Specializing Generative Adversarial Networks To Render Terrain Textures

Hein H. Aung

March 20, 2018

Abstract

I studied Neural Networks for texture generation. In particular, I have researched and created Generative Adversarial Networks (GANs) for rendering new images of different types of ground terrain textures (such as soil, grass plains, drought plains, etc). A GAN consists of two neural networks: a discriminator and a generator. The generator create a batch of images with random pixels. On the other hand, the discriminator tries distinguish between the real image and fake image by taking the inputs from a set of real images and a set of fake images from the generator. There has been ample research done with GANs that produce synthetic 2D images or 3D graphical models. The GAN I have created, given a set of input images, will dynamically render a set of different variations of original images of terrain textures. For example, if the training input is a set of green grass, the output will be a set of withered grass plain, fresh grass plain, and so on. I sent out a survey to the campus to evaluate these images and my final results indicate that people can identify the generated textures type.

Contents

List of Figures

List of Tables

1 Introduction

The concept of artificial neural network is basically introduced from the subject of biology where neural networks play an important and key role in human cognition. In human body work is done with the help of neural network. Neural network is a web of inter connected neurons which are millions in number. Using these concepts, we have tried to create machines capable of not just logic and reasoning but also creativity and art. So far, the most striking successes in neural networks have involved discriminative networks [2], that classify a dimensional input to a class label [10]. These striking successes have primarily been based on the backpropagation algorithm, a method of training a neural network in which the initial system output is compared to the desired output, and the system is adjusted until the difference between the two is minimized and dropout algorithms, a very efficient way of performing model averaging with neural networks [11, 7]. Deep generative models have had less of an impact, due to the difficulty of approximating many unmanageable probabilistic computations that arise in the random estimation and related strategies. [9] These difficulties have been sidestepped by a new generative model estimation procedure, known as Generative Adversarial Networks (GAN), developed by Ian J. Goodfellow.

A generative adversarial network consists of two neural networks: generator and discriminator. The generator stochastically generates an image, and after a certain time of training, the images generated will ideally be indistinguishable from the training dataset of images.
Since it is generally infeasible to engineer a function that tells whether that is the case, a discriminator network is trained to do the assessment, and since networks are differentiable, we also get a gradient, the loss value (a value that tells whether the discriminator or the generator is doing a good job(closer to zero) or a bad job(far from zero)), we can use to steer both networks to the right direction.

As a fan in computer gaming, I find it quite amusing to encounter real world graphics, especially the real world environment in games. Engineering or drawing such textures takes times for artists. As GAN is filled with new artifacts for synthesis, I got motivated to develop GAN that synthesize terrain textures that mimic real world terrains. A texture can be represented as an image with repeated patterns. The goal of this ground terrain texture synthesis is to infer a generating process from a set of example textures, which then allows to generate arbitrarily many new similar samples of that texture. Success in this task is judged primarily by visual quality and similarity to the original texture as estimated by human observers, but also by other criteria such as the speed of learning [15], ability to generate different variations of the input texture. More applications of image rendering can be found in architecture, simulators, movies, and visual

effects.

So far, terrain have been procedurally generated through a host of algorithms designed to mimic real-life terrain. In games, the terrain and the game world need to be created by the game designers and artists. This greatly limits the extent to which the player can experience since the human designer can only build the static game worlds to a limited extent. However, being able to train machine learning algorithms to learn terrain is going to allow the game to dynamically generate new areas. This allows for more varied game play experiences such as Figure 1, "No-man sky" where the player has no restrictions or invisible walls and can travel extensively since the game world keep expanding. My research question is **Can we specialize Generative Adversarial Network to render terrain textures?**

In the body of this thesis, **Section 2** talks about related work done with GANs and other methods for texture synthesis. **Section 3** includes background of GAN, specifically about image classification, convolution neural networks, and a detailed description of GAN. **Section 4** talks about my approach. **Section 5** lists the tools I used and then in the following **Section 6** talks about methodology for my evaluation. **Section 7** begins to explain the algorithms of GAN I implemented, data processing and the training process of my GAN. **Section 8** and **Section 9** explain about the images (generated) and the evaluation of the images. Then, **Section 10** analyzes these evaluations. Lastly, **Section 11**, discusses the issues that come up during this research which will be left for future work.

2 Related Work

There has been ample research done with GANs that produce synthetic 2D images or 3D graphical models. Grigory Antipov et al. [1], proposed a GAN-based method for automatic face aging. Contrary to previous works that utilize GANs for altering of facial attributes, they made an emphasize on maintaining the original person's identity in the aged version of the person's face.

Gatys et al. [6] present a more data driven parametric approach to allow generation of high quality textures from a variety of natural images. Using filter correlations, a basic operation that extract information from images, in different layers of the convolution networks, an operation in artificial neural networks that is applied to analyze visual imagery, which is trained discriminatively on large natural image collections that results in a powerful technique that nicely captures expressive image statistics, classification of image to a certain class. However, creating a single output texture requires solving an optimization problem with

iterative back propagation, which is costly in time and memory. Therefore, in my thesis, I have decided to use a python library, Tensorflow, [15] which solves the back propagation.

Recent papers, Ulyanov's [16] and Johnson's[12] deal with that problem and train feed-forward convolution networks in order to speed up the texture synthesis approach of [5]. Instead of doing the costly optimization of the output image pixels, they utilize powerful deep learning networks that are trained to produce images minimizing the loss value. A separate network is trained for each texture of interest and can then quickly create an image with the desired statistics in one forward pass.

In designing generator and discriminator networks, I have implemented both networks based on Ulyanov's and Gatys' algorithms both of which includes using convolution networks.

There are other types of texture synthesis as well. Efros' paper[5] presents a simple image-based method of generating novel visual appearance in which a new image is synthesized by stitching together small patches of existing images. This process as they call it, Image Quilting. They first use quilting as a fast and very simple texture synthesis algorithm which produces surprisingly good results for a wide range of textures. Then they extend the algorithm to perform texture transfer, which is rendering an object with a texture taken from a different object.

3 Background

In this section, I will explain briefly about the concepts that are used for my thesis such as image classification, convolutional neural networks, and a formal description of generative adversarial network.

3.1 Image Classification

Images are 3-dimensional arrays of integers from 0 to 255, of size (Width x Height x 3). The 3 represents the three color channels Red, Green, Blue (RGB). The task in Image Classification is to predict a single label such as cats, dogs, ships etc. for a given image. Since it is relatively trivial for a human to recognize a visual concept (e.g. cat), it might not be true for computers. So, it is worth considering the challenges involved from the perspective of a Computer Vision algorithm. [4] In my thesis, since I will be using terrain textures as training datasets, I will have to make sure that the viewpoint variation, which is a single instance of an object that can be oriented in many ways with respect to the camera, should be solved.

Figure 1: No-Man's Sky.[3] The game uses procedural generation of terrains and textures of non-player characters (NPCs) and allow the player to explore different planets, solar systems, and star systems.

