EM distance is very intuitive, let's consider distribution p_{data} and $P_{generated}$ as two piles of sand. What EM distance do is to measure the minimum cost of pushing the sand of p_{data} to the position of the sand of $P_{generated}$. Compared to JS-divergence, EM distance can represent the distance between two distribution even if they do not have significant overlap, which can provide a more substantial and meaningful gradient for both generator and discriminator. However, EM distance cannot be directly applied to GAN because the infimum in the formula cannot be solved. Instead, author transformed the formula to the formula below formula according to the Kantorovich-Rubinstein duality .

$$W(\mathbb{P}_r, \mathbb{P}_\theta) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x\sim\mathbb{P}_r}[f(x)] - \mathbb{E}_{x\sim\mathbb{P}_\theta}[f(x)]$$

This formula bring us a new restriction, it required the discriminator to satisfy 1-Lipschitz constraint and in WGAN it is achieved by weight clipping. The formula above is

complicated but the implementation is easy, the author made four changes on the original GAN.

1. Removing the sigmoid activation function in the last layer of discriminator
2. Removing the logarithm function in both discriminator's and generator's loss function.
3. Clipping the weight of discriminator to stay between [-c, c] where c is a fix constant.
4. Do not use optimizer that based on momentum algorithm such as Adam.

As a result, WGAN has achieved great success in addressing the problem of mode collapse and proposing a tool that can accurately measure the quality of the generated images. However, some scholars had found that weight clipping would cause the model to not converge. Because weight clipping turns most of the weight in the discriminator to be -c or c where c is a constant, the discriminator will only learn a simple mapping function resulting in the optimization difficulties. Soon the author introduced gradient penalty as a substitution of weight clipping in WGAN-GP.

$$\lambda \underset{\hat{\boldsymbol{x}} \sim \mathbb{P}_{\hat{\boldsymbol{x}}}}{\mathbb{E}} \left[(\|\nabla_{\hat{\boldsymbol{x}}} D(\hat{\boldsymbol{x}})\|_2 - 1)^2 \right].$$

This term penalizes the gradient norm of the discriminator output and forces the discriminator's gradient norm to be close to 1. Consequently, the discriminator learns a smoother decision boundary, providing the generator meaningful gradient to go on learning. WGAN-GP completely solved the problem of GAN's training instability.

2.5. Training tricks

Some scholars suggest using some tricks to help stabilize the learning. For example, Kaiming He suggests using "he initialization" in the neural network to accelerate the learning [31]. A suitable initialization method can not only solve the problem of saturated neurons but can also accelerate the convergence of the model.

TTUR [32] is a training strategy proposed by Martin Heusel in 2017. The approach assigns a larger learning rate to the discriminator and a lower learning rate to the generator. In this way, the ability of the discriminator will be stronger than the ability of the generator. As a consequence, discriminator can converge faster, which can guide the training of the generator.

However, there are many kinds of initialization methods training strategies to train GAN, and sometimes we do not know which one is more suitable for our task. Bad initialization and training strategy will lead to training difficulties. Sergey Ioffe proposed a Batch Normalization [33] method that reduces the model's dependency on initialization. During the training process, network parameters will be continuously updated through backpropagation. Weight changes in the underlying network will cause the input distribution changes in every layer of the network, and the upper network needs to adapt to these changing distributions continually. This problem is called Internal Covariate Shift (ICS). ICS will lead to slower convergence of the network. Furthermore, it scales down the network's learning ability because a part of the network needs to be used to deal with the ICS problem. The BN's idea is to normalize the input so that the mean and the variances of the distribution are 0 and 1 for all features. With BN, the model is less sensitive to the parameters like learning rate and initialization and more stable. On the other hand, BN allows input distribution for all features to be relatively stable, which speeds up the model convergence.

2.6. Image manipulation

Scholars have been researching image manipulation tasks for a long time. Examples of image manipulation include super-resolution [34], image retargeting [36], image harmonization [37], and denoising [38]. Some researches are based on CNN. For example, Justin Johnson proposed Perceptual Losses for Real-Time Style Transfer and Super-Resolution [10] in 2016. This model combined the feedforward transformation network with the high-level perceptual features extracted by the pre-trained VGG-16 network instead of using per-pixel losses. As a result, this model produces impressive results in both the Super-Resolution task and Style Transfer task. While some researches are based on GAN. For example, Junyan Zhu proposed pix2pix [39] in 2016, where a conditional GAN is designed in order to handle the Image-to-Image translation task. The authors suggest a U-net structure in this paper, as skip connection in U-net is able to share some underlying features between input and output. Also, the authors combine the adversarial loss with L1 loss. The idea is to model high-frequencies by adversarial loss and leave the job of low-frequencies for L1 loss. The loss function makes the model available to handle a wide arrange of image translation tasks.

2.7. Single image generation

InGAN [15] was the first GAN model trained on a single natural image. It can precisely capture the internal statistic of the image patches by using a multi-scale patch discriminator. This property enables InGAN to reconstruct the image in any size, ratio, and even in non-rectangle shapes like a triangle with the same internal distribution of patches.

However, this model is only suitable for the retargeting task. After that, Tamar Rott Shaham proposed SinGAN [17], which is also trained on a single natural image. Similarly, SinGAN also applied a multi-scale discriminator in the model. What is different is that SinGAN is an unconditional image generation model that generates images purely from noise, while InGAN maps an input image to a retargeted image. SinGAN adopts the concept of progressive learning in the model in order to generate higher quality images. Moreover, SinGAN can be performed on plenty of computer vision tasks, including paint to image translation, image editing, image harmonization, super-resolution, and animation. SinGAN's success has aroused scholars' confidence in single-image training.

Soon after SinGAN's proposal, CoSinGAN [18] was proposed by Tobias Hinz to improve the performance of SinGAN. They point out two problems of SinGAN. First, training only one stage at a time is ineffective. Second, propagating the images to the next stage generator can lead to training difficulties. As a solution to the first problem, CoSinGAN trained a part of the stages simultaneously. For the second problem, instead of propagating the generated image to the next stage, CoSinGAN propagates the image features to the next stage. As a result, CoSinGAN reduces the training time and generates a comparable result to SinGAN.

2.8. Technologies

Python

In recent years, Python has been very popular because of its ease of use and easy to learn. It supports almost all mainstream deep learning frameworks, so it became the first choice for more and more data scientists and artificial intelligence researchers. Moreover, Python has many mature tool libraries that set you free from recreating wheels. So I choose Python as my programming language.

Numpy

Numpy is a python library for processing scientific computing. It is the basis of many libraries like pandas. It can perform matrix operation and array operation much faster than

Python. So basically, all deep learning algorithms or frameworks use Numpy as the way to handle data.

Tensorflow 2.0

TF2.0 is the most famous deep learning framework. It encapsulates the complex data operations at the bottom layer and provides low-level APIs to allow users to construct their deep learning models flexibly. It has many advantages such as cross-platform, fast inferencing, etc. which make it very commonly used in the industry.

TF.Keras

TF.Keras is a high-level deep learning API. It encapsulates all the components needed to build the model into a highly-abstract API, and users only need to consider the structure of the model when building the model without considering the underlying implementation. TF.Keras can build a neural network model with just a few lines of code.

Pil

Pil is a python image processing library. It has very easy-to-use picture processing functions, including loading and saving pictures, converting picture formats, picture cropping, and changing picture size. These functions can meet our basic needs in computer vision tasks.

Skimage

Skimage is also a python image processing library. But it has some high-level functions such as topology processing and filter operations, which can meet the needs of more complex tasks.

3. Methodology

In this project, my first goal is to build a GAN model that is trained on single natural images and is able to generate realistic and highly diverse images that can semantically resemble the training images. Moreover, I want this model to perform on multiple image manipulation tasks including image editing, image harmonization, and paint to image translation. This project is mostly based on the paper of SinGAN[17], I reimplemented the model of SinGAN in Tensorflow and made some changes.

3.1. Overall architecture

This model consists of multiple stages and each stage is trained individually. At each stage, I train a discriminator network and a generator. The training of the model follows a coarse-to-fine manner. The lower stage generator generates a low-resolution image and passes to the higher stage generator as a prior, and the higher stage generator generate a higher-resolution image according to the prior. The training starts at the bottom stage of the model.

