Toward Improving Procedural Terrain Generation With GANs

Toward Improving Procedural Terrain Generation with GANs

William Mattull

A Thesis in the Field of Information Technology

For the Degree of Master of Liberal Arts in

Extension Studies

Harvard University

December 2019

Abstract

The advance of Generative Adversarial Networks (GANs) are able to create new and meaningful outputs given arbitrary encodings for three dimensional terrain generation. Using a two-stage GAN network, this novel system takes the terrain heightmap as the first processing object, and then maps the real texture image onto the heightmap according to the learned network. The synthetic terrain image results perform well and are realistic. However, improvements can be made to generator stability during training.

Acknowledgements

I want to thank my professors and teaching fellows for taking the time to assist me throughout the years. I would also like to thank my wife and daughters for their unwavering support during this academic journey.

Contents

List of Figures

List of Tables

List of Equations

Chapter 1

Introduction

Procedural generation in video games is the algorithmic generation of synthetic image intended to increase replay value through interleaving the game play with elements of unpredictability. This is in contrast to the more traditional 'hand-crafted' generation of content, which is generally of higher quality but with the added expense of labor. One of the most well known examples of a video game that relies entirely on procedural generation is Minecraft by Mojang, shown by Figure 1.1.

Figure 1.1: Procedural generation example: Minecraft by Mojang.

One process widely used for generating for synthetic terrain in modern video games

has been to use a heightmap image paired with a texture map image. A heightmap image contains a single dimension (height) channel. The value of a pixel within the image is interpreted as a distance of displacement or "height" from the "bottom" of a surface. Collectively, the pixels form a $n \times n$ rectangular grid to render a raster (bitmap) image. As can been seen in Figure 1.2, visually a heightmap is viewed as a grayscale image, with black representing minimum height and white representing maximum height. The displacement of each pixel can be controlled to form the appearance of terrain at either a small or large scale. A terrain rendering algorithm then coverts the heightmap into a 3D mesh, which is then overlaid with a texture map to render complex terrain and other surfaces. Heightmaps are commonly used in Geographic Information Systems (GIS), where they are called digital elevation models.

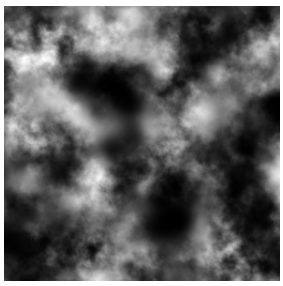

Figure 1.2: Example of a heightmap created with Terragen (A3r0, 2006a).

Although heightmaps can be crafted by hand using a raster graphics editor, for use with video game creation, it is more efficient to use a special terrain editor software such as Picogen. Using a graphics editor, one can visualize terrain in three dimensions while customizing the surface to mimic the desired synthetic terrain to generate heightmaps. Alternatively, terrain generation can be created using algorithms such as Diffusion-limited aggregation (Witten and Sander, 1981) or more recently using Simplex noise (Perlin, 2001).

In modern video games, the elements of a heightmap represent the coordinates of polygons in a mesh, as shown in Figure 1.3, or a rendered surface, as shown in Figure

1.4. A texture map image is applied (mapped) to the shape of the under laying polygon mesh or surface (Hvidsten, 2004). The texture map is a raster (bitmap) image created by computer graphics software or algorithmically created such as a procedural texture using Perlin noise (Perlin, 1985) or Simplex noise (Perlin, 2001).

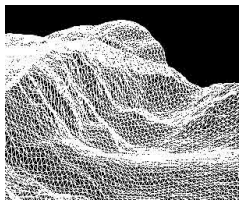

Figure 1.3: An example of polygon mesh created from a heightmap.

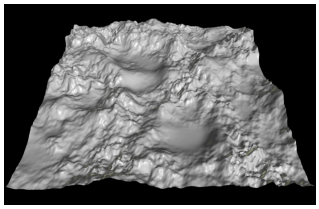

Figure 1.4: A version of the heightmap rendered with Anim8or and no textures (A3r0, 2006b).

These separate but related synthetic image creation processes, heightmap creation and texture map creation, can be brought together by using a deep learning approach known as Generative Adversarial Networks (GANs) (Goodfellow et al., 2014). Initial research conducted by Beckham and Pal (2017) showed synthetic terrain generation is indeed plausible by using a two-stage GAN with openly available satellite imagery from NASA. The process utilizes a DCGAN (Radford et al., 2015) for the heightmap generation, then proceeds with an image translation GAN, pix2pix (Isola et al., 2016), for texture map generation. Using updated techniques, it is possible to overcome issues experienced in the original research in the area of GAN training stability and improvements in the quality of output from the synthetic image generation process.

1.1 Prior Work

Three dimensional terrain modeling is active research topic not only in video games, but other areas such as scientific applications and training simulators. Algorithmic approaches to creating heightmaps and textures is called procedural generation. Procedural generation is widely used in video games (Smith, 2015), as it is used to create

large amounts of synthetic terrain content in a game. It should be acknowledged often times human generated content may be widely used in a video games as well, however this paper focuses mainly on terrain generated using algorithmic and deep learning techniques. Early approaches to creating heightmaps or textures using procedural generation generally utilized one of two approaches, fractal generation or noise generation.

1.1.1 Synthetic Terrain From Fractal Based Approaches

Creating heightmaps from fractals utilizes the properties of fractal self-similarity. That is fractal terrains appear and maintain their statistical properties even when inverted. Several techniques for fractal terrain synthesis exist. The original method for generating a fractional Brownian, fBm, surface uses Poisson faulting (Kirkby, 1983). This method involves application of random faults in a Gaussian distribution resulting in the statistically indistinguishable surfaces. This method is computationally intensive, resulting in an $\mathcal{O}(n^3)$ computational complexity.

Fournier et al. (1982) created what was to be become the essential algorithm for fractal terrain generation, called the midpoint displacement method. This algorithm starts with an initial square that is subdivided into smaller squares as can be seen in Figure 1.5.

Suppose there are four vertex points given by:

$$[x_0, y_0, f(x_0, y_0)], [x_1, y_0, f(x_1, y_0)], [x_0, y_1, f(x_0, y_1)], [x_1, y_1, f(x_1, y_1)]$$

Next, a new vertex is added $[x_{\frac{1}{2}}, y_{\frac{1}{2}}, f(x_{\frac{1}{2}}, y_{\frac{1}{2}})]$ to the center of the square such that:

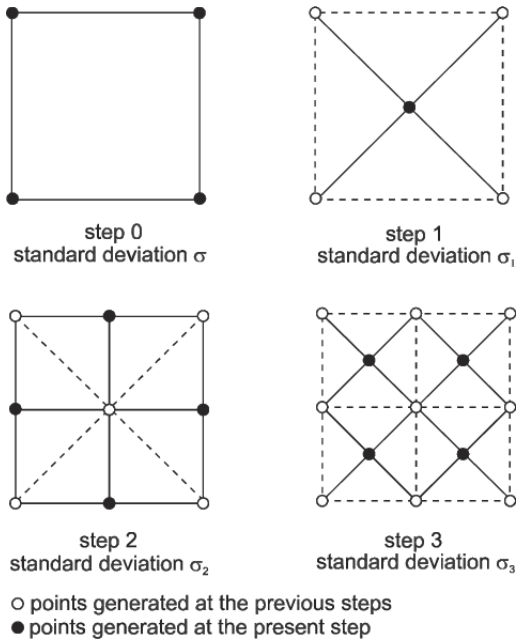

Figure 1.5: Iterations of Random Midpoint Displacement (Capasso, 2001).

$$\begin{aligned} x_{\frac{1}{2}} &= \frac{1}{2}(x_0 + x_1) \\ y_{\frac{1}{2}} &= \frac{1}{2}(y_0 + y_1) \\ f(x_{\frac{1}{2}}, y_{\frac{1}{2}}) &= \frac{1}{4}(f(x_0, y_0), f(x_1, y_0), f(x_0, y_1), f(x_1, y_1)) \end{aligned} \tag{1.1}$$

The center vertex is shifted in z-coordinate direction by random value denoted by δ. The algorithm runs recursively for each sub-square.

δ is generated within the definition of fBm, therefore δ_i is generated within the Gaussian Distribution, $N(\mu, \sigma^2)$. At the $i-th$ iteration σ^2 is modified such that:

$$\sigma_i^2 = \frac{1}{2^{2H(i+1)}}\sigma^2 \tag{1.2}$$

where H denotes Hurst exponent (Kirkby, 1983) ($1 \leq H \leq 2$).

In the second step, the midpoint of each edge of the initial square is calculated

by performing a transformation function, rotating the square 45° . Just as in step one, the same iteration is performed. To conclude the algorithm, the transformation returns to the original starting coordinates and continues with the recursion on four new squares. The recursion halts after a given number of iterations. An example of random midpoint displacement can be seen in Figure 1.6 as applies to a line.

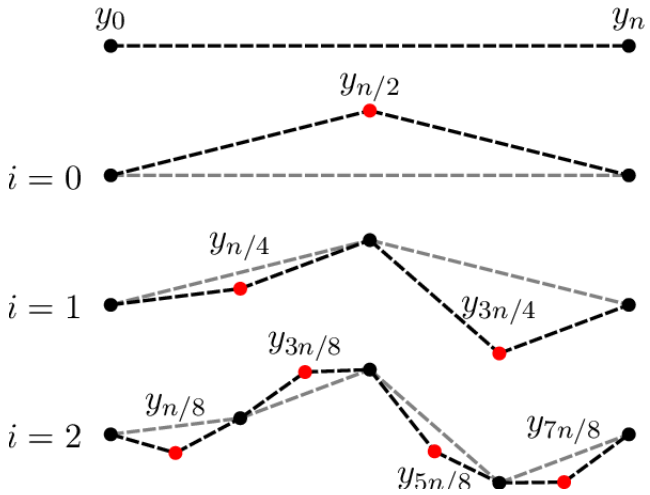

Figure 1.6: Random Midpoint Displacement applied to a line (Sund, 2014) using three iterations.