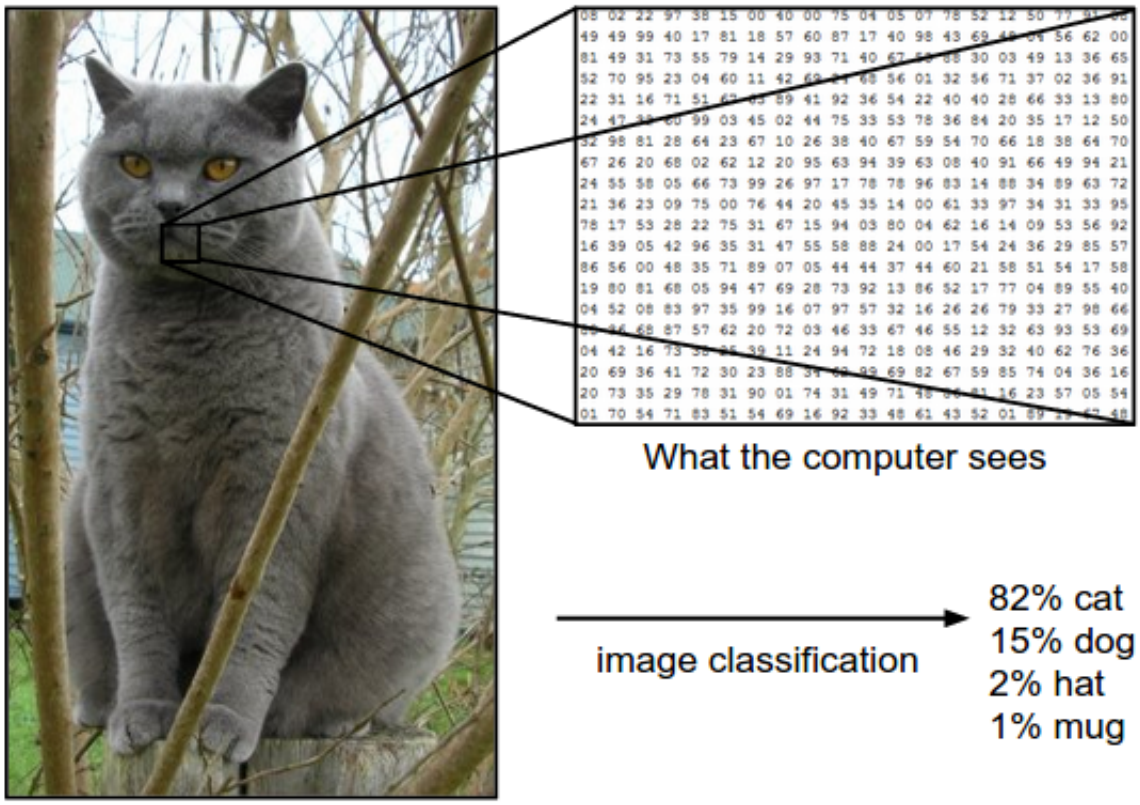

Figure 2: In the image above an image classification model takes a single image and assigns probabilities to 4 labels, cat, dog, hat, mug. [4]

3.2 Convolutional Neural Network (CNN)

Before we talk about convolution,we start with the definition of neural network. A neural network works like a network of neuron cells in human nervous system and they are all connected by *synapses*. The Figures 3 and 4, show a biological neuron and a mathematical form of a neuron. In Figure 3, each neuron receives input signals from dendrites and produces output signals along the *axon*. The axon eventually branches out and connects through synapses to the other *dendrites* of other neurons. In the mathematical model of a neuron in Figure 4, the signals that travel along the axons such as x_1 interact multiplicatively with the dendrites w_1 of other neurons to produce a signal w_1x_1. The idea is that the synaptic strengths w_1 are learned and control the strength of influence of one neuron on another. Moreover, they also decides whether w_1 is positive(excitatory impulse) or negative(inhibitory impulse). In this model, the dendrites carry the signal to the cell body where all the signals get summed. If the final sum is above a certain threshold, the neuron can fire, sending an impulse along its axon. In the mathematical model, the neuron

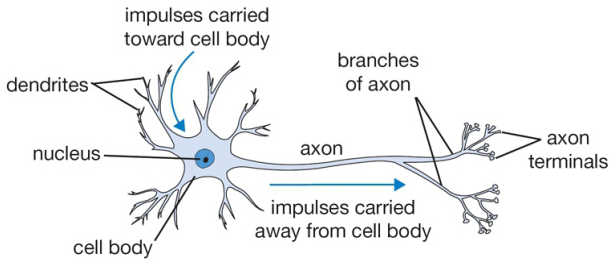

Figure 3: A cartoon of Neuron[17]

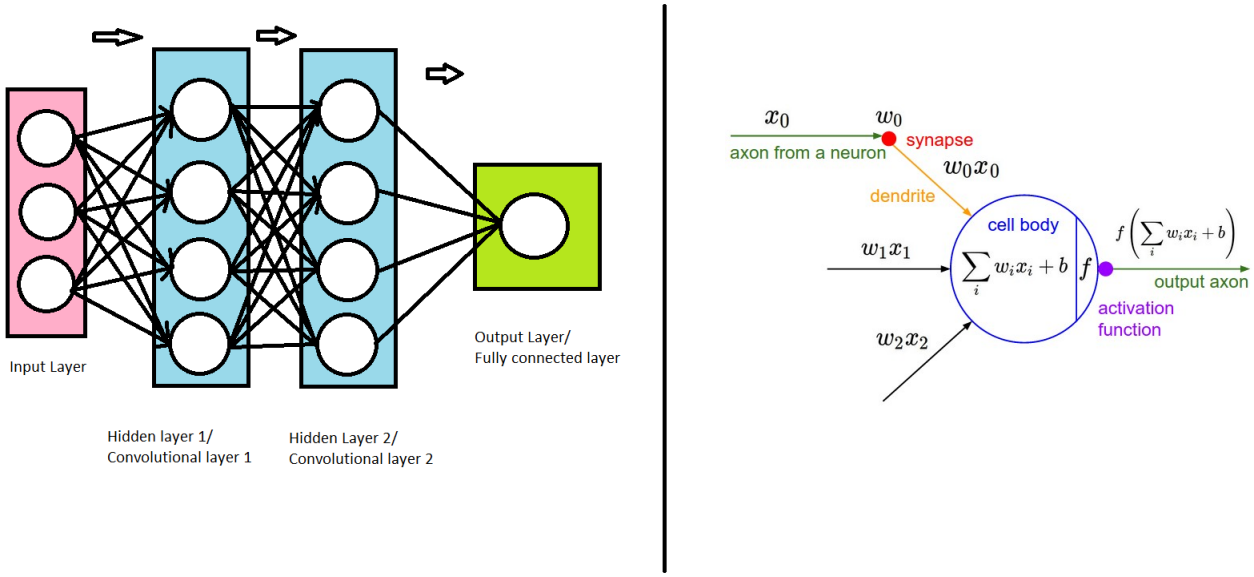

Figure 4: Multi-layered Convolution Network. Left: A Regular 3-layer Neural Network. Right: How a neuron would look like Mathematically[13].

fires its impulses through an *activation function* which are discussed in detail in a later section.