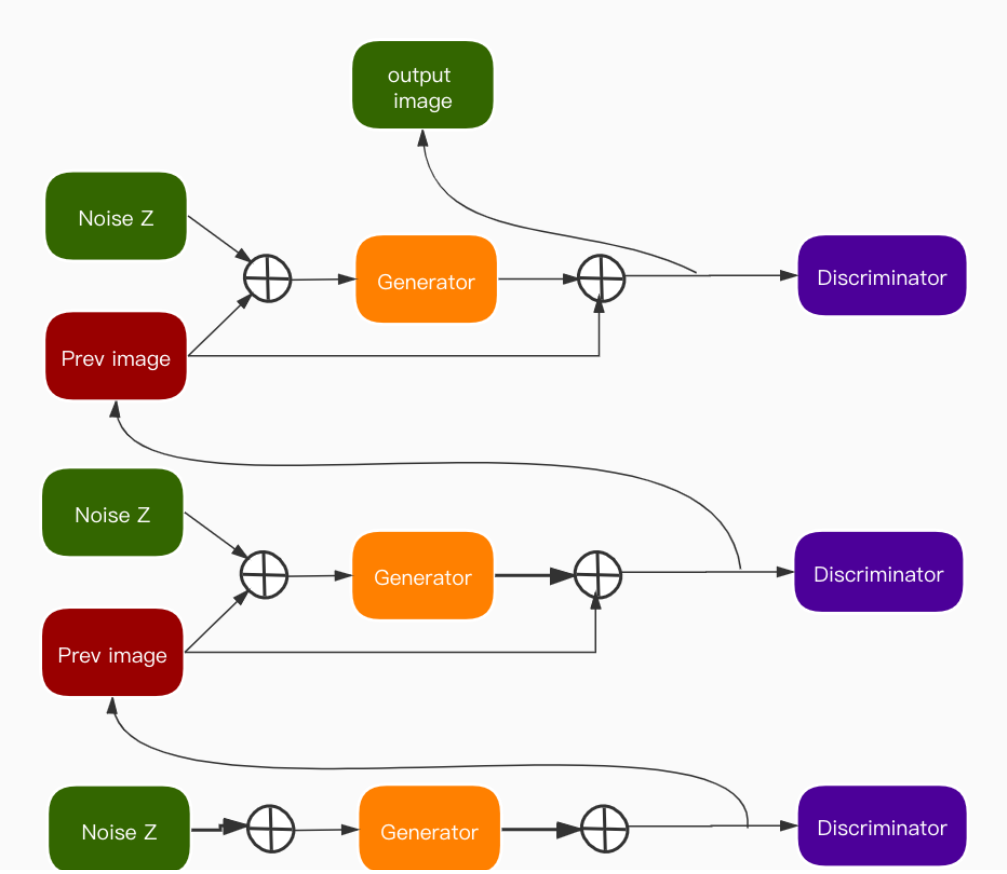

Figure 1: Model Structure

The training can be seen as optimizing the following formula where adversarial loss is WGAN-GP loss, and the reconstruction loss is the Mean Square Error (MSE) between the reconstruction image and downsampled real image at stage n. The BETA (10.0 in default) is the coefficient that controls the ratio of reconstruction loss and adversarial loss. The training can be seen as optimizing the following formula.

$$\min_{G_n} \max_{D_n} L_{adv}(G_n, D_n) + BETA * L_{rec}$$

3.1.1. Adversarial Loss

WGAN-GP loss is used in this model. It can accurately measure the distance between the generated distribution and real distribution. The gradient penalty ensures the smoothness of the discriminator's decision boundary so that the generator can always receive meaningful gradients from the feedback of the discriminator. In this model, the adversarial loss is used to minimize the distance between data distribution and generated distribution. The adversarial loss can be represented by the following formula.

$$L_{adv} = E_{x \sim P_{data}(X)} [D(X)] - E_{z \sim P_z(z)} [D(G(z))] - \lambda E_{\hat{x} \sim P_{\hat{x}}} [(||\nabla_{\hat{x}} D(\hat{x})||_2 - 1)^2]$$

3.1.2. Reconstruction Loss

It is proved to be beneficial to combine the GAN loss function with a traditional loss function like MSE [39]. So that the generated image can be more similar to the training image in an L2 sense. The reconstruction loss can be represented by the following formula.

$$L_{rec} = MSE(G_n((x_{n-1}^{rec}) \uparrow_{upsample}, 0), X_n^{real}), \text{ for n>0}$$

For the case n=0 we have:

$$L_{rec} = MSE(G_n(0, z_{rec}), X_n^{real})$$

3.2. Progressive learning

Progressive learning is a training framework that allows the model to generate images in coarse-to-fine criteria. Progressive learning can help to improve the stability and effectiveness of the generative network. This is an appealing property for GAN, so scholars proposed all kinds of implementations. There are two most well-known implementations, LapGAN and PGGAN. LapGAN trains a set of discriminator networks and generator networks in a pyramid structure; PGGAN train only one discriminator and one generator but gradually add layers on top of them.

In my model, I use the structure that is similar to LapGAN. I trained a set of discriminator networks and generator networks in multiple stages, and the generator and discriminator at each stage of the pyramid are trained individually. A set of downsampled training images were created by a downscaling factor.

$$down_n = real * s^{N-1-n}$$

Here $down_n$ indicate the downsample version at n stage, s is the downscaling factor, and N is the number of stage in this pyramid. Generators are trained on these downsampled training images.

The training starts from the bottom stage (state 0), where the generator is purely unconditional. It receives a random white Gaussian noise as the input and generates an image sample.

$$x_0 = G_0(0,\ z_0)$$

Once the training is finished at this stage, I upsample the generated image and deliver it to the generator in the finer stage as a prior. Since the upsample images only capture the general structure of the image, so the generator at the finer stage is responsible for learning to produce the missing finer details in the images. Here I add another noise to the generator so that it can imitate the details of the image and add some variety to the generated image.

$$x_n = G_n(\,(x_{n-1}) \uparrow_{upsample}, Z_n), \quad for\ n > 0$$

3.3. Residual learning

Kaiming He proposes residual learning in Resnet [40]. It has shown excellent results in speeding up the deep neural network, and it has become the standard structure of recent deep neural networks. Motivated by the degradation problem, he pointed out that directly learning the desired underlying function can be difficult, but learning a residual function can lead to a quicker convergence and accuracy gain.

I also applied residual learning in this project. The generator received an upsampled generated image and a white Gaussian noise as the inputs and output the residual image as additional details to the previous generator's upsampled image. So the generated image can be represented by the following formula.

$$x_n = G_n(\ (x_{n-1}) \uparrow_{upsample}, Z_n) = (x_{n-1}) \uparrow_{upsample} + X_n(\ (x_{n-1}) \uparrow_{upsample} + Z_n)$$

3.4. Markovian Discriminator (Patch-GAN)

Instead of using the normal discriminator, Markovian Discriminator was used in this project. Do not be freaked out by the name. Markovian Discriminator is just a regular convolutional network. The only difference between a Markovian Discriminator and a normal GAN discriminator is that Markovian Discriminator output an NxN matrix of scalar indicating each patch in the image is real or fake, while the typical GAN discriminator output only one scalar indicating the whole image is real or not.

In this project, I run Markovian Discriminator at every stage of the training. At the bottom stage (stage 0) of training, the receptive field is 11x11, occupying a large proportion in the images. So the generator learns to generate the larger patches of the training image, including general structure, base color, and large object. As the training stage goes higher, the receptive field occupies a smaller proportion of the image, which means the discriminator needs to look into a tiny little patch, such as a leaf in a picture of some trees, and decide whether it is real or fake. These force the generator to generate the finer details like image texture and small object, otherwise the image will be classified as fake.

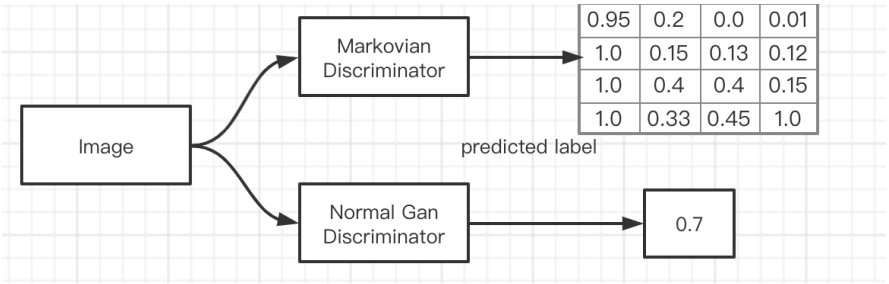

Figure 2: Markovian Discriminator

4. Implementation:

The training pipeline shows the process of construction and training of the generative adversarial network, which is training on a single natural image and can be applied to several tasks, including unconditional image generation, image editing, image harmonization, and paint to image transformation.