Diamond-square is an improvement on midpoint displacement method. The procedural generational approach diamond-square (Fournier et al., 1982), builds a 2D array of size $2^n + 1$ (5x5, 17x17, 33x33, etc.), then randomly generates terrain height from pre-seeded values in the four corners of the array. An example of the diamond-square algorithm applied to a 5×5 array can be seen shown in Figure 1.7. The diamond-square algorithm loops over progressively smaller step sizes, performing a 'square step' (steps a, c, e) and then a 'diamond step' (steps b, d) until each value in the array has been set, resulting in a heightmap. Even though diamond-square is an improvement on midpoint displacement method, the generated terrain still suffers from axis aligned ridges and is considered flawed by the authors.

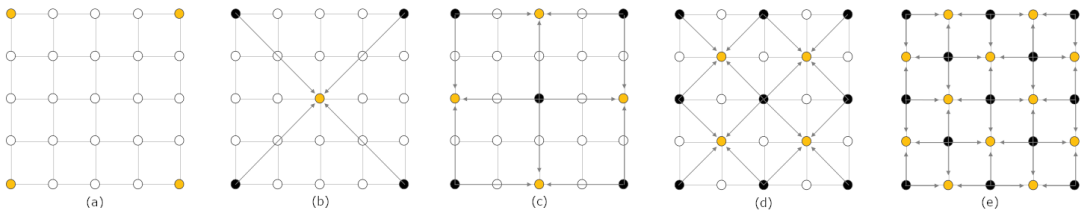

Figure 1.7: Steps involved in running the diamond-square algorithm on a 5×5 array (Ewin, 2015).

1.1.2 Synthetic Terrain from Noise Generation Based Approaches

Approaches using random noise generation have been adopted for synthetic terrain creation. Perlin (1985) introduced a noise generation algorithm, known as Perlin noise, differed from the the fractal based approaches. Perlin noise, also referred to as gradient noise, sets a pseudo-random gradient at regularly spaced intervals (points) in space, then interpolates a smoothing function between those points. As shown in Figure 1.8, to generate Perlin noise in one dimension, create a random gradient (or slope) for the noise function at each and every integer point, and set the function's value at each integer point to zero. For a given point somewhere between two integer points, the value is interpolated between two values.

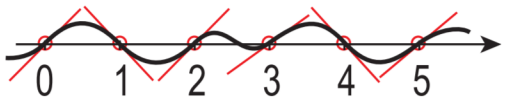

Figure 1.8: Perlin noise (gradient noise) example along one dimension.

Perlin (2001), improving upon his previous work, introduced the Simplex noise algorithm, which claims lower computational complexity and requires fewer multiplications. Processing utilizes a well-defined and continuous gradient, resulting in fewer visible directional artifacts. Simply speaking, the techniques are essentially a three dimensional mesh and when rendered produce terrain. These techniques are indeed

computationally fast, however the results produce simple terrains that are not of high realism compared to actual nature.

1.1.3 Synthetic Terrain from Generative Adversarial Networks (GANs)

An alternative method to software packages that utilize the previously mentioned algorithms, a Generative Adversarial Network (Goodfellow et al., 2014), a form of machine learning that employs two deep neural nets, a discriminator and generator, that work with and against each other during model training. GANs have the potential to produce high quality images. Beckham and Pal (2017) employed the use of two generators and two discriminators within a Generative Adversarial Network (GAN) to generate terrain. The method uses one generator to map noise to heightmaps, and the second generator to map from heightmap to textures. Likewise two discriminators are used $D(x)$ and $D(x, y)$. $D(x)$ computes the probability that x is a real heightmap, while $D(x, y)$ is the probability that (x, y) is a real heightmap/texture compared to real heightmap/fake texture. The authors suggest some improvements to the two stage method in order to achieve model stability and reduction of resulting artifacts within images.

The formulation for the two-stage GAN as proposed by Beckham and Pal (2017) is as follows:

Suppose $\mathbf{z}' \sim p(\mathbf{z})$ is a k-dimensional sample drawn from a prior distribution, $\mathbf{x}' = G_h(\mathbf{z}')$ the heightmap h, which is generated from $\mathbf{z}'$, and $\mathbf{y}' = G_t(\mathbf{x}')$ is the texture t, generated from the corresponding heightmap. This process can be theorized as being comprised of two GANs: the 'DCGAN' (Radford et al., 2015) which generates the heightmap from noise, and 'pix2pix' (Isola et al., 2016), which (informally) refers to conditional GANs for image-to-image translation. If we denote the DCGAN generator and discriminator as $G_h(\cdot)$ and $D_h(\cdot)$ respectively, then we can formulate the training

objective as:

$$\min_{G_h} \ell(D_h(G_h(\mathbf{z}')), 1) \tag{1.3}$$

Equation 1.1: The DCGAN Generator training objective.

$$\min_{D_h} \ell(D_h(\mathbf{x}), 1) + \ell(D_h(G_h(\mathbf{z}')), 0) \tag{1.4}$$

Equation 1.2: The DCGAN Discriminator training objective.

where ℓ is a GAN-specific loss, e.g. binary cross-entropy for the regular GAN formulation and squared error for LS-GAN (Mao et al., 2016). We can write similar equations for the pix2pix GAN, where now we have $G_t(\cdot)$ and $D_t(\cdot, \cdot)$ where instead of $\mathbf{x}$ and $\mathbf{x}'$ we have $\mathbf{y}$ (ground truth texture) and $\mathbf{y}'$ (generated texture) respectively. Note that the discriminator in this case, D_t, actually takes two arguments: either a real heightmap / real texture pair $(\mathbf{x}, \mathbf{y})$, or a real heightmap / generated texture pair $(\mathbf{x}, \mathbf{y}')$. Also note that for the pix2pix part of the network, we can also employ some form of pixel-wise reconstruction loss to prevent the generator from dropping modes. Therefore, we can write the training objectives for pix2pix as such:

$$\min_{G_t} \ell(D_t(\mathbf{x}, G_t(\mathbf{x}')), 1) + \lambda d(\mathbf{y}, G_t(\mathbf{x}')) \tag{1.5}$$

Equation 1.3: The pix2pix Generator training objective.

$$\min_{D_t} \ell(D_t(\mathbf{x}, \mathbf{y}), 1) + \ell(D_t(\mathbf{x}, G_t(\mathbf{x}')), 0) \tag{1.6}$$

Equation 1.4: The pix2pix Discriminator training objective.

where $d(\cdot, \cdot)$ can be some distance function such as L_1 or L_2 loss and λ is a hyperparameter denoting the relative strength of the reconstruction loss. Hyperparameter values are shown in Chapter 7, Results, for each experimental run.

1.2 Project Goals

We propose the following goals for the project:

1. Port the original project to use Keras and Tensorflow. The authors used the Keras, Lasagna, and Theano stack in the original project. Lasagna and Theano run exclusively on Linux. Replacing Lasagna and Theano enables the project to run on additional platforms and simplifies the architecture of the code by eliminating the use of Lasagna.

2. Include updated modeling techniques to DCGAN and pix2pix for quality improvements to generated images. For the DCGAN, this includes the introduction of a Gaussian input layer to make learning for discriminator more complex. Also updating the hidden layers to use LeakyReLU instead ReLU and replace the sigmoid function as the activation for the output with layer with the tanh function. The performance of the model with and without the addition of the Gaussian layer will be analyzed.

3. Provide utility notebooks in Jupyter that are used to manage image sets and explore the rendered images after training is complete. In addition, create a notebook that can be used to explorer the latent space used with the DCGAN.

Chapter 2

Requirements

The requirements to train a Generative Adversarial Networks (GANs) are similar to that of training any Deep Learning model. Generally there are considerations to be made for software and hardware. The requirements for the overall construction of the deep learning layer will be covered in the Design and Implementation chapters respectively.

2.1 Image(s) and Image Preproccessing Requirements

In accordance with the original Beckham and Pal (2017) paper, the performance of the two stage GAN model as presented requires two high resolution images, a heightmap and a texture (terrain) map, to be downloaded from NASA's Visible Earth Project:

- `https://eoimages.gsfc.nasa.gov/images/imagerecords/73000/73934/gebco_08_rev_elev_21600x10800.png`
- `https://eoimages.gsfc.nasa.gov/images/imagerecords/74000/74218/world.200412.3x21600x10800.jpg`

The images require further processing prior to introduction to the GANs. The DCGAN, will train on grayscale 256 x 256 pixel images with a single grayscale scale channel. The pix2pix GAN, requires an 256 x 256 image pair for image translation. The image to be translated, referred to as A, will be a grayscale heightmap of 1-channel, will be reshaped to a 3-channel image prior to translation. The second image of the pair, referred to as B, is a 256 x 256 color texture map of 3-channels. All images are placed into HDF5 binary data file, labeled textures or heightmaps as 8-bit unsigned integer (uint8) with a range of 0-255.

Since the hyperbolic tangent activation function is used as the output for both the DCGAN and pix2pix GAN, Radford et al. (2015) recommends scaling the 0-255 values. For this, a second HDF5 data file is created storing the scaled values. Additionally 90%/10% training/validation split is performed. The result of the split, data sets are stored as xt, yt, xv, and yv, where x represents the heightmaps and y represents the textures.

The original heightmap and texture images are cropped into 256 x 256 size images by starting in the upper left hand corner of the original image and then stepping the to the right by 50 pixels per crop until reaching the lower right hand corner. To assist in this process, an image prepossessing utility has been created as a Jupyter notebook for the project.