3.3 Generative Adversarial Network

As proposed by Ian Goodfellow [8], a generative adversarial network (GAN) consists of a generator and a discriminator, where the discriminator tries to classify real objects and objects synthesized by the generator, and the generator attempts to confuse the discriminator. To build a GAN, we have to create two neural networks. Then we make them compete against each other, endlessly attempting to out-do one another. In the process, they both become better at what they do. A common analogy that is used to describe the discriminator in GANs as a brand new police officer who is being trained to detect counterfeit money. Its job is to look atmoney and report if it is fake or real. For the generator, it will be a counterfeiter, who will

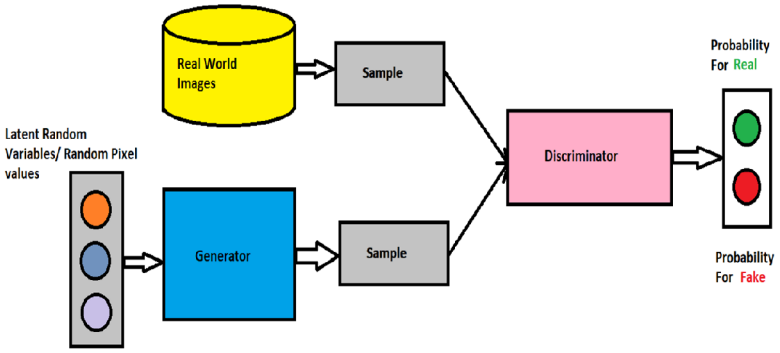

Figure 5: Generative Adversarial Network

create fake money from scratch, without any knowledge of what real money looks like.
The counterfeiter, in the first round, will create a batch of images that barely resemble money at all since it does not know anything about what money is supposed to look like. Since the police officer is also new at its job, we have to step in and show the officer a batch of real dollar bill. This back-and-forth game between the counterfeiter and the police officer continues thousands of times until both become experts. Eventually the counterfeiter is producing near-perfect counterfeit money and the police officer has turned into a master detective looking for the slightest mistakes.

To give a more formal mathematical definition of a GAN, Generative means the ability to produce, Adversarial refers to involving oppositions and thus, Generative Adversarial Networks can be defined as a network, that uses a contest to produce an image that look at least superficially authentic to human observers, having many realistic characteristics. As shown in Figure 5, the the generator takes in a creates a batch of fake images and the discriminator takes in two inputs, a batch of real images, and a batch of fake images created by the generator.

To learn the distribution of the generator, p_g over a batch of data Z, we have to have random input noise variable $p(Z)$ that represents a mapping of data space as $G(Z)$. We also define the discriminator distribution$D(X)$ which will outputs a single scalar. $D(X)$ represents the probability that X came from the data rather than p_g. We train the discriminator D to maximize probability of training examples and samples from the generator, G. At the same time, we train G to minimize $\log(1 - D(G(z)))$. A more detailed description of these mathematical algorithms will be explained in a later section.

4 Approach

To implement a GAN for texture synthesis, I include the schedule as follows.

1. Created a prototype: **Single Convolution layer** discriminator and a generator based on the algorithms of Ulyanov [16], Gatys [6], and tensorflow[15] tutorial, which is a python library for neural network.
2. To test the accuracy for this model, I have chosen MNIST dataset(Modified National Institute of Standards and Technology) which is a large database for hand-written digits used for training various image processing systems.
3. Find and process dataset. I searched for dataset for terrain textures and found a texture library. I specifically downloaded terrain textures of four categories (cracked ground, grass ground, pebbles, and pavement). Then, processed all images to have similar standard size.
4. Modify the prototype into **Multi Convolutional layer** along with the Generator. I implemented one more convolution layers in both discriminator and generator.
5. Develop Training algorithm based on Tensorflow tutorial and Siraj Raval's pokeGAN [14].
6. Train GAN and generate different variations of terrain textures using the trained model of GAN.
7. Design Evaluation in Google forms for generated images. Since I used human subjects as evaluators, I submitted my design Human Subjects Research Review Committee (HSRRC) of Union College.
8. Sent out Google form to campus and collected evaluations.)
9. Analyze Evaluations and write report.

5 Tools

I use Python for the entire implementation along with two libraries Nympy for scientific computations, and Tensorflow [15] a library for deep learning.

There are other libraries that help build neural networks such as Pythorch. However, I chose Tensorflow due to it's ease of use and gentle learning curve. The declaration and control of model parameters and variables are simple and putting together the model is intuitive Tensorflow has support for GPU with simple API calls making training faster. Therefore, I will also uses automatic differentiation, eliminating the need to write custom backpropagation methods.

Google open-sourced a Python library for performing fast gradient-based machine learning on GPUs. It comes with a vast knowledge base and tutorials including examples for most neural network architectures. Moreover it also has good developer community offering help for most roadblocks.
With Tensorflow, we can quickly build and test a neural network with few lines of code. Moreover, we can also change the type of processing units form CPU to GPU. For my prototype GAN, I created a single-layered discriminator and a single-layered generator for a prototype. To test this for, a training session is implemented using Tensorflow source code from the tutorial. For the final GAN, another training session is implemented using Siraj Raval's pokeGAN training algorithm [14].

6 Method for Evaluation

For evaluation, will be performing the manual qualitative evaluations on the results/ images synthesized from the generator after the whole training process is done. The manual evaluations include a survey where I will ask a set of random evaluators to determine what the synthesized images look like with a short text description within the ground terrain context as shown in figure 6. It is done in Google forms which was sent out campus wide to get the evaluations. Therefore, no research fund was needed. Since I used human subjects as evaluators, I submitted my design Human Subjects Research Review Committee (HSRRC) of Union College. Then, from the collected evaluations, I analyzed the text descriptions of each image and calculated the percentage in which the majority of people regard which class of ground texture, the image belongs to.

7 Architecture and Design

In this section, I will be explaining the algorithm I used to build the GAN. Since both the discriminator and the generator are neural networks, their architectures are formed by layers *Convolution* and *activation* (which are described down below). Each Layer accepts an input 3D volume which are *height x width x color channel* and transforms it to an output 3D volume through a differentiable function. For example, using an image of 128 x 128 pixels, the basic layers can be broken down into the following. Note that the images are (RGB) colored images, and thus, the number of color channel is 3.

7.1 Discriminator

- Input Layer: [128 x 128 x 3] will hold the raw pixel values of the image, in this case an image of width 128, height 128, and with three color channel.