4.1. Experiment setting:

As I haven't got any computer with powerful GPUs, the whole project was carried out on the Google Colab platform, which provided me a P100 GPU. Training a 250x250 image takes about 60 minutes. And generating 100 random samples takes about 10 seconds. In the preprocessing part, the training image is resized to below 250, and larger than 25. In the initialization part, I initialized a set of generators and discriminators. Each of them is equipped with five convolutional blocks. The convolutional block's top is a convolutional layer with 32 kernels, and then followed by a batch normalization layer and a LeakyReLU layer. The alpha of the LeakyReLu layer is set to 0.2. I trained the model in 2000 epochs. In each epoch, I trained both the discriminator and the generator three times. The learning rate is an Adam optimizer was used to optimized the learning process, in which beta_1 is 0.5 and beta_2 is 0.999. The coefficient for the gradient penalty is 0.1, and the coefficient for reconstruction loss is 10.0.

4.2. Data preprocessing

The data preprocessing part aims to provide the model with standardized data to ensure that the model can run stably for a long time and avoid training failures caused by format errors. In this project, data preprocessing consists of three parts: handling input image, calculating the downsampled scale factor and the scale number, and creating a list of downsampled images according to the scale factor and the scale number.

In the first part, I used the PIL module to read the image. However, the number of channels may vary from different images, and the Tensorflow model needs to be given a fixed number of channels explicitly. Usually, an image has three channels, which are Red, Blue, Blue (RGB). While the grayscale image has only one channel, every pixel is represented by one number in the range of [0,255], where 0 represents the black color, and 255 represents the white color. There is also a four-channel image with one more value alpha that describes the pixel's transparency compared to the three-channel picture (RGBA). To deal with this problem, I turn all images into typical 3-channels images by

using the convert function of PIL. After that, the img_to_array function and type caster were used to convert images into an array with float values. After that, to ensure the networks have similar performance and time consumption for different images, I resized the image if the longer side of the image is larger than the parameter MAX_SIZE, 250 in default. Then the length of the longer side of the image will become MAX_SIZE, and the resize factor will resize the image's shorter side. For the case that the length of edges is all smaller than MAX_SIZE, the image remains original size. After that, the image will be normalized from [0,255] to [-1,1]. This is a widely accepted method to accelerate the model convergence. Then return the array of the images.

Second, the downscaling factor the number of scales is calculated by the function calc_scale_and_factor(). This function's idea is to calculate the number of scales needed to resize the shorter length of the image to be close to MIN_SIZE using the initial scale factor. And then to use the number of scales calculated above to calculate the scale factor that can resize the shorter length of the image to MIN_SIZE. The first step is to find out the length of the shorter edge of the image. The number of scales is calculated by the following formula, where init_s is the initial scale factor of the model, MIN_SIZE is the minimum length of image acceptable for this mode, and shorter_length is the length of the shorter edge of the image.

$$scale \leq log_{init\_s} \frac{MIN\_SIZE}{Shorter\_length}$$

And the scale factor is calculated by this formula, where scale is the number of scales calculated in the above formula, MIN_SIZE and Shorter_length is the same meaning as in the above formula.

$$scale\_factor = \sqrt[scale]{\frac{MIN\_SIZE}{Shorter\_length}}$$

Finally, a list of downsampled images was created by the create_downsamples() function. Basically, this part performs the operation of $down_n = real * s^{N-1-n}$. Besides, tf.expand_dims is used to add an empty dimension before the RGB channel because the Tensorflow model receives a four-dimensional array (Batch size, R, G, B) as an input.

4.3. Initialization of discriminators and generators

After data preprocessing, the next step is to initialize the network. I created multiple-stage discriminators and generators where the stage amount is the number of scales calculated in the data preprocessing part.

The discriminator network is designed by Tamar Rott Shaham in SinGAN, consisting of five convolutional blocks where the former four blocks follow the structure of 3x3 Conv-BatchNormalization-LeakyReLU, and the last block is only a 3x3 Convolutional layer. However, I made a change to this structure. In Tamar Rott Shaham's paper, the filter number of convolutional layer start with 32 and double every 4 stages in the pyramid structure of SinGAN. However, I made filter number 32 throughout all stages. Because I found the training at a higher stage with SinGAN's structure is very unstable, and the generated images are distorted. I will show the difference in the following experiment section. The following picture shows the structure of the convolutional block and discriminator network.

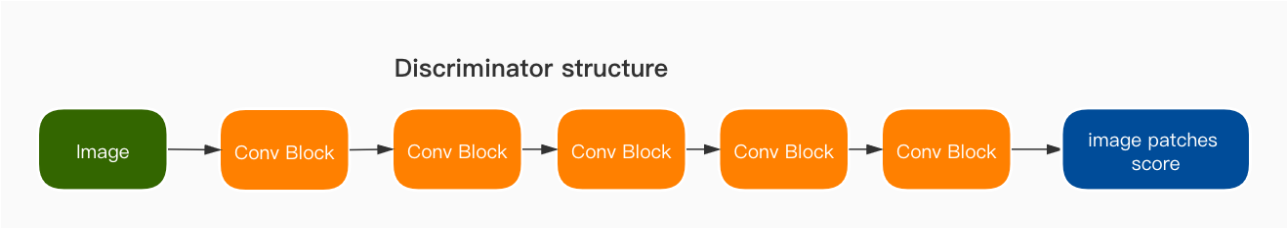

Figure 3: Discriminator Structure

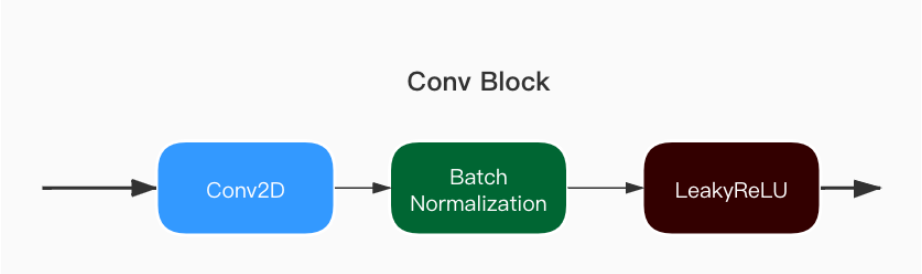

Figure 4: Convolutional Block Structure

The generator network is similar to the discriminator, which is consisting of 5 blocks of 3x3 Conv-BatchNormalization-LeakyReLU. There are two differences between the discriminator network and the generator network. First, a Tanh activation layer is added at the end of the network, which turns the distribution between -1 and 1, which is the same interval of the training image after preprocessing. Second, there is a skip connection between the input upsample image and the output. So that the generator can learn the residual images of the input upsample images, rather than the learning the desired underlying mapping directly. This skip connection plays a role as job dispatcher, distributing the job of learning the structure to the lower stage generators and distributing the job of learning details to the higher stage generators. The following picture shows the structure of the generator network.

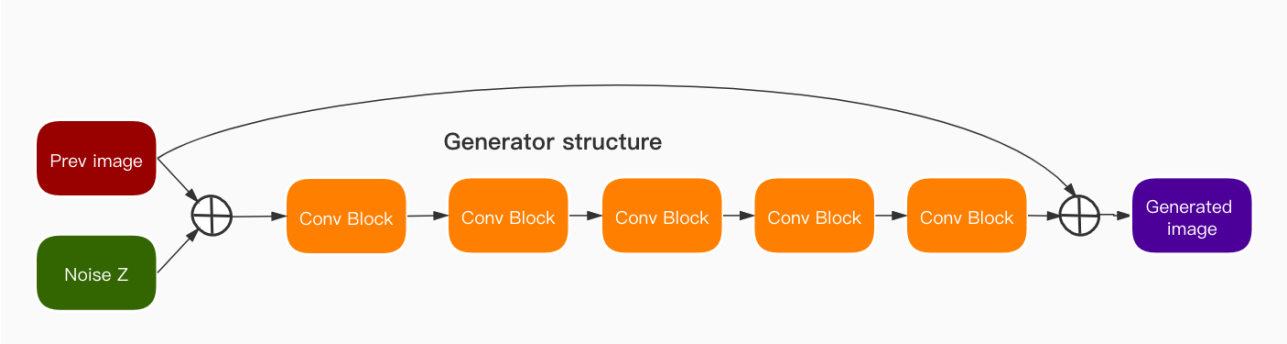

Figure 5: Generator Structure

Since both generator and discriminator are fully convolutional, so they can receive images with any size. To implement this, I add an input layer in which the width dimension and height dimension are set to None. The figures below are the implementations of the discriminators and the generators.

```
def create_generator(cur_scale):
  filter_num = 32
  pad = tf.keras.layers.ZeroPadding2D(5)

  prev_img = tf.keras.layers.Input(shape=[None,None,3],name="prev_img")
  noise = tf.keras.layers.Input(shape=[None,None,3],name="noise")
  pad_prev_img = pad(prev_img)
  pad_noise = pad(noise)

  x = tf.keras.layers.Add()([pad_prev_img, pad_noise])
  for i in range(4):
    x = convBlock(filter_num, 3, 1)(x)

  x = convBlock(3, 3, 1, norm="none",activation="tanh")(x)
  x = tf.keras.layers.Add()([prev_img,x])

  return tf.keras.Model(inputs=[prev_img, noise], outputs=x)
```

Figure 6: Implementation of Generator

```
def create_discriminator(cur_scale):
  filter_num = 32
  img = tf.keras.layers.Input(shape=[None,None,3])
  x = img

  for i in range(4):
    x = convBlock(filter_num, 3, 1)(x)

  x = convBlock(1, 3, 1,norm = "none",activation="none")(x)

  return tf.keras.Model(inputs=img, outputs=x)
```

Figure 7: Implementation of Discriminator

After creating these sets of generators and discriminators, I use Gaussian distribution with mean equals to 0.0 and a standard deviation equal to 0.02 as initializer to initialize the networks' weight.