2.2 Software Requirements

The two main considerations when training a deep leaning model(s) is the choice of programming and deep learning framework. Beckham and Pal (2017) chose to use Python as the programming language and build the project using Theano, Lasange, and Keras. This project implementation also uses Python as the programming language. As outlined in the project goals, the project was reconstructed to use Tensor-

flow, while keeping the use of the Keras framework. Lasange is compatible with only Theano so it was not used in this project. To simplify package management with Python, the data science platform Anaconda was used.

A detailed listed of software packages along with version used in this project is listed in Figure 2.1 (Note: Installing Anaconda will install additional packages other than the ones listed below. Some packages will not be included with default Anaconda install and will be sourced separately, although most will be available using Anaconda's *conda* tool.):

Name	Version
blas	1.0
cudatoolkit	9.0
cudnn	7.6.0
cv2	3.3.1
graphviz	2.38
h5py	2.7.1
imageio	2.4.1
keras-applications	1.0.4
keras-base	2.2.2
keras-contrib	2.0.8
keras-gpu	2.2.2
keras-preprocessing	1.0.2
matplotlib	2.1.2
numpy	1.15.2
numpy-base	1.15.2
pillow	5.1.0
python	3.5.6
python-graphviz	0.10.1
scikit-image	0.14.0
scipy	1.1.0
tensorboard	1.10.0
tensorflow	1.10.0
tensorflow-base	1.10.0
tensorflow-estimator	1.13.0
tensorflow-gpu	1.10.0

Table 2.1: List of required Python modules.

Additional software requirements notes: A working driver to support at least one

GPU for training purposes.

2.3 Hardware Requirements

Essentially a GAN is represented by two deep neural networks (DNN). Just like DDNs, GANs require high amounts of computations. Training GANs on a CPU can be quite limiting. Modern day approaches use one or more graphics processing units (GPU) to process the repetitive calculations required for forward propagation and back propagation steps performed during training. The key hardware components as used in this project are listed in Figure 2.2.

Component	Type
CPU	Intel 8 core i7 2.6 GHz
GPU	NVIDIA GEFORCE GTX 960M
RAM	16 GB

Table 2.2: List of key hardware components used for this project.

2.4 System Requirements

Deep learning describes a particular configuration of an artificial neural network (ANN) architecture that has many 'hidden' or computational layers between the input neurons where data is presented for training or inference, and the output neuron layer where the numerical results of the neural network architecture can be read. In the case of GANs, there are usually two deep neural networks (DNNs) used, one for the generator and one for the discriminator. A conceptual overview of a deep learning system which is comprised of the compute, programming, and algorithms stack can be viewed in Figure 2.1.

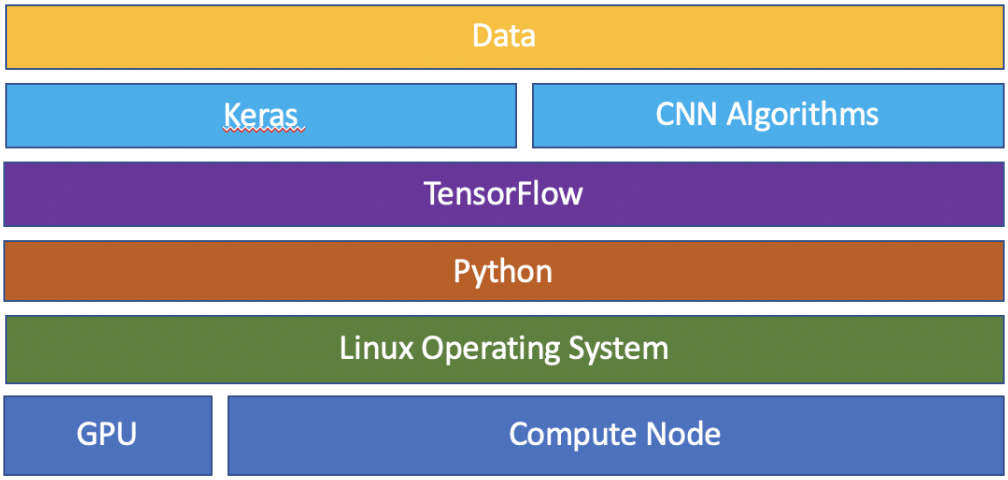

Figure 2.1: Compute Stack for a Deep Learning System.

The training for the DNNs is computationally intensive, and thanks to fairly recent introduction of Graphics Processing Units (GPUs) has allowed for more powerful DNNs to be built. At the compute layer, a GPU is able to make the intensive calculations that are required to build deep learning models. As the GPU have hundreds to thousands of cores available for compute, this allows for the ability to add many more hidden layers than was previously capable by using Central Processing Units alone. In effect, training of the DNN can be offloaded from the CPU to the more compute intensive GPU. GPUs are a key enabler of modern deep learning systems.

A deep learning system also requires flow control and mathematical libraries as well. Python is a widely adopted programming language, although not exclusive, within the deep/machine language community. A deep learning library or package, which is essentially mathematical routines and algorithm implementations, are used to compliment the programming language which allows for efficiencies in training deep learning models as one does not have to start from scratch and write all necessary code for a deep learning implementation.

Many of the components necessary for a deep learning system are in fact open source software and free to use. Tensorflow (Google Research, 2015) and Keras (Chollet et al., 2015) are two examples of open source deep learning frameworks. Tensorflow

provides deep/machine learning algorithms that can be implemented on large-scale distributed systems for model training. Keras provides a convenient syntactic framework that simplifies the coding structures required for deep learning and utilizes Tensorflow as the compute backend. Keras can also run on Theano (Theano Development Team, 2016), a different deep learning library.

Chapter 3

Design

3.1 Introduction

The overall architecture of generating synthetic terrian using GANs as proposed by Beckham and Pal (2017) is that of a two-stage GAN. This formulation utilizies two GANs, the DCGAN (Radford et al., 2015), which generates a heightmap from noise, and pix2pix (Isola et al., 2016), a type of GAN that is capable of performing image-to-image translation. In this chapter, the concept of the two-stage GAN is introduced along with GAN training process control. Additionally, the architecture of the DCGAN and pix2pix is reviewed. A detailed overview of GANs is included in Chapter 6. The implementation of the design is discussed in depth in Chapter 4.

3.2 Creating the two-stage GAN

In essence two GANS, DCGAN and pix2pix as shown in Figure 3.1, are integrated at the conclusion of training each GAN, where the DCGAN provides a synthetic heightmap to the pix2pix GAN that is translated into a synthetic texture map.

The process control of training each respective GAN, along with the ability to supply and change training and GAN related hyperparameters, will be accomplished

through a driver integrated into the code. The driver takes user supplied input from the command line interface (CLI) for parameters such as number of epochs, number of samples to include per batch, size of latent space, etc. The driver also controls the frequency of periodically exporting a trained generator model and tracks training analytics such as discriminator and generator loss. Periodically during the training process, at the conclusion of every 10 epochs, the trained generator will produce a set of images, using the models prediction function. The test images will be saved as an image matrix and used to assess the quality and diversity of images produced by the generator. At this interval, the current trained model used to produce the test images will be exported and saved to disk for later use.

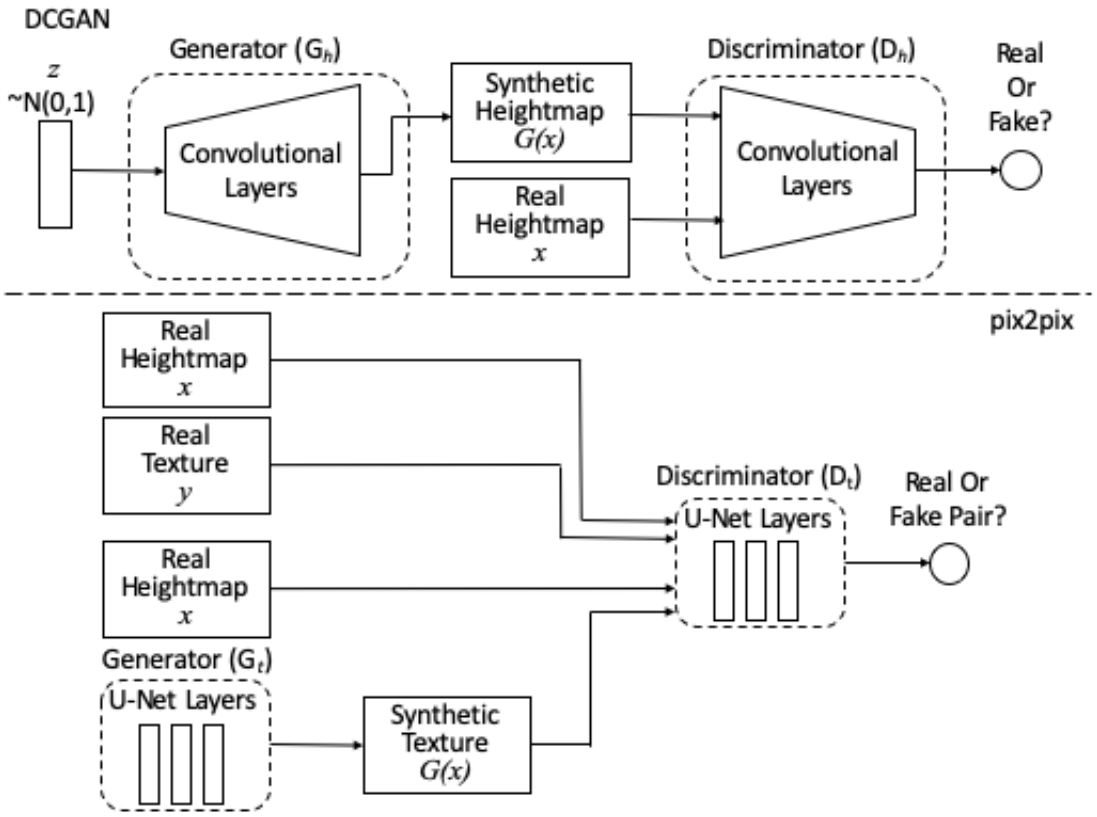

Figure 3.1: Two-stage GAN using DCGAN and pix2pix GAN.