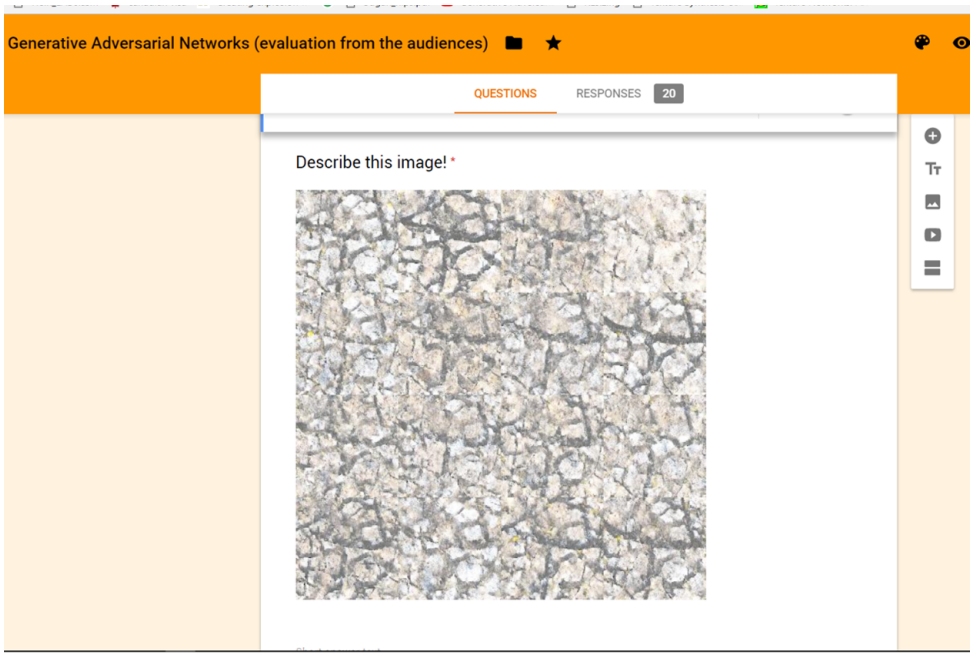

Figure 6: Google forms for evaluation

- Convolutional Layers: will compute the output, the 3D volumes of features, which is the matrix multiplication between the weight matrix W, called "filter", and a small region in the input volume called, Kernel size. This will result in [128 x 128 x W] since we are using W filters.
- Flattening will convert the output of the convolution part of the CNN into a 1D feature vector, to be used by the classifier. It gets the output of the convolution layers, flattens all its structure to create a single long feature vector (e.g [1, 128 * 128 * 3]) to be used by the dense layer for the final classification.
- Activation layer: will compute the class scores, for each convolution layer and transfer the results into other convolution layers. This convolution and activation repeats depending on how many convolution layer we have. For my case, it will be two layers.

Below is the architecture of the discriminator.

DISCRIMINATOR($input$)

filters = 64
// convolutes the input image into 64 filters so that
// we have[128 x 128 x 64]
conv1 = convolute(input, filters)

// relu means leaky rectified linear unit for activation function.
out-conv = relu $(conv1)$

// Flatten the out-conv
out-flat = reshape (out-conv, [-1])
// flattens into 1-D.[15]

// First fully-connected layer
initialize weight for fully connected layer w_{fc}
initialize bias for fully connected layer b_{fc}

// Activated function for fully connected layer
// matrix multiply w_{fc} and out-flat
out-fc = add(matmul(w_{fc}, out-flat)+b_{fc})

7.2 Generator

The Generator can be considered as a reverse of Convolution network. It takes a dimensional random pixel values and **upsample** the them to have a [128 x 128 x 3] image. Upsampling means increasing the sample size. Then, an activation function is used to stabilize the output and mapping it down to have [1, 128 x 128 x 3] pixel to match texture image from the discriminator.

- Input Layer: [64 x 64 x 32] will hold the random pixel values of seed images for creating the generated images. Since we used th filter of 64 in the discriminator, we chose the dimension of first input to be [64 x 64 x 32]. Just like the discriminator, we want to flatten the layer so that it is easier to multiply with the weights. Thus, the layer is *flattened* into [n, 64 x 64 x 32]. n refers to the number of seed images.

- Fully-connected Layer: is created using the random weights w and bias b as mentioned before.

- Deconvolution Layer: you want to have images of dimensions [128 x 128 x 3] in the end. Thus,this layer learns the filters/ weights from training and apply them back to the seed images.
- Output: an image with dimensions (128 x 128 x 3).

GENERATOR($input, rand\_dim$)

```
// setting a fully connected layer so that we can upsample it layer by layer
initialize weight for fully connected layer w_fc with rand_dim and
shape of [64 x 64 x 32].
rand_dim is usually 100 [14]
initialize bias for fully connected layer b_fc

// Activate layer
out_deconv = add(matmul(w_fc, input)+b_fc)

// flatten to get dimension of [64x 64 x 32]
flat_deconv = reshape(out_deconv, [-1, 64 x 64 x 32])

out_dim = CHANNEL = 3 // 3 color channels
// Deconvolution starts (uses manual upsampling using kernel filters)
// de-convlutes flattened image (deconvoulte = transpose & convolute)
// get [128 x 128 x 3]
deconv1 = convolute(transpose(flat_deconv, out_dim))

// Activated layer
out_deconv = tf.tanh(deconv1)
// tf.tanh will return a tensor respectively with
// the same float type as x[15]
```

8 Data

I have chosen four categories from texturelib.com, which are (1) cracked ground, (2) pebbles, (3) pavements, and (4) grass ground, with 120 - 190 images each. The choice is due to their distinctions among them such

Figure 7: Dataset: crackedground, pebbles, pavement, and grass ground

as pavements are more symmetric than cracked grounds so the generated image will also have distinctions for a human observer to evaluate/classify.

I feed in each category and let the networks train in the Union college CROCHET lab. The processor of the computer used to trained is Intel Core i5_4570s CPU @ 2.90 GHz x 4 and the graphic card is NVE7 64_bit. The training hour is dependent on the number of range I chose(usually more than 1000). I chose to the range to be 2000.

9 Training

The accuracy test for the prototype single convolution layer classifier was used based on the algorithm from Tensorflow and Siraj Raval [14]. One of the most important part about training GANs is to consider the loss value. The loss value can be referred to as the value of the difference between the fake image(predicted value) and the real world image (actual value). The loss value will be high if the discriminator is doing a poor job at classifying the training data, and will be low otherwise. Similarly, the generator will also have high loss value if it is creating poor images. We want the loss value to get closer to 0 so that both networks are actually learning and improving one another.

$$D_{loss} = \frac{1}{m}\sum_{i=1}^{m}[\log D(x^i) + \log(1 - D(G(z^i)))]$$

EQ 1: From [14] Computing loss function for the discriminator.The delta symbol denote the gradient of the

generator. m is the number of samples. D refers to the probability function by the discriminator and G refers to that of generator. x is a batch of real images and z is a batch of fake images.

The first part of the equation, x^i, refers to optimizing the probability that the real data (x) is computed highly. The second part refers to optimizing the probability that the generated data or fake data, $G(z)$ is rated poorly. Logarithmic functions are applied to these probabilities so that it is easier to decide the gradient − whether the loss is going high or low for the discriminator.

$$G_{loss} = \frac{1}{m} \sum_{i=1}^{m} \log(1 - D(G(z^i))$$

EQ 2: From [14] Computing loss function for the generator. In Figure 1.5, the term refers to optimizing the probability that the generated data G(z) is rated highly.

Since the generator is constantly improving images and the discriminator is constantly getting better at distinguishing the images, the idea is to create a loss function that can affect both the generator and the discriminator. The back and forth alternation of this gradient optimization between the discriminator and the generator, using these expressions on new batches of real and generated fake data each time, the GAN will eventually start to producing data that is as realistic as the network is capable of modeling.

As shown in Figure5 the discriminator must take in real world images and fake images from the generator. So, we calculate the probability of real world image (D_{loss}) and the probability of fake images (G_{loss}) as shown in Figure.

10 Results and Analysis of Evaluations

As I have explained in my methodology section, I sent out a Google Forms for evaluation via email campus wide. The form includes informed consent and thirty images of generated pictures in random order. The participants are asked to write a short description of each given image within the context of ground textures. I collected data on the Google Form from 3/1/2018 to 3/9/2018. I received 20 responses. For analyzing these responses, I have chosen to share, in this paper, the best results from each categories and two of the worst results.