4.4. Trainer

The trainer is the abstract of the training pipeline. It combines all the training processes, including data preprocessing, initialization of discriminators and generators, and the training loop into the main_train class. So the training pipeline can be done by creating a new instance of main_train class and invoke the train() method.

Training single image

```
[ ]  trainer = main_train()
     trainer.train("balloons.png")
```

Figure 8: Trainer Instance

Once a main_train object is created, the object's init function will be called. The init function defines some of the hyperparameters of the model, including the number of training epochs, the number of gradient step for discriminator (D_STEP) in each epoch, the number of gradient step for generator (G_STEP), the coefficient of gradient penalty (ALPHA), and the coefficient of Reconstruction loss (BETA).

The model starts training when the train() function is called. The train function first preprocesses the input image. And then, init_model() is called to create multiple stages of discriminators and generators. Next, I train each stage individually by calling the train_single_scale() function. I first define a varying learning rate by using the ExponentialDecay function, and after the training reaches a certain number of times, the learning rate reduces to one-tenth of the original. The intention of this is to allow the network to move quickly from the initial parameter value space to a better parameter value space at the beginning of training and reduce the learning rate in the middle of the training so that the model can gradually converge to the local optimum. Then I use Adam optimizer to optimize the learning process of the model, in which beta_1 is 0.5 and beta_2 is 0.999. Except for the bottom stage, the discriminator and generator at each stage load the model weight trained from the previous stage. On the one hand, since natural images have many cross-scale similar patches, the parameters of the previous stage can be optimized with a little learning, so the model of this stage can converge faster. On the other hand, loading the weight of the previous stage helps maintaining the continuity of the image style. I also found that loading the weights of the previous stage can generate more realistic images and avoid the problem of model collapse in experiments.

Now the model starts to run the epoch loop. In each epoch, the model generates an upsampled image and a reconstruction image generated by the previous generator. Then I calculate the noise_amp by 0.1* RMSE, where RMSE is the root mean square error between the reconstructed image and downsample real image at this stage and for stage larger than 0, for stage 0, as it does not have a previous generator, the noise_amp is set to be 1.0. This noise amplitude indicates the proportion of the noise that should be added to the image. After that, the model generates a Gaussian noise with the same shape as the real image and then multiplies it with noise_amp. Now that the model has a real image, a reconstruction image, and noise, it can start training.

First I train the discriminator by using the tf.GradientTape() function. This function automatically differentiates the loss function and provide the gradient for the neural network. In specific, first the generator generates a fake image, the discriminator gives the fake image a score “d_fake_score” and gives the real image a score “d_real_score.” Then the adversarial loss of discriminator is “d_fake_score - d_real_score” as it is trying to classify the real image.

The gradient penalty is calculated by the function gradient_penalty(), which creates an interpolation between fake and real images by a random value between 0 and 1. After that, I calculate the gradient of this interpolation. Here is one thing different from the gradient penalty in WGAN-GP. Instead of calculating the gradient of the whole image, I calculate the gradient of pixels in the image. Because in the original WGAN-GP, the model is trained on a batch of images, which require the 1-Lipschitz constraint to be satisfied for each image. While this model is trained on the patches inside the image, this requires the 1-Lipschitz constraint to be satisfied for each pixel. The implementation is shown in the figure below.

```
def gradient_penalty(dnet, real, gen_img):
  epsilon = tf.random.uniform(shape=[1],maxval=1.0)
  interpolation = epsilon * real + (1 - epsilon) * gen_img
  with tf.GradientTape() as t:
      t.watch(interpolation)
      interpolation_score = dnet(interpolation)
  gradients = t.gradient(interpolation_score, interpolation)
  gradient_per_pixel = tf.sqrt(tf.reduce_sum(gradients ** 2, axis=[3]))
  gp = tf.reduce_mean((gradient_per_pixel - 1.0) ** 2)
  return gp
```

Figure 9: Implementation of Gradient Penalty

Then the discriminator loss is the sum of the adversarial loss and the gradient penalty. Then I calculate the gradient of the loss and apply it to the discriminator variables to update the network weights. And then I Iterate the above steps 3 times in order to train a better discriminator, which can provide the generator with a more meaningful gradient. The implementation is shown in the figure below.

```
#----------------------------
#training discriminator
for step in range(self.D_STEP):
  with tf.GradientTape() as disc_tape:
    fake_image = self.gnets[cur_scale]([prev_img,noise])
    d_real_score = tf.reduce_mean(self.dnets[cur_scale](real_image))
    d_fake_score = tf.reduce_mean(self.dnets[cur_scale](fake_image))
    d_adv_loss = d_fake_score - d_real_score
    gp = gradient_penalty(self.dnets[cur_scale],real_image,fake_image)
    d_loss = d_adv_loss + self.ALPHA * gp

  d_grad = disc_tape.gradient(d_loss,self.dnets[cur_scale].trainable_variables)
  d_optimizer.apply_gradients(zip(d_grad,self.dnets[cur_scale].trainable_variables))
```

Figure 10: Training Discriminator

Then it comes to the training of the generator. The adversarial loss is the discriminative score of the fake image. Because the generator is trying to fool the discriminator, when the fake image gets a higher score, the model's loss is lower. And reconstruction loss is the MSE between the reconstruction image and real image. And then I iterate the above step 3 times. The generator loss is the sum of adversarial loss and reconstruction loss. The implementation is shown in the figure below.

```
#----------------------------
#training generator
for step in range(self.G_STEP):
  with tf.GradientTape() as gen_tape:
    fake_image = self.gnets[cur_scale]([prev_img,noise])
    fake_rec = self.gnets[cur_scale]([prev_rec,self.noise_rec[cur_scale]])
    g_real_score = tf.reduce_mean(self.dnets[cur_scale](real_image))
    g_fake_score = tf.reduce_mean(self.dnets[cur_scale](fake_image))
    g_adv_loss = -g_fake_score
    rec_loss = tf.reduce_mean(tf.square(fake_rec - real_image))
    g_loss = g_adv_loss + self.BETA * rec_loss

  g_grad = gen_tape.gradient(g_loss,self.gnets[cur_scale].trainable_variables)
  g_optimizer.apply_gradients(zip(g_grad,self.gnets[cur_scale].trainable_variables))
```

Figure 11: Training Generator

Then I finish training one epoch. After 2000 epochs, the training for this stage is finished, and I save the network weights of this stage and start training the next stage in the same process until all of the stages are trained. Then the training is done.

4.5. Inference

The method main_inference does the inference part. It is responsible for using the trained model to generate images that meet the needs of the task, including generating random samples, paint-to-image translation, image harmonization, and image editing. In the inference part, only the set of generators will be used. In the implementation, I first create a loader used to load the trained model's parameters and network weight. The loader's implementation is shown in the following picture.

```
def create_loader(img_path):
  loader = main_train()
  #load num_scale, scale_factor, real_images
  loader.preprocess(img_path)
  #load gnet, noise_amp
  loader.init_model()
  for scale in range(loader.num_scale):
    loader.gnets[scale].load_weights(f'weight/{loader.file_name}/g/{scale}')
  amp_dir = f'weight/{loader.file_name}/amp.npy'
  loader.noise_amp = np.load(amp_dir)
  #load noise_rec
  rec_dir = f'weight/{loader.file_name}/rec.npy'
  loader.noise_rec[0] = tf.convert_to_tensor(np.load(rec_dir))
  for scale in range(1,loader.num_scale):
    loader.noise_rec[scale] = tf.zeros_like(loader.real_images[scale])

  return loader
```

Figure 12: Implementation of Loader

The design of loader is mainly to reduce memory pressure. Since each training picture need a model to train, so if I want to train a dataset of pictures, there will be many models, and the memory will overflow. So after training, I save the network weights and model parameters, and then destroy the trainer object. After all the pictures in the training set have been trained, use the loader to load their network weights and model parameters one by one to inference.