3.3 DCGAN Model Design

Radford et al. (2015) presented the DCGAN model which is a deep convolutional neural network, shown in Figure 3.2, for generating images. This type of model utilizes a discriminator convolutional neural network model for classifying whether an given image is real or synthetic (fake) along with a generator model that uses inverse convolutional layers to transform an input to a full two-dimensional image. The generator and discriminator are considered to be standalone models. To make the training of the generator work correctly, a third model is introduced, a composite generator model. The composite generator model allows for generator model performance feedback from the discriminator which allows for generator model updates, but freezes any updates to the discriminator model to control overtraining on synthetic images from the generator.

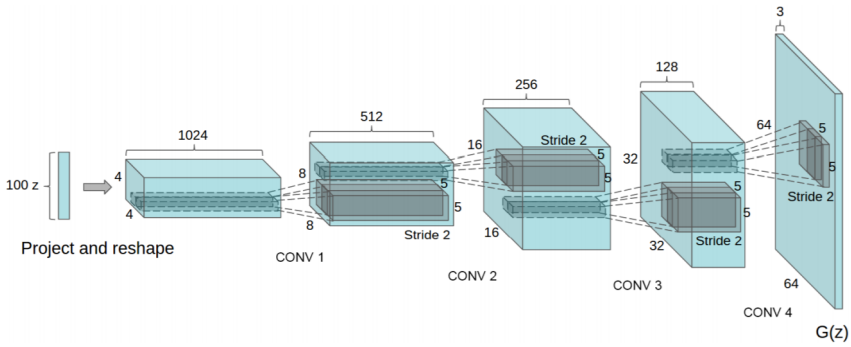

Figure 3.2: DCGAN generator used for LSUN scene modeling (Radford et al., 2015)

.

If the standalone generator model is trained successfully, as defined by model training success criteria, the standalone discriminator can be discarded, leaving just the standalone generator which can be use to generate new grayscale heightmaps.

3.3.1 Standalone Discriminator

The discriminator model, as shown in Listing 3.1, takes 256 x 256 sample image from the heightmap data set as input and outputs a classification prediction as to whether

the sample is real or fake. The discriminator model has six convolutional layers with Leaky ReLU as the activation function with an alpha of 0.2 and is used to predict whether the input sample is real or synthetic (fake). The model is trained to minimize the binary cross-entropy loss function which is appropriate for binary classification. Additionally, recommended best practices by Salimans et al. (2016) in defining the architecture of the discriminator model by using Dropout, Batch Normalization, and the Adam version of stochastic gradient decent with a learning rate of 0.0002 and a momentum of 0.7. To meet the project goal of improving training stability that was noticed by original paper's authors, an addition of a Gaussian noise layer after the input layer as suggested by Sønderby et al. (2016) to the original DCGAN architecture is discussed in Section 3.3.4 Improvements to DCGAN.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 256, 256, 8)       80
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 256, 256, 8)       0
_________________________________________________________________
dropout_1 (Dropout)          (None, 256, 256, 8)       0
_________________________________________________________________
average_pooling2d_1 (Average (None, 128, 128, 8)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 128, 128, 16)      1168
_________________________________________________________________
batch_normalization_1 (Batch (None, 128, 128, 16)      64
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 128, 128, 16)      0
```

```
_________________________________________________________________
dropout_2 (Dropout)          (None, 128, 128, 16)      0
_________________________________________________________________
average_pooling2d_2 (Average (None, 64, 64, 16)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 64, 64, 32)        4640
_________________________________________________________________
batch_normalization_2 (Batch (None, 64, 64, 32)        128
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 64, 64, 32)        0
_________________________________________________________________
dropout_3 (Dropout)          (None, 64, 64, 32)        0
_________________________________________________________________
average_pooling2d_3 (Average (None, 32, 32, 32)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 32, 32, 64)        18496
_________________________________________________________________
batch_normalization_3 (Batch (None, 32, 32, 64)        256
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 32, 32, 64)        0
_________________________________________________________________
dropout_4 (Dropout)          (None, 32, 32, 64)        0
_________________________________________________________________
average_pooling2d_4 (Average (None, 16, 16, 64)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 16, 16, 128)       73856
_________________________________________________________________
batch_normalization_4 (Batch (None, 16, 16, 128)       512
_________________________________________________________________
```

```
leaky_re_lu_5 (LeakyReLU) (None, 16, 16, 128)    0
_________________________________________________________________
dropout_5 (Dropout)       (None, 16, 16, 128)    0
_________________________________________________________________
average_pooling2d_5 (Average (None, 8, 8, 128)   0
_________________________________________________________________
conv2d_6 (Conv2D)         (None, 8, 8, 256)      295168
_________________________________________________________________
batch_normalization_5 (Batch (None, 8, 8, 256)   1024
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU) (None, 8, 8, 256)      0
_________________________________________________________________
dropout_6 (Dropout)       (None, 8, 8, 256)      0
_________________________________________________________________
average_pooling2d_6 (Average (None, 4, 4, 256)   0
_________________________________________________________________
flatten_1 (Flatten)       (None, 4096)           0
_________________________________________________________________
dense_1 (Dense)           (None, 128)            524416
_________________________________________________________________
leaky_re_lu_7 (LeakyReLU) (None, 128)            0
_________________________________________________________________
dense_2 (Dense)           (None, 1)              129
=================================================================
Total params: 919,937
Trainable params: 918,945
Non-trainable params: 992
```

Listing 3.1: Baseline DCGAN standalone discriminator.

3.3.2 Standalone Generator

The generator model, as shown in Listing 3.2, is responsible for creating new synthetic but plausible images as heightmaps. This is accomplished by taking a point from the latent space as input and outputting a grayscale image that represents a heightmap. The latent space is an arbitrarily vector space of Gaussian-distributed values, in this case, of 250 dimensions. The sampled vector from the latent space has no intrinsic meaning, however the generator will assign meaning to the latent points during the training phase of the network. After training completes, the latent vector space represents a compressed representation of the output image, that consequently only the generator knows.

Developing a generator model requires a transformation of the latent space with 250 dimensions to to a 2D array with 256 x 256 or 65,536 values. A proven approach is to create a Dense layer as the first hidden layer that has enough nodes to represent a low-resolution version of the output image. To determine the starting input size to the first layer, repeatedly half the desired output image size of 256 x 256 until a small pixel value is reached, in this case 4 x 4 or 16 nodes, which is one sixty-fourth of the original pixel size.

This would produce a single 4 x 4 low resolution image. In a reversal of steps from the pattern in convolutional neural networks, where many parallel filters result in multiple parallel activation maps or feature maps, would be many parallel versions of the output with different learned features that can be collapsed in the output layer into a final image. Thus the first hidden layer for the generator, the dense layer needs enough nodes for multiple low-resolution versions of the output image, such as 128.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
reshape_1 (Reshape)          (None, 1, 1, 250)         0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 4, 4, 256)         1024256
_________________________________________________________________
leaky_re_lu_8 (LeakyReLU)    (None, 4, 4, 256)         0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 4, 4, 256)         1048832
_________________________________________________________________
batch_normalization_6 (Batch (None, 4, 4, 256)         1024
_________________________________________________________________
leaky_re_lu_9 (LeakyReLU)    (None, 4, 4, 256)         0
_________________________________________________________________
up_sampling2d_1 (UpSampling2 (None, 8, 8, 256)         0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 8, 8, 128)         524416
_________________________________________________________________
batch_normalization_7 (Batch (None, 8, 8, 128)         512
_________________________________________________________________
leaky_re_lu_10 (LeakyReLU)   (None, 8, 8, 128)         0
_________________________________________________________________
up_sampling2d_2 (UpSampling2 (None, 16, 16, 128)       0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 16, 16, 64)        73792
_________________________________________________________________
batch_normalization_8 (Batch (None, 16, 16, 64)        256
```

```
_________________________________________________________________
leaky_re_lu_11 (LeakyReLU) (None, 16, 16, 64)     0
_________________________________________________________________
up_sampling2d_3 (UpSampling2 (None, 32, 32, 64)   0
_________________________________________________________________
conv2d_10 (Conv2D)         (None, 32, 32, 32)     18464
_________________________________________________________________
batch_normalization_9 (Batch (None, 32, 32, 32)   128
_________________________________________________________________
leaky_re_lu_12 (LeakyReLU) (None, 32, 32, 32)     0
_________________________________________________________________
up_sampling2d_4 (UpSampling2 (None, 64, 64, 32)   0
_________________________________________________________________
conv2d_11 (Conv2D)         (None, 64, 64, 16)     4624
_________________________________________________________________
batch_normalization_10 (Batc (None, 64, 64, 16)   64
_________________________________________________________________
leaky_re_lu_13 (LeakyReLU) (None, 64, 64, 16)     0
_________________________________________________________________
up_sampling2d_5 (UpSampling2 (None, 128, 128, 16) 0
_________________________________________________________________
conv2d_12 (Conv2D)         (None, 128, 128, 8)    1160
_________________________________________________________________
leaky_re_lu_14 (LeakyReLU) (None, 128, 128, 8)    0
_________________________________________________________________
up_sampling2d_6 (UpSampling2 (None, 256, 256, 8)  0
_________________________________________________________________
conv2d_13 (Conv2D)         (None, 256, 256, 1)    73
_________________________________________________________________
```

```
activation_1 (Activation) (None, 256, 256, 1)     0
=================================================================
Total params: 2,697,601
Trainable params: 2,696,609
Non-trainable params: 992
_________________________________________________________________
```

Listing 3.2: DCGAN standalone generator for 256 x 256 1-channel image generation.