For analysis of grass, since I have given freedom to the evaluators for text descriptions with less than

Require: D_{loss}, G_{loss}

$real\_image$ =$random\_image$
$random\_Input = random\_Dimension$
$fake\_image = generator(random\_Input)$

$real\_Result = discriminator(real\_image)$
$fake\_Result = discriminator(fake\_image)$

$D_{loss} = fake\_Result - Real\_Result$
$G_{loss} = -fake\_Result$

while $i \leq Epoch$ **do**
  while $j \leq batch$ **do**
    Update discriminator (d_{loss})
    Update generator (g_{loss})
    Print image

Algorithm 1: $Train_{GAN}()$

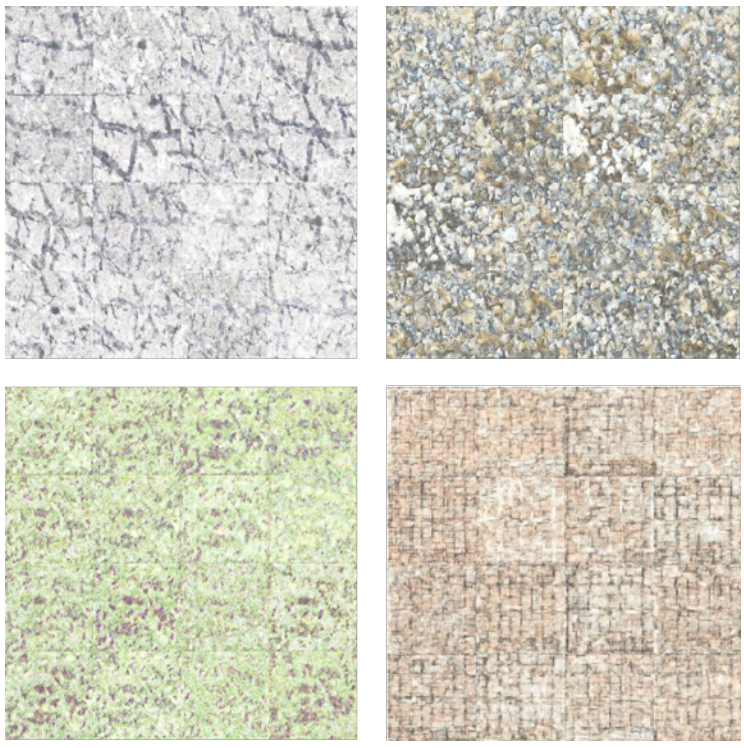

Figure 8: Generated images: cracked ground (left top), pebbles (right top), grass ground (left bottom), and pavements (right bottom)

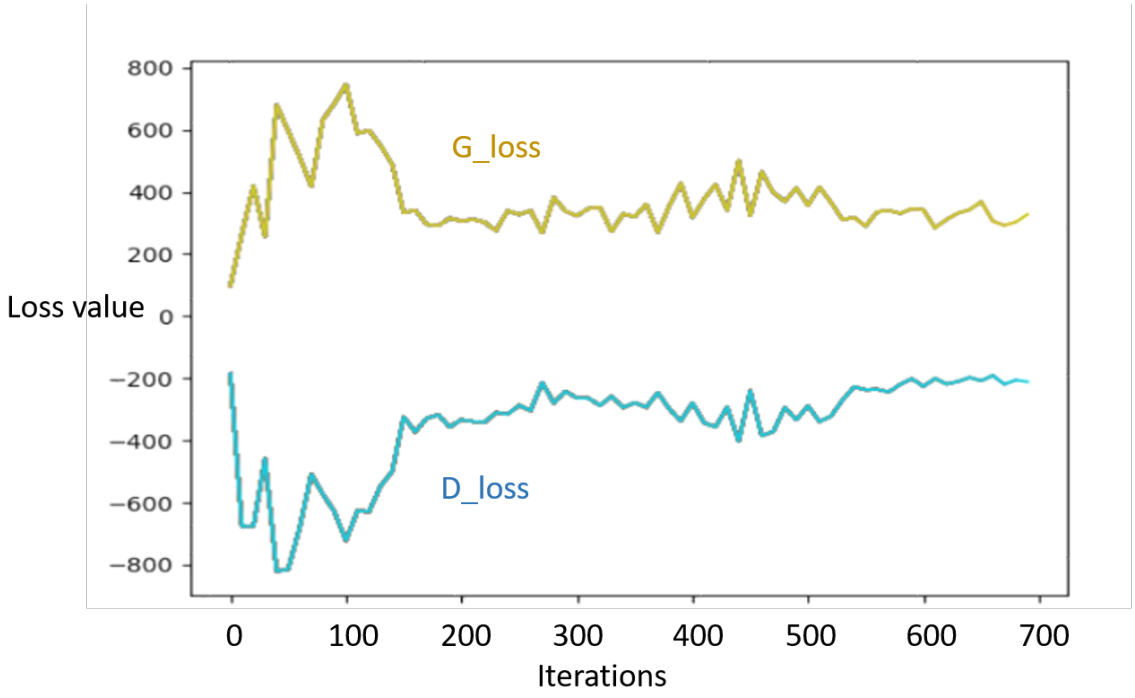

Figure 9: Loss values of generator and discriminator while training. We are only considering the absolute value of each of the loss value. the negative value is used just for visual aid. We want two lines to get closer to each other.

four words, I have regarded these words including the text ”grass”, ”moss”, ”forest”, ”fields”, ”plants”, ”bushes”, ”algae”, ”lawn”, ”leaves” as grass ground. For pebbles, I have regarded these words including the text ”cobble”, ”gravel”, ”rocks”, ”stones” as pebbles. For cracked ground, I have regarded these words including the text ”dry”, ”crack”, ”desert”, ”broken”, ”jagged”, ”fractured”, ”splintered”, ”chipped” as cracked ground. For pavement, I have regarded these words including the text ”rows”, ”tiles”, ”columns”, ”walls”, ”bricks” as pavement. I will be deciding that each evaluated image is ideally good and realistic if 50% of responses or more chose to say the above words in their descriptions and bad otherwise.