After creating the loader, the loader will load the network weights and parameter according to the name of the training image. Then according to different tasks, it will execute different processes.

4.6. Generating random samples

In the generating random samples task, there are two kinds of image pattern, low variance, and high variance compared to the training image. For the high variance pattern, starting from the bottom stage, I feed the Gaussian white noise into every generator network stage. The generated image is purely generative. For the low variance pattern, I feed the generator networks with fixed reconstruction noise.

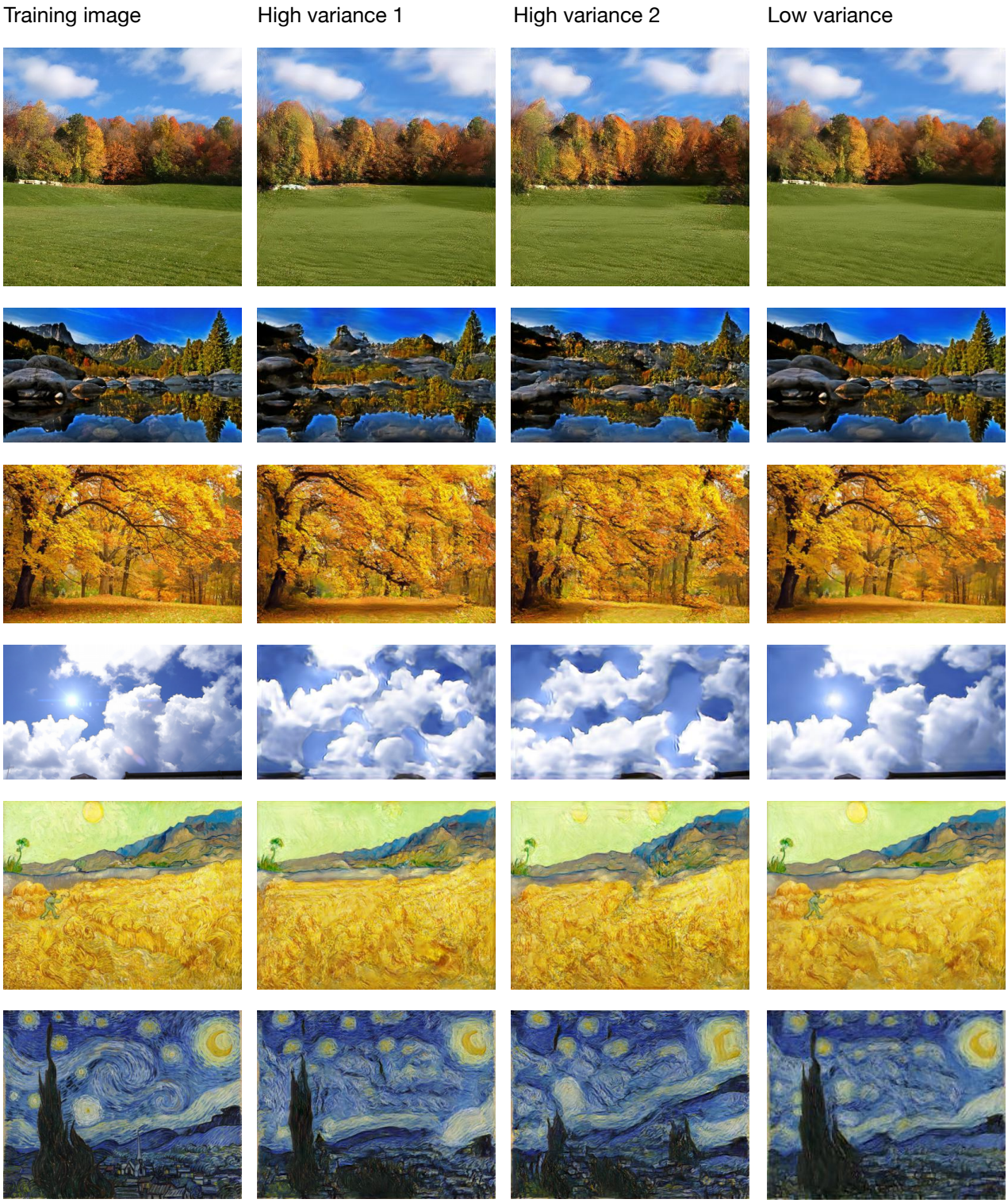

Figure 13: Generation of Random Samples

4.7. Paint to image translation

For the paint to image translation, the model receives a paint, and then generate an image with similar distribution of the training image. So the structure of the image should be similar to the paint, and the details for example texture should be similar to the training image. So it is a little bit different in implementation. First I inject a downsampled version of paint into the generator in N stage, where N is larger than 0, and the generator at this stage generated an image by Gaussian white noise and the injected image. After that I deliver it to the upper stage generator. And the rest part is the same as generating random samples task that past the upsample generated image throughout the network.

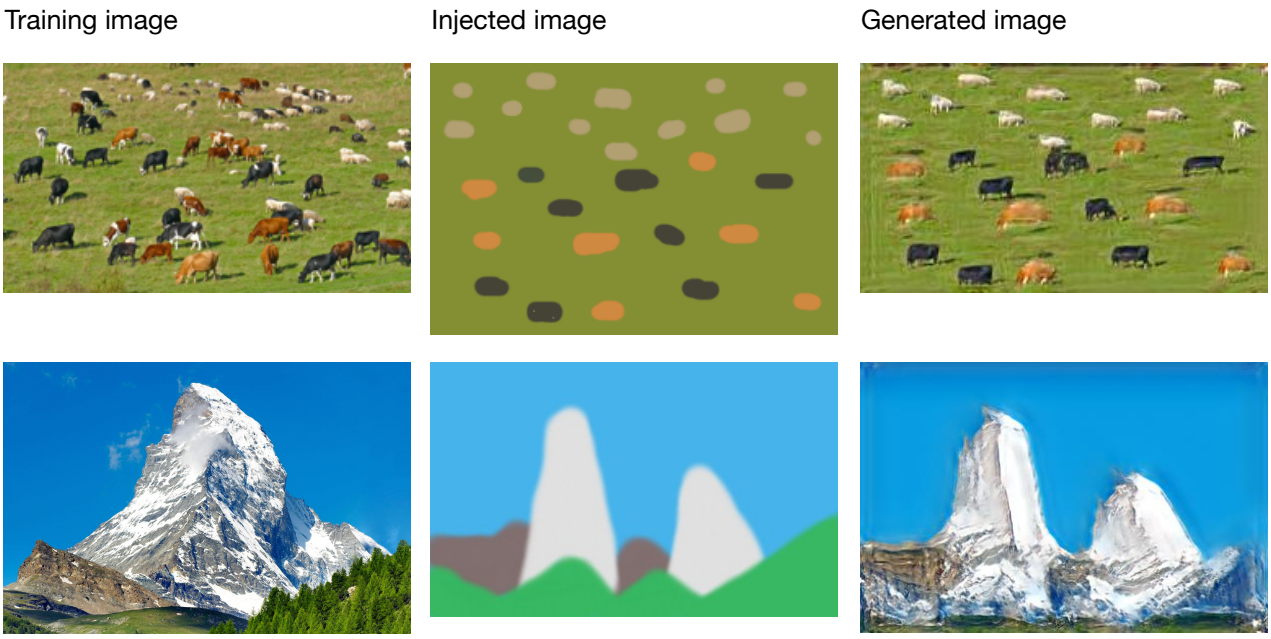

Figure 14: Paint to Image Translation Examples

Note: The training image and the injected image of the cows comes from the dataset of SinGAN.

4.8. Image harmonization

For the image harmonization task, the model needs an injected image and a binary mask image as inputs. The injected image is constructed by two images, where one image is the training image and the other is an object image, and the object image is pasted naively into the training image. Also, the model needs a binary mask to indicate where the training image has been manipulated. To be specific, first the injected image is fed into the generator list and the model generates the fake image. And then, I dilate the mask by the Skimage.morphology.binary_dilation() function, so that the mask becomes smoother. And then I cropped the fake image according to the mask image and filled the remaining part with the real image. Then the image harmonization part is done. Here are some examples of image harmonization.