3.3.3 Composite model

The weights in the generator model are updated based on the performance of the accuracy of the discriminator model. When the discriminator performs well at detecting synthetic images, the generator is updated more often, and when the discriminator model is performing poorly when detecting synthetic images, the generator is updated less frequently. This back and forth between the generator and the discriminator defines the adversarial relationship between the two models. A simple approach to implement this adversarial training approach using the Keras API (further defined in Chapter 4) is to create a new composite model that combines both the generator and discriminator models. More specifically, a new GAN model is defined that combines the generator and discriminator such that the generator receives as input random points in the latent space and generates samples that are fed into the discriminator model directly, classified, and the output of the combined model is used to update the model weights of the generator.

This new composite model exists only as a new logical model that uses the layers and weights as defined by the standalone generator and the standalone discriminator. This composite model exists to provide the generator with performance feedback of the discriminator model on synthetic images, adjusting the generator weights accordingly. As such, the discriminator layers are marked as not trainable when it is part

of the composite GAN model, which prevents the discriminator weights from being updated and therefore overtrained on fake examples. The standalone discriminator is trained outside of the composite GAN model.

When training the generator via this logical composite model, the generator will attempt to fool the discriminator by generating synthetic images and apply a class label marked as real ($class = 1$). The discriminator will then classify the synthetic samples as not real ($class = 0$) or a having a low probability of being real (~ 0.5). In this case, the backpropagation process used to update the generator model weights reacts to this large error and will then update the generator model weights to correct for this error. This results, in theory, to improve the generator's ability to output synthetic images that will fool the discriminator into classifying the synthetic image as real ($class = 1$).

3.3.4 Improvements to DCGAN

- **Gaussian Layer for Discriminator:** GANs can suffer from training instability and several training tips have been suggested from Salimans et al. (2016) and Radford et al. (2015). There is a belief that the idealized GAN and what is present in reality are disconnected with respected to the underlying distributions of the generator (synthetic) data and real data, especially early on in training. Sønderby et al. (2016) suggests adding *instance noise* in the form of a Gaussian layer to the discriminator. By adding a noise layer the discriminator is less prone to overfitting because it has a wider training distribution. For this project, as shown in Listing 3.3, a Gaussian layer is added after the input layer, prior to the first convolutional layer in the discriminator model of the DCGAN.

```
Layer (type)                 Output Shape              Param #
=================================================================
gaussian_noise_1 (GaussianNo (None, 256, 256, 1)  0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 256, 256, 8)     80
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 256, 256, 8)     0
_________________________________________________________________
dropout_1 (Dropout)          (None, 256, 256, 8)     0
_________________________________________________________________
average_pooling2d_1 (Average (None, 128, 128, 8)  0
_________________________________________________________________
...
_________________________________________________________________
dense_1 (Dense)              (None, 128)             524416
_________________________________________________________________
leaky_re_lu_7 (LeakyReLU)    (None, 128)             0
_________________________________________________________________
dense_2 (Dense)              (None, 1)               129
=================================================================
Total params: 919,937
Trainable params: 918,945
Non-trainable params: 992
_________________________________________________________________
```

Listing 3.3: Input and output lyers of the DCGAN standalone discriminator with Gaussian Noise.

3.4 pix2pix Model Design

The second stage of the synthesis terrain generation process is the use of a GAN that performs image-to-image translation. At this stage, both synthetically generated and real heightmaps will be translated into a synthetic, hopefully realistic, terrain map. So called image-to-image translation, also known as conditional image synthesis, consists of synthesizing an image based on another image. Image-to-image is a complex task, since a source image can have multiple possible translations, requiring the model and a loss function that are able to choose between these possible translations instead of picking a single one. Using a GAN framework for image-to-image translation can alleviate these issues by one: allowing the generator to sample from the underlying distribution of the latent vector and use it as an extra condition to disambiguate between possible translations; and two: avoid hand-engineered features that represent characteristics in the images for translation. Finally the GAN framework does not require a defined loss function for desirable translation characteristics.

To accomplish the task of image-to-image translation, the pix2pix GAN (Isola et al., 2016) will be used. pix2pix is comprised of the U-Net architecture (Ronneberger et al., 2015) for the generator and a PatchGAN (Isola et al., 2016) as the discriminator. The U-Net architecture that is used for the generator, includes a skip connection between layers of the generator that are increasingly far from one another. The PatchGAN differs from other discriminator architectures by producing multiple output values per image.

The layers used in pix2pix models are the comprised of two main components: the first component is an encoding block, which is a stack of convolutions with batch normalization and a leaky ReLU non-linearity; the other is the decoding component, which is a stack of upsampled convolutions with batch normalization and leaky ReLU non-linearity. The decoding component has a skip connection that concatenates a skip input to the layers' output.

3.4.1 Standalone Discriminator

As shown in listing 3.4, the discriminator used in pix2pix is a PatchGAN (Isola et al., 2016). Unlike other discriminators that output a single value per image, the discriminator in the PatchGAN outputs multiple values per image, where each value corresponds to a patch on the image. This is achieved by stacking convolutions until the desired number of patches is reached. Upon reaching the desired number of patches, the projected output of that convolution to a single channel to obtain a single value per patch is made. The single value per patch represents the output of the discriminator with respect to that patch.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 256, 256, 3) 0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 128, 128, 64) 3136
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 128, 128, 64) 0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 64, 64, 128) 131200
_________________________________________________________________
batch_normalization_1 (BatchNor (None, 64, 64, 128) 512
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 64, 64, 128) 0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 32, 32, 256) 524544
_________________________________________________________________
batch_normalization_2 (BatchNor (None, 32, 32, 256) 1024
```

```
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 32, 32, 256) 0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 16, 16, 512) 2097664
_________________________________________________________________
batch_normalization_3 (BatchNor (None, 16, 16, 512) 2048
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 16, 16, 512) 0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 8, 8, 512)  4194816
_________________________________________________________________
batch_normalization_4 (BatchNor (None, 8, 8, 512) 2048
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 8, 8, 512)  0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 4, 4, 512)  4194816
_________________________________________________________________
batch_normalization_5 (BatchNor (None, 4, 4, 512) 2048
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)    (None, 4, 4, 512)  0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 2, 2, 512)  4194816
_________________________________________________________________
batch_normalization_6 (BatchNor (None, 2, 2, 512) 2048
_________________________________________________________________
leaky_re_lu_7 (LeakyReLU)    (None, 2, 2, 512)  0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 1, 1, 512)  4194816
_________________________________________________________________
```

```
activation_1 (Activation)    (None, 1, 1, 512)  0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTrans (None, 2, 2, 512) 4194816
_________________________________________________________________
batch_normalization_7 (BatchNor (None, 2, 2, 512) 2048
_________________________________________________________________
dropout_1 (Dropout)          (None, 2, 2, 512)  0
_________________________________________________________________
concatenate_1 (Concatenate)  (None, 2, 2, 1024) 0
_________________________________________________________________
activation_2 (Activation)    (None, 2, 2, 1024) 0
_________________________________________________________________
conv2d_transpose_2 (Conv2DTrans (None, 4, 4, 512) 8389120
_________________________________________________________________
batch_normalization_8 (BatchNor (None, 4, 4, 512) 2048
_________________________________________________________________
dropout_2 (Dropout)          (None, 4, 4, 512)  0
_________________________________________________________________
concatenate_2 (Concatenate)  (None, 4, 4, 1024) 0
_________________________________________________________________
activation_3 (Activation)    (None, 4, 4, 1024) 0
_________________________________________________________________
conv2d_transpose_3 (Conv2DTrans (None, 8, 8, 512) 8389120
_________________________________________________________________
batch_normalization_9 (BatchNor (None, 8, 8, 512) 2048
_________________________________________________________________
dropout_3 (Dropout)          (None, 8, 8, 512)  0
_________________________________________________________________
concatenate_3 (Concatenate)  (None, 8, 8, 1024) 0
```

```
____________________________________________________________
activation_4 (Activation)    (None, 8, 8, 1024) 0
____________________________________________________________
conv2d_transpose_4 (Conv2DTrans (None, 16, 16, 512) 8389120
____________________________________________________________
batch_normalization_10 (BatchNo (None, 16, 16, 512) 2048
____________________________________________________________
concatenate_4 (Concatenate)  (None, 16, 16, 1024) 0
____________________________________________________________
activation_5 (Activation)    (None, 16, 16, 1024) 0
____________________________________________________________
conv2d_transpose_5 (Conv2DTrans (None, 32, 32, 256) 4194560
____________________________________________________________
batch_normalization_11 (BatchNo (None, 32, 32, 256) 1024
____________________________________________________________
concatenate_5 (Concatenate)  (None, 32, 32, 512) 0
____________________________________________________________
activation_6 (Activation)    (None, 32, 32, 512) 0
____________________________________________________________
conv2d_transpose_6 (Conv2DTrans (None, 64, 64, 128) 1048704
____________________________________________________________
batch_normalization_12 (BatchNo (None, 64, 64, 128) 512
____________________________________________________________
concatenate_6 (Concatenate)  (None, 64, 64, 256) 0
____________________________________________________________
activation_7 (Activation)    (None, 64, 64, 256) 0
____________________________________________________________
conv2d_transpose_7 (Conv2DTrans (None, 128, 128, 64) 262208
____________________________________________________________
```