From Table 1, we can that 19 out of 20 (95%) evaluators regards this image as grass related textures. Thus, I have chosen this image as an ideally realistic texture. Similarly, Table 2 has shown that 17 out of 20 (85%) regards this image as pebbles, which is also chosen to be a ideal texture. However, for cracked ground which has 8 out of 20 (40%) and pavement which has 6 out of 20 (30%) are regarded as not realistic enough. As we can see in table 5, after computing all the scores and getting the average, the score tends to be 11.2 with (56%). Thus, the

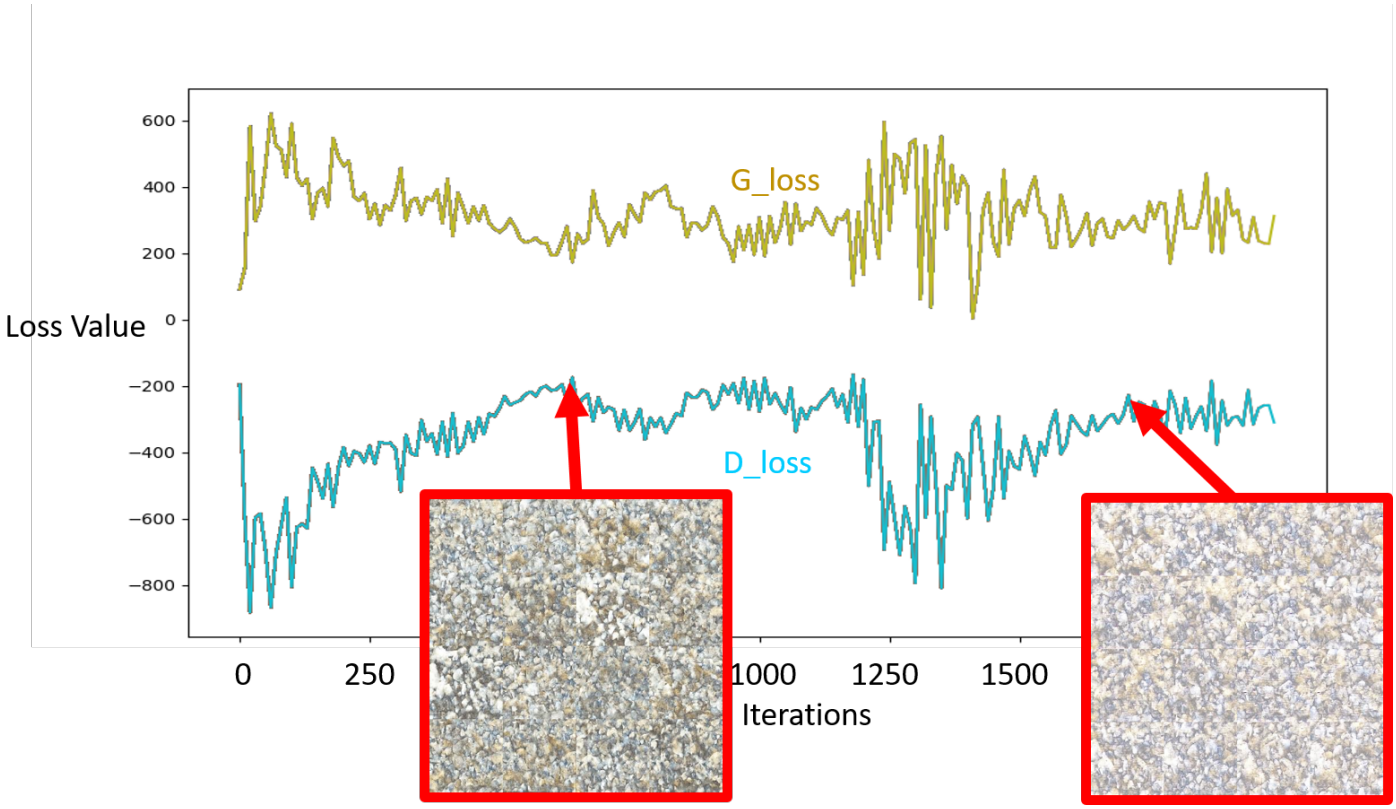

Figure 10: Loss values of discriminator and generator on the category of pebbles. As both networks are trained more and more, the loss values for the real images and the fake images become closer together. The resulting images on the left looks more like pebble than the right image which is brighter.

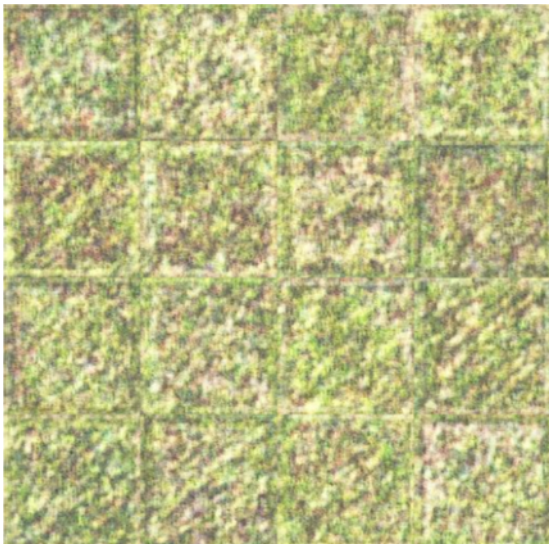

Figure 11: Generated grass

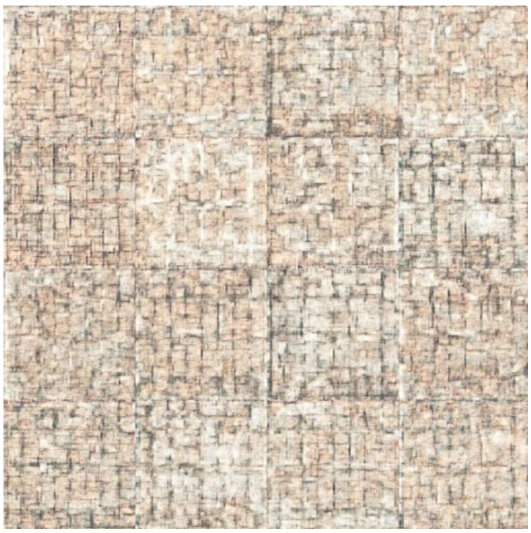

Figure 12: Generated pavement

Description of image!	Accounted as grass ground
wasabi	no
Grassy	yes
grass	yes
lawn	yes
Fairway green	yes
grassy	yes
Grassy field	yes
Grass	yes
16 squares of grass	yes
definitely grass	yes
grass	yes
Large Grass	yes
grass	yes
grass	yes
grass	yes
Grass	yes
grass, grassland	yes
lawn	yes
grass	yes
Grass	yes
	19/20

Table 1: Analyzed response table of grass ground (Figure 11)

Description of image!	Accounted as Pebbles
more rocks	yes
Rocky area but there are clouds in the way	yes
Rough terrain	no
gravel	yes
More gravel	yes
pebbly	yes
Sand and pebbles	yes
Pebbles	yes
16 grounds of rocks	yes
lots of rocks and sand	yes
Rock, minerals	yes
Larger Pebbles	yes
walkway rock	yes
little stones	yes
cross sign	no
Pebble	yes
city	no
sandy soil	no
rocky	yes
Pebbles	yes
	17/20

Table 2: Analyzed response table of Pebbles (Figure 13)

Description of image!	Accounted as cracked ground
rocks	no
Snowy mountain	no
rocks	no
cracked granite	yes
Charcoal	no
desert, parched	yes
Rocks	no
Cracked ground	yes
grounds of rocks	no
looks like rocks on dry soil almost desert like	yes
Marble	no
Dry Dessert	yes
marble	no
dry soil	yes
ice cube	no
Ground	no
stone, gravel	no
cliff face	no
cracked	yes
Dry Cracked Ground	yes
	8/20

Table 3: Analyzed response table of cracked ground (Figure 14)

Description of image!	Accounted as pavement
houses taken from above	no
Like a load of small cities	no
wood	no
abstract art	no
Wood	no
burlap	no
Mud Bricks	yes
Bricks	yes
bricks	yes
looks like the walls made up of brown blocks	yes
stones	no
Dry Soil	no
cut wood	no
aerial city view	no
wood	no
Tile	yes
city roads in europe	no
natural mosaic	no
dry	no
Pavement Tiles	yes
	6/20