Training image | Injected image | Generated image

Figure 15: Image Harmonization Examples

4.9. Image editing

For the image editing task, the model receives an injected image and a binary mask as the inputs. The injected image is the edited training image, in which the image block may be removed, expanded, or covered by other image blocks. The implementation of image editing is mostly the same as the image harmonization part, except the dilation coefficient for the mask is larger than image harmonization. Here are some examples.

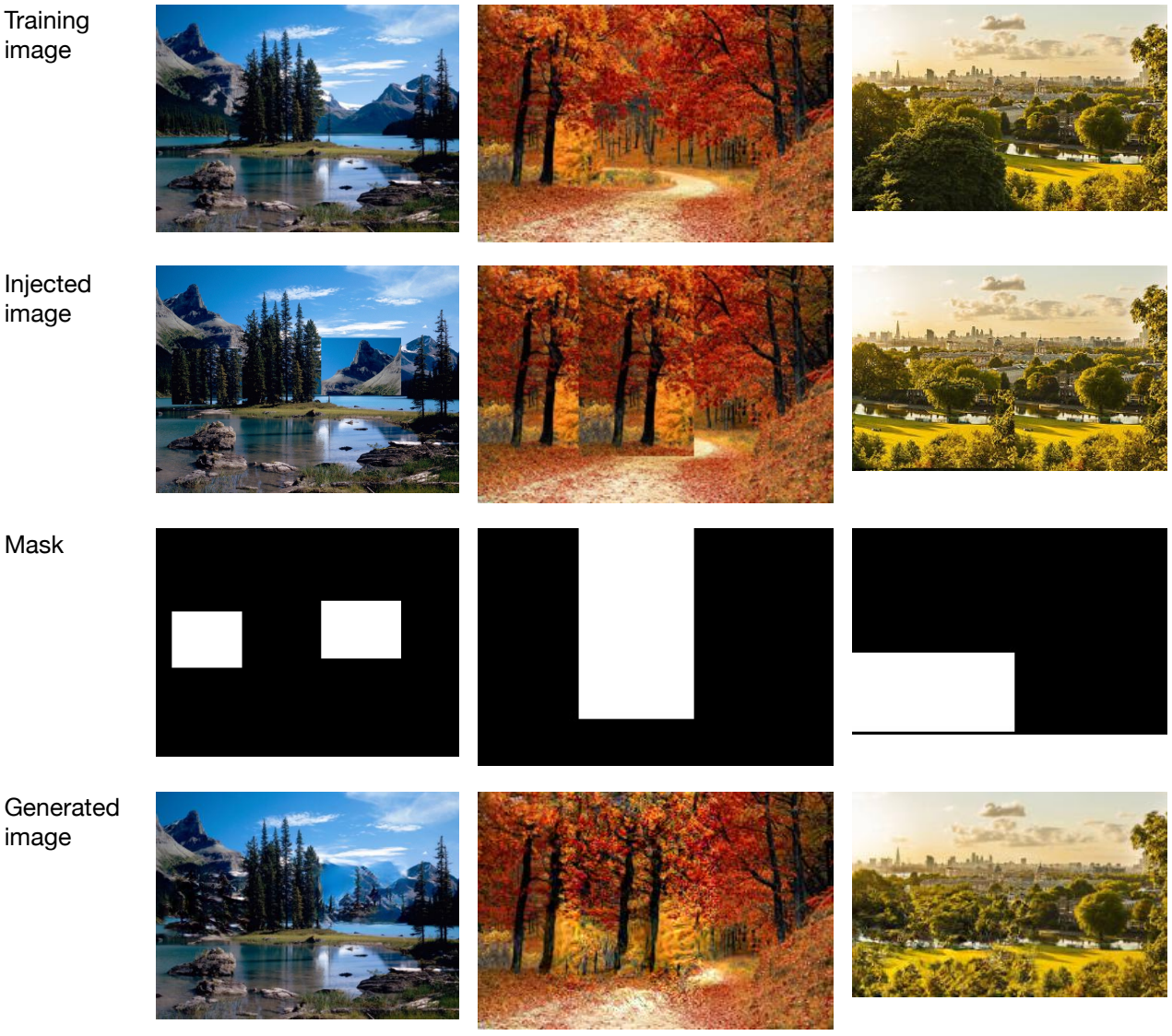

Figure 16: Image Editing Examples

5. Result

There are many hyperparameters and the model is very sensitive to them. Slightly changes on the hyperparameters can lead to a huge difference in the generated image. In order to find out the hyperparameters that are the best fit to the model, several experiments were conducted. And because the commonly used evaluation standard, Inception score (IS) and Fréchet Inception Distance (FID), are used for model that is training on a dataset, they are not suitable for this model. So I did a qualitative analysis of the generated images.

5.1. The effect of the number of training stages

The lower stage's receptive field occupies a large proportion in the images, and the receptive field at the higher stage occupies a smaller proportion of the image. So if the model only has one stage, the discriminator will look into the tiny little patches inside the image rather than looking at the whole structure. As a result, the model will learn and generate only the texture of the training images. But if the model has multiple stages, the model can learn both the general structure and fine details of the images, which will definitely increase the quality of the generated images.

Training image

2 stages

4 stages

6 stages

9 stages

Figure 17: Experiment on the Effect of the Number of Stages

5.2. The effect of different injected stage

This experiment is conducted for image harmonization, image editing, and paint to image translation tasks. These tasks require a manipulated image as input, and then feed it along with noise to a series of generators. However, choosing at which stage to inject the image has a great impact on the result.

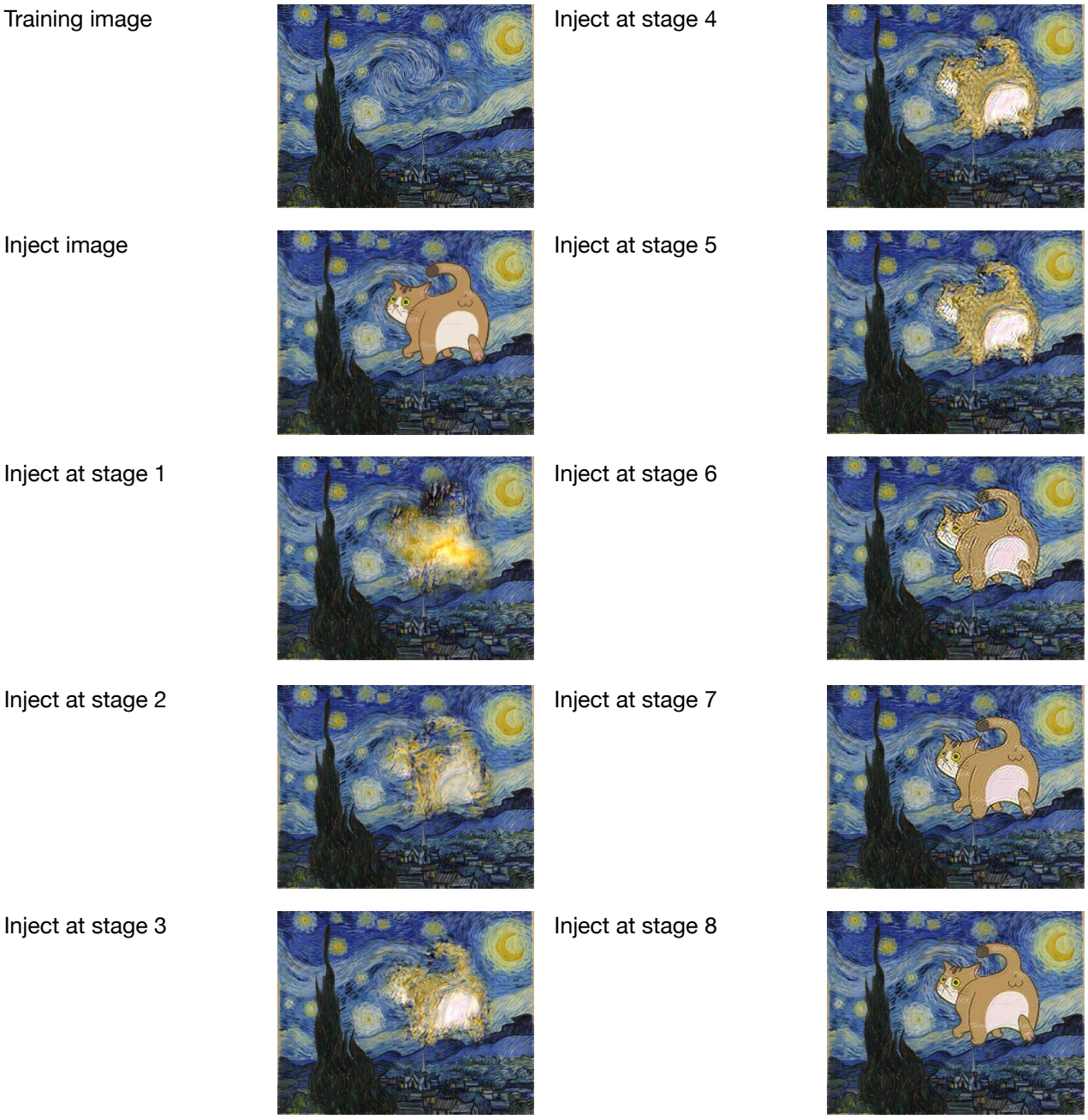

Figure 18: Experiment on the Effect of Different Inject Scale

If I inject the image at a lower stage, the model will generate a image with modified outlines and details. However, this will make the object completely unrecognizable. But if I inject the image at the upper stage, the model will only add some little details on it and it

is hardly to tell the difference between generated image and injected image. So there is a tradeoff between the consistency of the image and the recognizability of the object.