```
batch_normalization_13 (BatchNo (None, 128, 128, 64) 256

_________________________________________________________________

concatenate_7 (Concatenate)  (None, 128, 128, 128 0

_________________________________________________________________

activation_8 (Activation)    (None, 128, 128, 128 0

_________________________________________________________________

conv2d_transpose_8 (Conv2DTrans (None, 256, 256, 3) 6147

_________________________________________________________________

activation_9 (Activation)    (None, 256, 256, 3) 0

=================================================================

Total params: 54,429,315

Trainable params: 54,419,459

Non-trainable params: 9,856

_________________________________________________________________
```

Listing 3.4: pix2pix standalone discriminator.

3.4.2 Standalone Generator

The generator used in pix2pix as shown in listing 3.5, is a U-Net Generator, which consists of a stack of encoding blocks followed by a stack of decoding blocks.

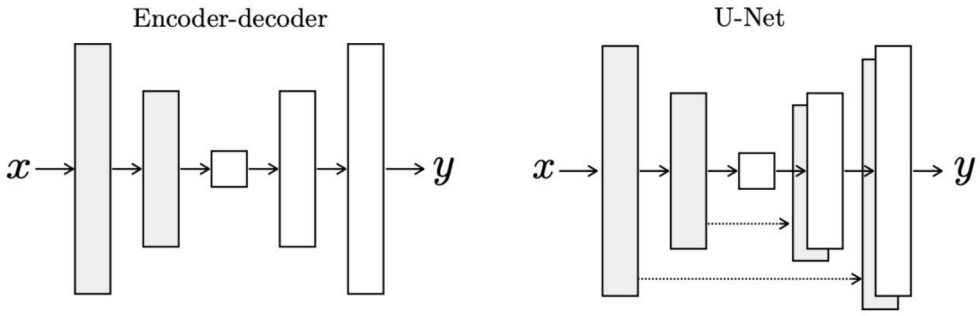

Figure 3.3: U-Net features an Encoder-decoder structure with skip connections that link both sides of the architecture.

.

Each encoding block is connected to a decoding block by skip connections between them, through concatenation. In the U-Net architecture shown in Figure 3.3, skip connections are created by going through encoding layers from first to last and adding a skip connection between that layer and the farthest decoding layer that does not have a connection. The U-Net name comes from the fact that the distance between layers diminishes as we move through the encoding layers, which resembles a U-shape.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 256, 256, 3) 0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 128, 128, 64) 3136
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 128, 128, 64) 0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 64, 64, 128) 131200
_________________________________________________________________
batch_normalization_1 (BatchNor (None, 64, 64, 128) 512
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 64, 64, 128) 0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 32, 32, 256) 524544
_________________________________________________________________
batch_normalization_2 (BatchNor (None, 32, 32, 256) 1024
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 32, 32, 256) 0
_________________________________________________________________
```

```
conv2d_4 (Conv2D)            (None, 16, 16, 512) 2097664
_________________________________________________________________
batch_normalization_3 (BatchNor (None, 16, 16, 512) 2048
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 16, 16, 512) 0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 8, 8, 512)  4194816
_________________________________________________________________
batch_normalization_4 (BatchNor (None, 8, 8, 512) 2048
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 8, 8, 512)  0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 4, 4, 512)  4194816
_________________________________________________________________
batch_normalization_5 (BatchNor (None, 4, 4, 512) 2048
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)    (None, 4, 4, 512)  0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 2, 2, 512)  4194816
_________________________________________________________________
batch_normalization_6 (BatchNor (None, 2, 2, 512) 2048
_________________________________________________________________
leaky_re_lu_7 (LeakyReLU)    (None, 2, 2, 512)  0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 1, 1, 512)  4194816
_________________________________________________________________
activation_1 (Activation)    (None, 1, 1, 512)  0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTrans (None, 2, 2, 512) 4194816
```

```
_________________________________________________________________
batch_normalization_7 (BatchNor (None, 2, 2, 512) 2048
_________________________________________________________________
dropout_1 (Dropout)          (None, 2, 2, 512)  0
_________________________________________________________________
concatenate_1 (Concatenate)  (None, 2, 2, 1024) 0
_________________________________________________________________
activation_2 (Activation)    (None, 2, 2, 1024) 0
_________________________________________________________________
conv2d_transpose_2 (Conv2DTrans (None, 4, 4, 512) 8389120
_________________________________________________________________
batch_normalization_8 (BatchNor (None, 4, 4, 512) 2048
_________________________________________________________________
dropout_2 (Dropout)          (None, 4, 4, 512)  0
_________________________________________________________________
concatenate_2 (Concatenate)  (None, 4, 4, 1024) 0
_________________________________________________________________
activation_3 (Activation)    (None, 4, 4, 1024) 0
_________________________________________________________________
conv2d_transpose_3 (Conv2DTrans (None, 8, 8, 512) 8389120
_________________________________________________________________
batch_normalization_9 (BatchNor (None, 8, 8, 512) 2048
_________________________________________________________________
dropout_3 (Dropout)          (None, 8, 8, 512)  0
_________________________________________________________________
concatenate_3 (Concatenate)  (None, 8, 8, 1024) 0
_________________________________________________________________
activation_4 (Activation)    (None, 8, 8, 1024) 0
_________________________________________________________________
```

```
conv2d_transpose_4 (Conv2DTrans (None, 16, 16, 512) 8389120
_________________________________________________________________
batch_normalization_10 (BatchNo (None, 16, 16, 512) 2048
_________________________________________________________________
concatenate_4 (Concatenate)  (None, 16, 16, 1024) 0
_________________________________________________________________
activation_5 (Activation)    (None, 16, 16, 1024) 0
_________________________________________________________________
conv2d_transpose_5 (Conv2DTrans (None, 32, 32, 256) 4194560
_________________________________________________________________
batch_normalization_11 (BatchNo (None, 32, 32, 256) 1024
_________________________________________________________________
concatenate_5 (Concatenate)  (None, 32, 32, 512) 0
_________________________________________________________________
activation_6 (Activation)    (None, 32, 32, 512) 0
_________________________________________________________________
conv2d_transpose_6 (Conv2DTrans (None, 64, 64, 128) 1048704
_________________________________________________________________
batch_normalization_12 (BatchNo (None, 64, 64, 128) 512
_________________________________________________________________
concatenate_6 (Concatenate)  (None, 64, 64, 256) 0
_________________________________________________________________
activation_7 (Activation)    (None, 64, 64, 256) 0
_________________________________________________________________
conv2d_transpose_7 (Conv2DTrans (None, 128, 128, 64) 262208
_________________________________________________________________
batch_normalization_13 (BatchNo (None, 128, 128, 64) 256
_________________________________________________________________
concatenate_7 (Concatenate)  (None, 128, 128, 128 0
```

```
_________________________________________________________________
activation_8 (Activation)    (None, 128, 128, 128 0
_________________________________________________________________
conv2d_transpose_8 (Conv2DTrans (None, 256, 256, 3) 6147
_________________________________________________________________
activation_9 (Activation)    (None, 256, 256, 3) 0
=================================================================
Total params: 54,429,315
Trainable params: 54,419,459
Non-trainable params: 9,856
_________________________________________________________________
```

Listing 3.5: pix2pix standalone generator.

3.5 Generating Heightmaps and Texture Maps

One of the core ideas behind using GANs is the adversarial nature between the GAN's generator and discriminator. The discriminators main goal is to minimize the success of the generator, by detecting a synthetic image produced by the generator. Conversely the goal of the generator is to maximize the incorrect classification of synthetic images by the discriminator. As training proceeds, hopefully, the quality and diversity of images produced by the generator improves to an acceptable level quality. Assuming this has been achieved, the trained generator model can be exported (saved) for reuse, preserving the weights of the deep nerual network, and the discriminator can be discarded. The saved generator model can then be used to generate synthetic images. During the training process for both the DCGAN and pix2pix GANs, trained models will be exported at a periodic interval, such as every ten epochs. Using a Jupyter notebook, trained generator models can be imported and used to generate new synthetic images.

In the case of the DCGAN, values are sampled from a Gaussian distribution from the latent space, z', and supplied to the generator, G_h, to generate new images. Thus $G_h : z' \mapsto x'$, where x' is the resulting image, a heightmap. Since the generator gives meaning to values from the latent space to generate images, it can be useful to explore the latent space by systematically sending values and then evaluate the resulting generated image. A utility for exploring the latent space of the generator to produce and evaluate images will be provided as a Jupyter notebook. This is covered is detail in Chapter 6.

In the case if of pix2pix, which is an image translation GAN, new images are generated not by sending values from a latent space, but rather from other images. For this project, supplied heightmaps from the DCGAN generator G_h for example, are translated into a texture map. The set of heightmaps to be translated are referred to as set A, and the resulting translation to texture maps as set B. Another way to state the process is image A is mapped to image B using the GAN's trained generator, such that $G_t : A \mapsto B$, where G_t is the generator from the pix2pix GAN. The process can also work in reverse $G_t : B \mapsto A$, where images from set B can be translated to set image set A. A Jupyter notebook with accommodating functionality for pix2pix translation will be provided.

Chapter 4

Implementation

The overall all design of the two-stage GAN was presented in the Design chapter (Chapter 3). This chapter describes how to create the data set, GAN implementation using the Keras Framework, helper functions, and the project folder structure. After discussing the data set and introducing key aspects of the Keras API, the remainder of the chapter focuses on key implementation details of GANs using Keras. A complete listing of the code used in the project can be found in Appendix B.

4.1 Creating the data set

The data used in the project is supplied by the NASA Visible Earth project using two extremely high-resolution 10800 x 21600 pixel size images. In order for the data set to be utilized by the GANs it must be processed into a usable format that consists of cropped 256 x 256 pixel size images. Furthermore, since the project utilizes two images of equal height and width, one that of a heightmap and one that of a texture map, each one must be processed in a separate but related method. That is to say when generating a 256 x 256 pixel cropped image from the heightmap image at a particular x, y coordinate, the same coordinates for creating an cropped image must be used on the texture map. This behavior is a requirement for the pix2pix GAN,

which is an image transformation GAN, requires image pairs. Typically the paired images are referred to as A and B, where image A is *translated* to image B. In this case, image A is a 1-channel (grayscale) heightmap and the other, image B is a 3-channel RGB texture map.