Table 4: Analyzed response table of pavement(Figure 12)

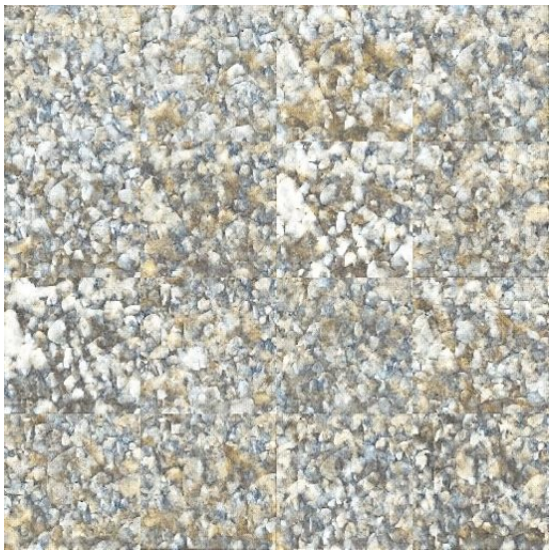

Figure 13: Generated pebbles

No.	Score of each image	Percentage	Maximum	Minimum
1	17/20	85%		
2	16/20	80%		
3	14/20	70%		
4	18/20	90%		
5	16/20	80%		
6	17/20	85%		
7	17/20(Table 2)	75%		
8	8/20	40%		
9	6/20	30%		
10	15/20	75%		
11	12/20	85%		
12	14/20	70%		
13	10/20	50%		
14	3/20	15%		yes
15	4/20	20%		
16	8/20(Table 3)	50%		
17	19/20 (Table 1)	95%	yes	
18	3/20	15%		yes
19	13/20	65%		
20	8/20	40%		
21	10/20	50%		
22	9/20	45%		
23	9/20	45%		
24	12/20	60%		
25	18/20	90%		
26	6/20 (Table 4)	30%		
27	6/20	30%		
28	16/20	80%		
29	7/20	35%		
30	5/20	25%		
Averages	Average Score= 11.2	Average % = 56%		

Table 5: Scores of each image

Figure 14: Generated pebbles

11 Conclusion

As I have asked in my research question: Can we create a Generative Adversarial Network that draw terrain textures? From the results of the experiment, it was successful enough to draw terrains from grass and pebbles due to the percentile that is greater than 75% shown in Table(1 and 2). However, for cracked ground and pavements, which have the percentile less than 50% (Table 3 and 4), it was not successful enough for evaluators to identify a certain type of ground. Finally, From Table 5, we can see the average of the percentages is 56%. Therefore, we can conclude that my GAN can render certain terrain textures such as Grass and pebbles but cannot render pavements and cracked ground.

For future work, there can be more improvements for the GAN I have developed for texture rendering, for example, I can try to create the generated image more and more realistic by getting more datasets. We can try to find the similar type of cracked ground or grass ground datasets to make a more realistic images. Due to the insufficient number of similar images, the evaluators seems to be unable to specifically identify a certain type ground. This is one of my limitations in my thesis. For example, one of the images form the training datasets, might be taken from a height of 1 meter and others might be taken from a different height. This is one of the issues wit image classification which is scale variation [4]. Thus, the sized of pebbles, cracks and bricks from pavements might be varied differently. If a dataset of similar type of terrain exists, the GAN could be improved for a better and more realistic images of terrain. Another limitation is that I can design the evaluation to be a check box rather than a text description which will greatly limits

the evaluators to answer the appointed image. This will also help me analyze the evaluations with convenience. Another way I could design the experiment is that, I can create a trivial question type for each image given the original image next to the generated one. Then, asking if the generated image look as realistic as the original image would help me evaluate the GAN as well.

11.1 Acknowledgement

I would like to thank Stanford course on Deep-learning that has helped me study about Convolution Neural Networks, Akshay Kashyap for referring me to this website and David Frey for keeping the reserved computer in the CROCHET lab.

References

[1] Grigory Antipov, Moez Baccouche, and Jean-Luc Dugelay. "Face Aging with Conditional Generative Adversarial Networks". In: (2017). URL: https://arxiv.org/pdf/1702.01983.pdf.

[2] Y Bengio. *Learning deep architectures for AI*. now Publishers Inc., 2009. ISBN: 78-1-60198-294-0.

[3] Gareth Bourn. *No-Man Sky*. 2017. URL: https://www.nomanssky.com.

[4] Csc-231n. *CS231n Convolutional Neural Networks for Visual Recognition*. 2017. URL: http://cs231n.github.io/classification/.

[5] Alexei A. Efros and William T. Freeman. "University of California, Berkeley". In: (2002). URL: https://people.eecs.berkeley.edu/~efros/research/quilting/quilting.pdf.

[6] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. "Texture Synthesis Using Convolutional Neural Networks". In: (2016). URL: https://papers.nips.cc/paper/5633-texture-synthesis-using-convolutional-neural-networks.pdf.

[7] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep sparse rectifier neural networks". In: (). URL: http://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf.

[8] Ian J. Goodfellow et al. "Generative Adversarial Nets". In: (2014). URL: https://arxiv.org/pdf/1406.2661.pdf.

[9] Ian J. Goodfellow et al. "Maxout Networks". In: (). URL: http://proceedings.mlr.press/v28/goodfellow13.pdf.

[10] Geoffrey Hinton et al. “Deep Neural Networks for Acoustic Modeling in Speech Recognition”. In: (2012). URL: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/38131.pdf.

[11] Kevin Jarrett et al. “What is the Best Multi-Stage Architecture for Object Recognition?” In: (). URL: http://yann.lecun.com/exdb/publis/pdf/jarrett-iccv-09.pdf.

[12] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. “Perceptual Losses for Real-Time Style Transfer and Super-Resolution”. In: (2016). URL: https://cs.stanford.edu/people/jcjohns/papers/eccv16/JohnsonECCV16.pdf.

[13] Hong K. *Artificial Neural Network (ANN)*. 2016. URL: http://www.bogotobogo.com/python/scikit-learn/Artificial-Neural-Network-ANN-1-Introduction.php.

[14] Siraj Raval. *Generative Adversarial Networks*. 2017. URL: https://github.com/llSourcell/Generative_Adversarial_networks_LIVE/blob/master/EZGAN.ipynb.

[15] *TensorFlow*. URL: https://www.tensorflow.org/.

[16] Dmitry Ulyanov et al. “Texture Networks: Feed-forward Synthesis of Textures and Stylized Images”. In: (2016). URL: https://arxiv.org/pdf/1603.03417.pdf.

[17] Wikipedia. *Neurone*. 2017.