5.3. The effect of the Removal of BN

It is mentioned in WGAN-GP that BN may destroy the independence of the samples, causing gradient penalty to be useless. However, both gradient penalty and BN were used in the SinGAN's model. So I conducted an experiment on the effect of using BN and removing BN in this model.

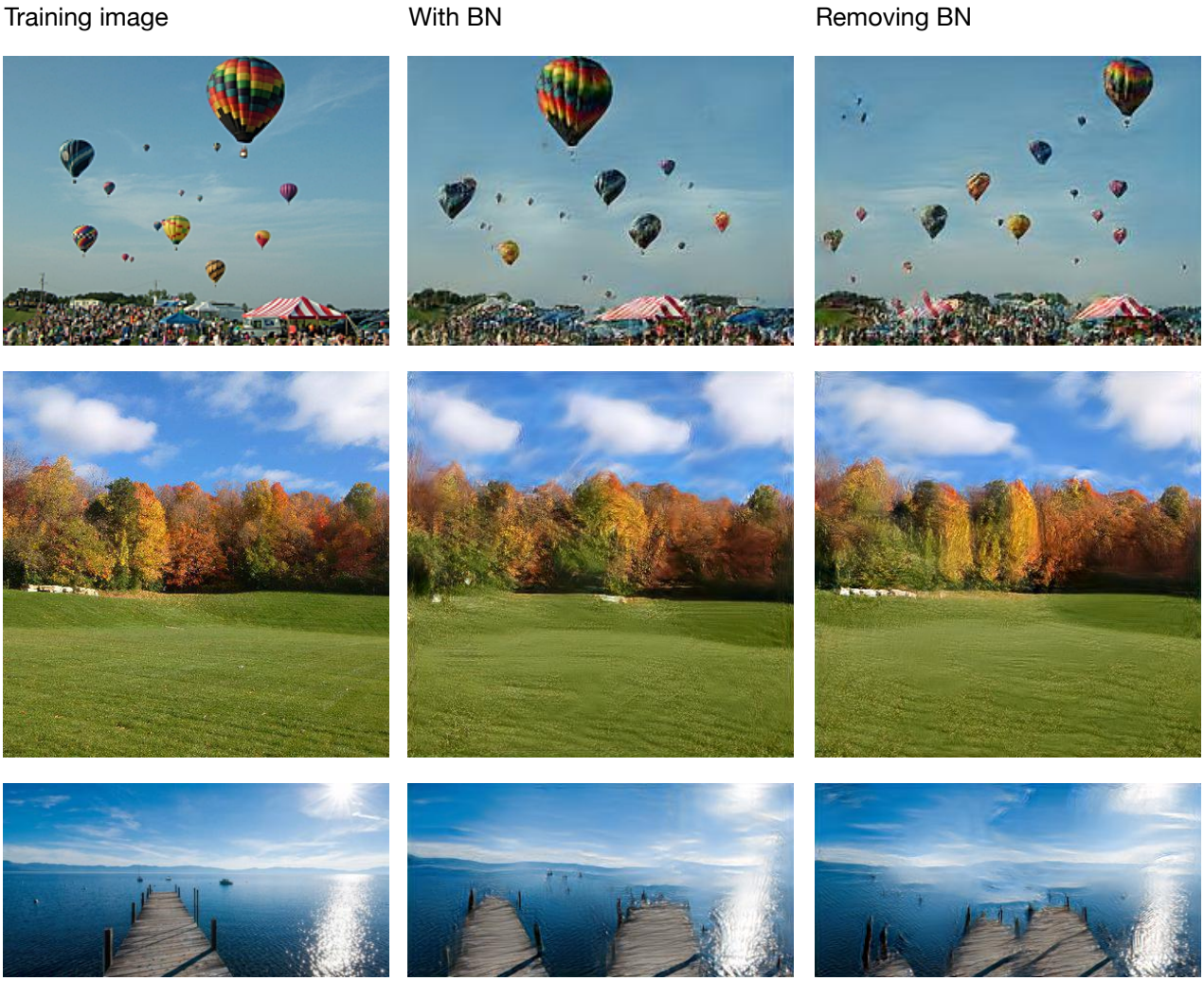

Figure 19: Experiment on the Effect of Removing BN

As a result, we can hardly tell the difference between the generated image under these two setting. I think the reason is that for the BN and gradient are incompatible in the model that is trained on the dataset. However, this model is trained on single image, in other word, the batch size is 1. So the existing of BN will not break the independence of the sample, and not decrease the quality of generated images.

5.4. The effect of Weight Loading

When I was still implementing this model, there is a period that no matter how I adjusted the hyperparameters, the generated images were very “abstract" as shown in the figure below. It only captured the very general contours and colors in the picture. I could recognize the blue sky and white clouds above, the very fuzzy trees in the middle, and the grass below. However, the details in the picture such as the leaves in the tree, the texture of the grass, and the texture of the clouds are not generated at all.

Figure 20: Experiment on the Effect of Weight Loading

However when the networks load the weight from previous stage, the quality of generated images are way better. Because there are many similar cross-scale patches in a natural image, so part of the network weights of this stage are also applicable to the networks of the next stage. Under the same number of training, this model can generate more detailed and realistic pictures, and the loss of the network will converge faster.

5.5. The effect of the number of filters

Every time we train a new neural network, we need to adjust the number of filters. Because the number of filters will affect the network's performance, this is especially true in gan. An appropriate number of filters can help the discriminator find a better decision boundary, and also allow the generator to generate a more reasonable image. So I tested the impact of different filter numbers on the generated pictures.

Figure 21: The effect of the number of filters

However, I found that in all experiments, as long as the number of filters is greater than 32, the model collapsed at a certain stage of training, generating a meaningless monochrome image. This is one difference between this model and the SinGAN. SinGAN uses 32 kernels at the bottom stage and doubles it every four stages, while this model uses a fixed number of filters 32 in all stages.

5.6. Comparison between SinGAN

This project is developed based on the singan model, so they are similar. And there are also some changes based on the experimental results to better fix this project. So I compared the generated pictures of the two models to compare the performance of the two models.

Figure 22: Comparison between SinGAN

SinGAN generate very nice balloons samples with high variance, the generated image learn both the shape and the texture of the balloons. In my model, although the texture of the balloons is well generated, the shape of the balloons are distorted. This means that the generator at lower stage did not correctly learn the image feature. For the image “night”, the image generated by the two models are similar. So, although sometimes it can produce results comparable to SinGAN, SinGAN is still better overall. I

think the main reason is these two models have different filter numbers. In SinGAN model, the filter number double in every 4 stage, which strengthen the ability of both network. While in my model, the filter number is 32 throughout all stages, so the networks can be too weak to learn the features of the training image. However, SinGAN's approach is incompatible with my model because it leads to model collapse in every experiment.

6. Limitation and Future work

After analyzing many generated results, I found that there is a limitation within this model. This model is not suitable to train on the images with complex textures. To be specific, when the training image has many different objects and appears dispersedly in the image, the model will generate an unrealistic image, as shown in the following figure.

Figure 23: Failure case

Several points can be improved in the future. First, to address the problem above, the network structure should be adjusted. I will try to introduce a deeper network for this model to capture the training image distribution better. Second, This model should be able to support more image manipulation tasks. I will develop the model in order to handle not only the animation generation and image super-resolution, as shown in SinGAN, but also extend it to video inpainting and denoising tasks. Third, I will embed this model into a website or mobile app, and then provide some interactive methods like dragging an object into the training image or removing a patch from the training image to use it more conveniently.