To assist in the creation of the data set, a data set creation utility has been constructed as a Jupyter notebook. The notebook will download the images used in the project and create 256 x 256 pixel sized cropped or chunked images. Starting at the origin, which is located in the upper left corner, the 256 x 256 crop boundary is moved (the *stride* value) by a 50 pixel width per chunk.

At each crop, which is a 256 x 256 x 1 numpy array in the case of the heightmap and 256 x 256 x 3 numpy in the case of the texture map, is stored in an HDFS5 binary data file format. After using the provided Jupyter notebook **Image Cropping Utils.ipynb**, a total of 33,341 paired images are created.

There is an additional step that is performed on the data set. The grayscale and RGB values from the cropped images are in the range of 0 to 255. The output layer from the generator of the pix3pix GAN uses *tanh* as the activation function. The range of input for the *tanh* function is from -1 to 1. Therefore, the input images must be scaled as such using the scaling equation shown as 4.1, where A is a $n \times n$ Numpy matrix with values of a single cropped image.

$$\frac{\mathrm{A} - 127.5}{127.5} \tag{4.1}$$

Equation 4.1: Scaling RGB image values 0,255 to -1,1

This scaling function was implemented as part of the train-validation split data set creation utility found in the Jupyter notebook provided. The HDFS5 file with raw image data that was created in the previous step is used as input. A function performs a randomized split of the image data into a 90% train and 10% validation

data set. During this process, the data is also scaled between values of -1 and 1. The result is a new scaled train-validation HDFS5 file. A further discussion of how the dataset is processed during the training loop is discussed under the *Helper Functions* section.

4.2 Keras Deep Learning Library

Keras is an open source framework written in Python for constructing neural networks on top of other deep learning backends. Keras was developed and maintained by Francois Chollet, an engineer at Google. Keras provides enough flexibility to code complex generative adversarial networks (GANs). Any Keras model that only leverages built-in layers will be portable across all supported backends. A keras model can be trained with one backend, and loaded it with another (e.g. for deployment). Available backends include:

- The TensorFlow backend (from Google)
- The CNTK backend (from Microsoft)
- The Theano backend

For this project Tensorflow is used. Keras contains support for processing on CPU add GPU. Distributed processing on GPUs is also available. It is released under the permissive MIT license.

4.2.1 Keras API Architecture

The main type of model in Keras is called a Sequence, which is a linear stack of layers. After instantiating a Sequence instance, additional layers can be added in the order that is required for the computation of a neural network model. After point the underlying backend framework is used to optimize the computation to be performed

by the implemented model. At the model compilation step, a loss function and an optimizer is specified.

Once the model is is compiled, the model is ready to be fit/trained on data. The fitting of the data can be done in a single batch, in several batches. Typically this fitting phase is designed to be a separate function that consists of several steps that manage the entire training process on the compiled model(s). When training is considered complete, the output is a model that can be used for predictions on new data. The trained model can be exported for export to another system or later use.

To create an example convolutional neural network (CNN), the **Sequential** object is instantiated and then convolutional, maxpooling, and dropout layers are added using the **add** function. The model flattens the output and passes it to a dense and dropout layer while finally passes it on to the output layer. The model is then compiled using the **compile** function where a loss function is specified along with optimizer which is used to minimize the loss function by updating the weights using the gradients. Useful metrics like accuracy are added as well for analyzing model training performance.

```
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPool2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Dropout

model= Sequential()
model.add(Conv2D(filters=32, kernel_size=(5, 5), activation='relu',
↪  input_shape=x_train.shape[1:]))
model.add(Conv2D(filters=32, kernel_size=(5, 5), activation='relu'))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Dropout(rate=0.25))
model.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu'))
model.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Dropout(rate=0.25))
```

```
model.add(Flatten())
model.add(Dense(256, activation='relu')
model.add(Dropout(rate=0.5))
model.add(Dense(10, activation='softmax')

model.compile(
    loss=categorical_crossentropy',
    optimizer='adam',
    metrics=['accuracy']
    )
```

4.2.2 GANs with Keras Implementation Overview

The Generative Adversarial Network, a GAN, is an architecture for training a generative model. The overall architecture is comprised of two models, the discriminator and the generator, along with a training sequence. In essence, the goal of the process is to use the discriminator model to assist in the training of the generator model. During the training process, the discriminator receives images from a set of real images, provided by the training data set, and images that are synthetic as the output from the generator. The discriminator is able to calculate loss and update weights on it's own just as any neural network would. Training the generator model however differs as it requires feedback from the discriminator in order to adjust layer weights during the training sequence. When the discriminator is not able to be fooled by the generator, and thus is able to determine the image from the generator is indeed synthetic, larger weight changes to the generator are made. Conversely, when the generator succeeds at fooling the discriminator, that is to output an input image label as real when the input image is actually synthetic, weight updates made to the generator are smaller.

To implement a GAN using Keras, one must create (at least) two models, the discriminator and the generator. Suppose one would like to create a GAN that outputs values from a one dimensional model, such as $y = f(x)$. In this case x are

the input values and y are the output values. The x^2 function is a simple function to use, which will return the square of the input. A discriminator can be defined with inputs that are two sample real value scalars and the output would be a binary classification, the likelihood the sample is real or synthetic.

If the sample input is restricted to the range between -0.5 and 0.5, the discriminator model can be constructed with an input layer of two values, a single hidden layer with 25 nodes, using a **ReLU** activation function and the **He** weight initialization method. The output layer returns the likelihood of the input as real using the **sigmoid** function. Additionally, since this is a binary classification method, the **binary crossentrophy** loss function is used along with the **Adam** optimizer. Finally accuracy is the performance metric of interest.

```
from keras.models import Sequential
from keras.layers import Dense

# define the standalone discriminator model
discriminator = Sequential()
discriminator.add(Dense(25, activation='relu',
↪  kernel_initializer='he_uniform', input_dim=2))
discriminator.add(Dense(1, activation='sigmoid'))

# compile the model
discriminator.compile(loss='binary_crossentrophy', optimizer='adam',
↪  metrics=['accuracy'])
```

After the discriminator model is compiled successfully, the model can be used in the training sequence.

For the generator, the input to the model is a sample from the latent space, which is used to generate a new sample, a vector with both the input and output of the function x and x^2 or y. The generator learns or gives meaning to the latent space during the training sequence as it receives performance feedback from the discriminator. A standard practice is to use a Gaussian distribution for each variable in the latent space where each drawing of random numbers will be $X \sim N(\mu, \sigma^2)$.

A generator model can be constructed using an input of five-element vector of Gaussian random numbers. A single hidden layer with 15 nodes, using the ReLU activation function and He weight initialization. The output layer will have two nodes, one for x and one for x^2 values respectively, along with a linear activation function. A linear activation function is used since the scale of the corresponding x value will range from -0.5 to 0.5 and the y value will range from 0.0 to 0.25. Notice the generator is not compiled. The reason being, when using Keras, this standalone generator is not fit directly, however receives weight updates from a third yet to be defined logical model in order to improve output from the standalone generator.

```
from keras.models import Sequential
from keras.layers import Dense

# define the standalone generator model
generator = Sequential()
generator.add(Dense(15, activation='relu',
↪  kernel_initializer='he_uniform', input_dim=5))
generator.add(Dense(2, activation='linear'))
```

As mentioned before, the weights in the generator model are updated based on the performance of the discriminator. When the discriminator performs well against the discriminator, the generator is updated with larger changes in the weight values through a larger error gradient. Conversely, when the discriminator is performing poorly against the generator, changes to the weights are smaller. This dynamic defines the adversarial relationship between the discriminator and generator model. To update the weights for the generator during training using Keras, a simple approach is to create a third composite model that encapsulates the standalone generator and the standalone discriminator models. Essentially this logical GAN model is one where the generator receives input from the latent space and generates samples that are sent to the discriminator for classification. The weights are updated on the generator however updates to the discriminator is this GAN model are disabled, which prevents

the standalone discriminator from being over trained on synthetic samples.

Another essential step for the training process in this composite GAN model, the labels on the synthetic images are set as real as the generator attempts to fool the discriminator. When the discriminator is performing well, the output from the discriminator will determine the incoming values from generator are not real. Then, through the backpropagation process, the weights for only the generator are updated to correct for this error. Over time, the generator improves the production of synthetic output.

```
from keras.models import Sequential
from keras.layers import Dense

# define the composite generator and discriminator model, for
↪  updating the generator

# discriminator model
gan_discriminator = Sequential()
gan_discriminator.add(Dense(25, activation='relu',
↪  kernel_initializer='he_uniform', input_dim=2))
gan_discriminator.add(Dense(1, activation='sigmoid'))
gan_discriminator.compile(loss='binary_crossentrophy',
↪  optimizer='adam', metrics=['accuracy'])
# make the discriminator not trainable when updating the generator
gan_discriminator.trainable = False

# generator model
gan_generator = Sequential()
gan_generator.add(Dense(15, activation='relu',
↪  kernel_initializer='he_uniform', input_dim=5))
gan_generator.add(Dense(2, activation='linear'))

# combine the two models and compile
gan_model = Sequential()
gan_model.add(gan_generator)
gan_model.add(gan_discriminator)
gan_model.compile(loss='binary_crossentrophy', optimizer='adam')
```

The final remaining major step in the process is to train the defined discriminator

and composite GAN together. The training process exists as a for loop, first training the discriminator, then the composite GAN. Within the training loop functions to generate real and synthetic samples for the discriminator are utilized. Likewise for the composite GAN, functions to sample vectors from the latent space and label inversion are included. In the training loop example below, the models are trained for 10000 epochs with a batch size of 128. The batch size controls the number of training samples to work through before the model's weight parameters are updated. Weight parameter updates occur once every epoch through backpropagation. This is handled by the **train_on_batch** function that is part of the **Sequential()** class. For brevity, code examples for generating data, real and synthetic, generating points in the latent space, and label inversion are left out. These processes are handled outside of the Keras framework. The entire example code can be view in Appendix B: Example GAN Using Keras.

```
# train the generator and discriminator
import numpy

num_epochs=10000
num_batch=128

# calculate half the size of one batch, for training the
↪  discriminator
half_batch = int(num_batch / 2)

# loop through the number of epochs
for i in range(num_epochs):

    # retrieve real samples
    x_real, y_real = generate_real_samples(half_batch)

    # retrieve synthetic samples
    x_fake, y_fake = generate_fake_samples(generator, half_batch)

    # train discriminator for one epoch
    discriminator.train_on_batch(x_real, y_real)
    discriminator.train_on_batch(x_fake, y_fake)
```

```
# retrieve points in the latent space as input for the generator
x_gan = generate_latent_points(latent_dim, num_batch)

# create inverted labels for the fake samples
y_gan = numpy.one((n_batch, 1))

# train the generator using the discriminator's error
gan_model.train_on_batch(x_gan, y_gan)
```

4.3 Creating the DCGAN using Keras

As discussed in the Design chapter, heightmaps are generated using a trained DCGAN. Creating the DCGAN using Keras has three main components: the discriminator model, the generator model, and the composite GAN model.