Appendices

A GAN code

```
#Author Hein H. Aung
#Created on            – Jan 9th, 2018
#last Updated on       – Feb 28th, 2018

import os
import tensorflow as tf
import numpy as np
import random
import scipy.misc
```

```
from utils import *
import cv2
import matplotlib.pyplot as plt

HEIGHT = 128
WIDTH = 128
CHANNEL = 3

save_path = 'Cracked_generated5'
newtexture_path = './' + save_path

def lrelu(x, n, leak=0.2):
    return tf.maximum(x, leak * x, name=n)

def discriminator(input, is_train, reuse=False):
    filters = 64
    with tf.variable_scope('dis') as scope:
        if reuse:
            scope.reuse_variables()
        # 128 * 128 * 64
        conv1 = tf.layers.conv2d(input, filters, kernel_size=[4, 4], strides=[2, 2], paddi
                                 kernel_initializer=tf.truncated_normal_initializer(stddev
                                 name='conv1')
        bn1 = tf.contrib.layers.batch_norm(conv1, is_training = is_train, epsilon=1e-5, de
updates_collections=None, scope = 'bn1')
        act1 = lrelu(bn1, n='act1')

        # Flatten
        dim = int(np.prod(act1.get_shape()[1:]))
        fc1 = tf.reshape(act1, shape=[-1, dim], name='fc1')

        #flattened weight
```

```
        w2 = tf.get_variable('w2', shape=[fc1.shape[-1], 1], dtype=tf.float32,
                             initializer=tf.truncated_normal_initializer(stddev=0.02))
        #flattened bias
        b2 = tf.get_variable('b2', shape=[1], dtype=tf.float32,
                             initializer=tf.constant_initializer(0.0))

        #Activate
        logits = tf.add(tf.matmul(fc1, w2), b2, name='logits')

        return logits

def generator(input, random_dim, is_train, reuse=False):

    output_dim = CHANNEL
    with tf.variable_scope('gen') as scope:
        if reuse:
            scope.reuse_variables()
        #64 * 64 * 32
        w1 = tf.get_variable('w1', shape=[random_dim, 64 * 64 * 32], dtype=tf.float32, ini

        b1 = tf.get_variable('b1', shape=[32 * 64 * 64], dtype=tf.float32, initializer=tf.

        flat_conv1 = tf.add(tf.matmul(input, w1), b1, name='flat_conv1')
        # Flattened 64 * 64 * 32
        deconv1 = tf.reshape(flat_conv1, shape=[-1, 64, 64, 32], name='deconv1')
        bn1 = tf.contrib.layers.batch_norm(deconv1, is_training=is_train, epsilon=1e-5, de
updates_collections=None, scope='bn1')
        act1 = tf.nn.relu(bn1, name='act1')

        # 128 * 128 * 3
        conv2 = tf.layers.conv2d_transpose(act1, output_dim, kernel_size=[4, 4], strides=[
                                           kernel_initializer=tf.truncated_normal_initializ
```

```
                                        name='conv2')

        act2 = tf.nn.tanh(conv2, name='act2')
        return act2

BATCH_SIZE = 16


def resizer():
    current_dir = os.getcwd()

    texture_dir = os.path.join(current_dir, 'Crack')
    data = []
    for each in os.listdir(texture_dir):
        data.append(os.path.join(texture_dir,each))
    # print images
    images = tf.convert_to_tensor(data, dtype = tf.string)

    images_queue = tf.train.slice_input_producer([images])

    content = tf.read_file(images_queue[0])
    image = tf.image.decode_jpeg(content, channels = CHANNEL)
    image = tf.image.random_brightness(image, max_delta = 0.01)
    image = tf.image.random_contrast(image, lower = 0.9, upper = 1.1)

    image = tf.image.resize_images(image, [128, 128])
    image.set_shape([128, 128, 3])



    image = tf.cast(image, tf.float32)
    image = image / 255.0
```

```
    images_batch = tf.train.shuffle_batch([image], batch_size = BATCH_SIZE,
                                          num_threads = 4, capacity = 200 + 3* BATCH_SIZE,
                                          min_after_dequeue = 200)
    num_images = len(data)

    return images_batch, num_images

EPOCH = 700

def train():
    random_dim = 100


    with tf.variable_scope('input'):
        real_image = tf.placeholder(tf.float32, shape = [None, 128, 128, 3], name='real_ima
        random_input = tf.placeholder(tf.float32, shape=[None, random_dim], name='rand_inpu
        is_train = tf.placeholder(tf.bool, name='is_train')


    fake_image_loss = generator(random_input, random_dim, is_train)
    real_result_loss = discriminator(real_image, is_train)
    fake_result_loss = discriminator(fake_image_loss, is_train, reuse=True)

    D_loss = tf.reduce_mean(fake_result_loss) - tf.reduce_mean(real_result_loss)
    G_loss = -tf.reduce_mean(fake_result_loss)


    t_vars = tf.trainable_variables()
    d_vars = [var for var in t_vars if 'dis' in var.name]
    g_vars = [var for var in t_vars if 'gen' in var.name]

    trainer_d = tf.train.AdamOptimizer(learning_rate=1e-3).minimize(D_loss, var_list=d_vars
```

```
trainer_g = tf.train.AdamOptimizer(learning_rate=1e-3).minimize(G_loss, var_list=g_vars
# clip discriminator weights
dis_clip = [v.assign(tf.clip_by_value(v, -0.01, 0.01)) for v in d_vars]


batch_size = BATCH_SIZE
image_batch, samples_num = resizer()

batch_num = int(samples_num / batch_size)
sess = tf.Session()
saver = tf.train.Saver()
sess.run(tf.global_variables_initializer())
sess.run(tf.local_variables_initializer())

coord = tf.train.Coordinator()
threads = tf.train.start_queue_runners(sess=sess, coord=coord)
g_loss_list = []
d_loss_list = []
print 'batch size: %d' % (batch_size)
print 'batch num per epoch: %d' % (batch_num)
print 'epoch num: %d'%(EPOCH)
for i in range(EPOCH):
    for j in range(batch_num):
        d_iters = 5
        g_iters = 1

        train_noise = np.random.uniform(-1.0, 1.0, size=[batch_size, random_dim]).astyp
         # Update the discriminator
        for k in range(d_iters):
            train_image = sess.run(image_batch)
            sess.run(dis_clip)
```

```
        _, dLoss = sess.run([trainer_d, D_loss],
                            feed_dict={random_input: train_noise, real_image: train

    # Update the generator
    for k in range(g_iters):
        _, gLoss = sess.run([trainer_g, G_loss], feed_dict={random_input: train_noi
        print i

if i%10 == 0:
    # save images
    if not os.path.exists(newtexture_path):
        os.makedirs(newtexture_path)
    sample_noise = np.random.uniform(-1.0, 1.0, size=[batch_size, random_dim]).asty
    print_img = sess.run(fake_image_loss, feed_dict={random_input: sample_noise, is



    save_images(print_img, [4,4] ,newtexture_path + '/epoch' + str(i) + '.jpg')
    g_loss_list.append(gLoss)
    d_loss_list.append(dLoss)



    print('Iter: {}'.format(i))
    print('D loss: {:.4}'.format(dLoss))
    print('G_loss: {:.4}'.format(gLoss))
    print()

    plt.plot(xrange(len(g_loss_list)), g_loss_list)
    plt.plot(xrange(len(d_loss_list)), d_loss_list)
    plt.xlabel('Iterations')
    #plt.axis([0,5000,dLoss,gLoss])
```

```
    coord.request_stop()
    coord.join(threads)
    print (g_loss_list)
    print (d_loss_list)
    plt.show()

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