7. Conclusion

In conclusion, I completed all the goals set for this project. I implemented a generative model based on single-image training. This model can accurately capture the internal distribution in the training images, and then generate very realistic and highly diverse images. At the same time, this model can also be applied to several image manipulation tasks, including image editing, image harmonization, and paint to image translation. Although the model will produce unreasonable results in training images with complex textures, this is still a very powerful generative model. I hope that one day in the future, it can be migrated to the mobile app and be known and used by more people.

8. Reflective Learning

During the development of this project, I encountered many difficulties and solved them one by one. I learned a lot from it, not only technical knowledge but also how to manage a project and some mental improvements.

First of all, in order to complete this project, I learned a series of deep learning frameworks and libraries such as Tensorflow, Skimage, and some Linux operations because I was running the model on Colab. Before this project I only learned Keras in class, so they are new to me. Keras is a very high-level deep learning API, so it does not provide enough supports for the underlying operations. In order to get to control the lower-level details, I have two choices: Tensorflow and Pytorch. Pytorch is actually a better choice because it is a more widely used framework in academia, but TensorFlow is commonly used in companies. So I taught myself tensorflow2.0. The learning process is very tortuous, it is very different from Keras's development process, and took me about a month to get acquainted with tensorflow2.0, and I have been learning during the development process. Also I studied GAN systematically, including its design philosophy, various model structures, and loss functions. What I learned is not only this knowledge but also the ability to learn specific knowledge quickly

I also learned the importance of the development process. The establishment of the development process will make the project proceed more smoothly and coherently. However, during this project, I often do not develop according to the process. For example, I would write several modules simultaneously, causing me to forget which modules have been modified, which eventually led to many bugs. Also, when I adjusted the model hyperparameters, I did not set up an experiment log, so there are many sets of parameters I tried and repeated several times on different days, which caused me to waste a lot of time.

The last point is about mental adjustment. After seeing this project's requirements, I read many papers and many codes, and then I naively thought that this project, including the code and papers, could be completed within a month. This wrong estimate caused slow progress on my project. I learned that people should not be overconfident but more humble.

9. Acknowledgements

Throughout the writing of this dissertation, many people have offered me unmeasurable help and encouragement.

First, I would like to show my greatest respect to my supervisor, Dr. Yukun Lai, who was always very kind to me. He was very patient to discuss project issues with me and gave me a lot of inspiration and useful guidance. This project could not be completed so smoothly without him.

Second I would like to thank my parents who gave me the opportunity to study in UK. The time spent in the UK not only increased my knowledge, but also broadened my mind. This is definitely an unforgettable memory for me.

Third I would like to thank my girlfriend who was always staying by my side, without whom I can’t study in a relaxed and cheerful mood.

10. Reference

[1] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation*, *1*(4), 541-551.

[2] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097-1105).

[3] Agnihotri, A., Saraf, P., & Bapnad, K. R. (2019, December). A Convolutional Neural Network Approach Towards Self-Driving Cars. In *2019 IEEE 16th India Council International Conference (INDICON)* (pp. 1-4). IEEE.

[4] Alakwaa, W., Nassef, M., & Badr, A. (2017). Lung cancer detection and classification with 3D convolutional neural network (3D-CNN). *Lung Cancer*, *8*(8), 409.

[5] Gidaris, S., & Komodakis, N. (2015). Object detection via a multi-region and semantic segmentation-aware cnn model. In *Proceedings of the IEEE international conference on computer vision* (pp. 1134-1142).

[6] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... & Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems* (pp. 2672-2680).

[7] Kingma, D. P., & Welling, M. (2013). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.

[8] Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2017). Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*.

[9] Liu, M. Y., Breuel, T., & Kautz, J. (2017). Unsupervised image-to-image translation networks. In *Advances in neural information processing systems* (pp. 700-708).

[10] Johnson, J., Alahi, A., & Fei-Fei, L. (2016, October). Perceptual losses for real-time style transfer and super-resolution. In *European conference on computer vision* (pp. 694-711). Springer, Cham.

[11] Ledig, C., Theis, L., Huszár, F., Caballero, J., Cunningham, A., Acosta, A., ... & Shi, W. (2017). Photo-realistic single image super-resolution using a generative adversarial network. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 4681-4690).

[12] Zhan, F., Zhu, H., & Lu, S. (2019). Spatial fusion gan for image synthesis. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 3653-3662).
[13] Guo, L., Liang, L., Zeng, S., He, J., & Dai, G. (2020, July). Gray Scale Image Coloring Method Based on GAN. In *Proceedings of the 3rd International Conference on Data Science and Information Technology* (pp. 71-75).
[14] Zhu, J. Y., Park, T., Isola, P., & Efros, A. A. (2017). Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE international conference on computer vision* (pp. 2223-2232).
[15] Shocher, A., Bagon, S., Isola, P., & Irani, M. (2018). InGAN: Capturing and Remapping the" DNA" of a Natural Image. *arXiv preprint arXiv:1812.00231*.
[16] Shocher, A., Cohen, N., & Irani, M. (2018). “zero-shot” super-resolution using deep internal learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 3118-3126).
[17] Shaham, T. R., Dekel, T., & Michaeli, T. (2019). Singan: Learning a generative model from a single natural image. In *Proceedings of the IEEE International Conference on Computer Vision* (pp. 4570-4580).
[18] Hinz, T., Fisher, M., Wang, O., & Wermter, S. (2020). Improved Techniques for Training Single-Image GANs. *arXiv preprint arXiv:2003.11512*.
[19] Huang, X., Li, Y., Poursaeed, O., Hopcroft, J., & Belongie, S. (2017). Stacked generative adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 5077-5086).
[20] Dinh, L., Krueger, D., & Bengio, Y. (2014). Nice: Non-linear independent components estimation. *arXiv preprint arXiv:1410.8516*.
[21] Van den Oord, A., Kalchbrenner, N., Espeholt, L., Vinyals, O., & Graves, A. (2016). Conditional image generation with pixelcnn decoders. In *Advances in neural information processing systems* (pp. 4790-4798).
[22] Oord, A. V. D., Kalchbrenner, N., & Kavukcuoglu, K. (2016). Pixel recurrent neural networks. *arXiv preprint arXiv:1601.06759*.
[23] Ballard, D. H. (1987, July). Modular Learning in Neural Networks. In *AAAI* (pp. 279-284).
[24] Razavi, A., van den Oord, A., & Vinyals, O. (2019). Generating diverse high-fidelity images with vq-vae-2. In *Advances in Neural Information Processing Systems* (pp. 14866-14876).

[25] Brock, A., Donahue, J., & Simonyan, K. (2018). Large scale gan training for high fidelity natural image synthesis. *arXiv preprint arXiv:1809.11096*.
[26] Kingma, D. P., & Dhariwal, P. (2018). Glow: Generative flow with invertible 1x1 convolutions. In *Advances in neural information processing systems* (pp. 10215-10224).
[27] Arjovsky, M., Chintala, S., & Bottou, L. (2017). Wasserstein gan. *arXiv preprint arXiv:1701.07875*.
[28] Radford, A., Metz, L., & Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*.
[29] Denton, E. L., Chintala, S., & Fergus, R. (2015). Deep generative image models using a laplacian pyramid of adversarial networks. In *Advances in neural information processing systems* (pp. 1486-1494).
[30] Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., & Courville, A. C. (2017). Improved training of wasserstein gans. In *Advances in neural information processing systems* (pp. 5767-5777).
[31] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision* (pp. 1026-1034).
[32] Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., & Hochreiter, S. (2017). Gans trained by a two time-scale update rule converge to a local nash equilibrium. In *Advances in neural information processing systems* (pp. 6626-6637).
[33] Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.
[34] Lim, B., Son, S., Kim, H., Nah, S., & Mu Lee, K. (2017). Enhanced deep residual networks for single image super-resolution. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops* (pp. 136-144).
[35] Zhu, J. Y., Zhang, R., Pathak, D., Darrell, T., Efros, A. A., Wang, O., & Shechtman, E. (2017). Toward multimodal image-to-image translation. In *Advances in neural information processing systems* (pp. 465-476).
[36] Rubinstein, M., Gutierrez, D., Sorkine, O., & Shamir, A. (2010). A comparative study of image retargeting. In *ACM SIGGRAPH Asia 2010 papers* (pp. 1-10).
[37] Tsai, Y. H., Shen, X., Lin, Z., Sunkavalli, K., Lu, X., & Yang, M. H. (2017). Deep image harmonization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 3789-3797).

[38] Buades, A., Coll, B., & Morel, J. M. (2005, June). A non-local algorithm for image denoising. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)* (Vol. 2, pp. 60-65). IEEE.
[39] Isola, P., Zhu, J. Y., Zhou, T., & Efros, A. A. (2017). Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1125-1134).
[40] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).