4.3.1 The Discriminator Model

The discriminator model takes a sample image as input and outputs a classification predicting whether the sample is *real* or *fake*. The input layer takes a $256 \times 256 \times 1$ size image as input. This first layer is a **GaussianNoise** layer which acts a regularization layer to prevent the discriminator from learning too quickly. The Gaussian noise takes an input value for the standard deviation of noise, a value of 0.2 is chosen arbitrarily. Each hidden layer is a convolutional **Conv2D** layer with a **LeakyReLU** activation function, using **Dropout** and **AveragePooling2D**. The output layer is a **sigmoid** activtion function that outputs a 1 for *real* and 0 for *fake*. The model is compiled using the **binary crossentropy** loss function and the **Adam** version of stochastic gradient descent with a learning rate of 0.0002 and a momentum of 0.5.

```
# example DCGAN discriminator
from keras.models import Sequential
from keras.optimizers import Adam
```

```
from keras.layers import GaussianNoise
from keras.layers import BatchNormalization
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import LeakyReLU
from keras.layers import Dropout
from keras.layers import AveragePooling2D

# define the standlone discrimintor model
def define_discriminator(in_shape=(256, 256, 1)):
    model = Sequential()

    # add Gaussian noise to prevent Discriminator overfitting
    model.add(GaussianNoise(0.2, input_shape=in_shape))

    # 256x256x1 Image
    model.add(Conv2D(filters=8, kernel_size=3, padding='same'))
    model.add(LeakyReLU(0.2))
    model.add(Dropout(0.25))
    model.add(AveragePooling2D())

    # 128x128x8
    model.add(Conv2D(filters=16, kernel_size=3, padding='same'))
    model.add(BatchNormalization(momentum=0.7))
    model.add(LeakyReLU(0.2))
    model.add(Dropout(0.25))
    model.add(AveragePooling2D())

    # 64x64x16
    model.add(Conv2D(filters=32, kernel_size=3, padding='same'))
    model.add(BatchNormalization(momentum=0.7))
    model.add(LeakyReLU(0.2))
    model.add(Dropout(0.25))
    model.add(AveragePooling2D())

    # 32x32x32
    model.add(Conv2D(filters=64, kernel_size=3, padding='same'))
    model.add(BatchNormalization(momentum=0.7))
    model.add(LeakyReLU(0.2))
    model.add(Dropout(0.25))
    model.add(AveragePooling2D())

    # 16x16x64
    model.add(Conv2D(filters=128, kernel_size=3, padding='same'))
```

```
model.add(BatchNormalization(momentum=0.7))
model.add(LeakyReLU(0.2))
model.add(Dropout(0.25))
model.add(AveragePooling2D())

# 8x8x128
model.add(Conv2D(filters=256, kernel_size=3, padding='same'))
model.add(BatchNormalization(momentum=0.7))
model.add(LeakyReLU(0.2))
model.add(Dropout(0.25))
model.add(AveragePooling2D())

# 4x4x256
model.add(Flatten())

# 256
model.add(Dense(128))
model.add(LeakyReLU(0.2))

model.add(Dense(1, activation='sigmoid'))

# compile the model
opt = Adam(lr=0.0002, beta_1=0.5)
model.compile(loss='binary_crossentropy', optimizer=opt,
↪  metrics=['accuracy'])

return model
```

4.3.2 The Generator Model

The generator role in the GAN is to generate new synthetic (fake) images. By taking input form the latent space and generating an image as output. The first layer is a combined **UpSampling2D** layer and a **Conv2D** layer into a **Conv2Dtransponse** layer. This breaks the input latent vector and reshapes the input into 256 different low resolution 4×4 feature maps with a **ReLU** activation function using the default slope of 0.2. The next two layers provide upsampling by using a convolutional **Conv2D** layer with a **'filter'** value of 256 and a **'kernel'** size of 4. Larger kernel sizes have shown to capture more more information and introduce smoothness within the feature

maps (Desai, 2018). The padding value of **'same'** creates a single feature map and maintains the $256 \times 256 \times 1$ input dimensions. The remaining hidden layers continue upsample however the **'kernel'** is changed to 3. The output layer in a **Conv2D** layer with an **sigmoid** activation layer which will output a single $256 \times 256 \times 1$ image.

```
# example DCGAN generator
from keras.models import Sequential
from keras.optimizers import Adam
from keras.layers import BatchNormalization
from keras.layers import Conv2DTranspose
from keras.layers import Conv2D
from keras.layers import Activation
from keras.layers import UpSampling2D

# define the standalone generator model
def define_generator(latent_dim):
    model = Sequential()

    model.add(Reshape(target_shape=[1, 1, latent_dim],
    ↪  input_shape=[latent_dim]))

    # 1x1x250
    model.add(Conv2DTranspose(filters=256, kernel_size=4))
    model.add(Activation('relu'))

    # 4x4x256
    model.add(Conv2D(filters=256, kernel_size=4, padding='same'))
    model.add(BatchNormalization(momentum=0.7))
    model.add(Activation('relu'))
    model.add(UpSampling2D())

    # 8x8x256
    model.add(Conv2D(filters=128, kernel_size=4, padding='same'))
    model.add(BatchNormalization(momentum=0.7))
    model.add(Activation('relu'))
    model.add(UpSampling2D())

    # 16x16x128
    model.add(Conv2D(filters=64, kernel_size=3, padding='same'))
    model.add(BatchNormalization(momentum=0.7))
    model.add(Activation('relu'))
    model.add(UpSampling2D())
```

```
    # 32x32x64
    model.add(Conv2D(filters=32, kernel_size=3, padding='same'))
    model.add(BatchNormalization(momentum=0.7))
    model.add(Activation('relu'))
    model.add(UpSampling2D())

    # 64x64x32
    model.add(Conv2D(filters=16, kernel_size=3, padding='same'))
    model.add(BatchNormalization(momentum=0.7))
    model.add(Activation('relu'))
    model.add(UpSampling2D())

    # 128x128x16
    model.add(Conv2D(filters=8, kernel_size=3, padding='same'))
    model.add(Activation('relu'))
    model.add(UpSampling2D())

    # 256x256x8
    model.add(Conv2D(filters=1, kernel_size=3, padding='same'))
    model.add(Activation('sigmoid'))

    return model
```

4.3.3 GAN Composite Model

The GAN composite model is used to train the generator model. During the training sequence the weights of the generator are updated based on the performance of the discriminators accuracy on detecting *real* and *fake* images. When the discriminator is performing well at detecting fake images as created from the generator, the weights occur more frequently and are larger in the size of the change. Conversely, as the detection of fake images by the discriminator get better overtime, leading to poorer performance of the discriminator, the weights for the generator are updated less frequently and are smaller in size. This is the core of the adversarial relationship between the discriminator and the generator.

As such, Keras provides a simple approach to facilitating generator weight updates by creating a third logical model. This third model allows the generator to test

the discriminator performance with *fake* images labeled as *(*real) and receive weight updates. During this process, true *real* images are not show to the discriminator and weight updates are frozen by setting the discriminator's **trainable** setting to **'False'**.

```
# example GAN composite model
from keras.models import Sequential
from keras.optimizers import Adam

# define the combined generator and discriminator model, for
↪  updating the generator
def define_gan(g_model, d_model):
    # make weights in the discriminator not trainable
    d_model.trainable = False

    # connect them
    model = Sequential()

    # add generator
    model.add(g_model)

    # add the discriminator
    model.add(d_model)

    # compile model
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt)

    return model
```

4.4 Creating pix2pix GAN using Keras

pix2pix GAN is an image translation GAN or cGAN, where the output image is conditioned on the input image. The discriminator is a PatchGAN and the generator is U-Net encoder-decoder generator that utilizes skip layers. Once again the addition of the third composite GAN will be to used for performing the generator weight updates during training steps.

4.4.1 The Discriminator Model

The PatchGAN, like most discriminators, works to minimize the success of the generator and produces binary outcomes of *real* or *fake* as we have seen before. The PatchGAN accomplishes this uses a receptive field to acheive this. As an example, a PatchGAN with a size 70×70 will classify 70×70 patches of the image as *real* or *fake*. The authors of the PatchGAN paper determined a size of 70×70 provided the best trade-off between image quality and performance.

The discriminator takes an image pair, image A and image B, as input. The two images are merged together at the channel level, meaning two 3-channel images become 6 channels of input. The layers of the PatchGAN consists of a block of layers: a convolution layer, a batch normalization layer, and a LeakyReLU activation layer. The block is usually referred to as C and used in the notation C64-C128-C256-C512. a **kernel** size of 4×4 is used and a **stride** of 2×2 is used in each block except for the last two. **LeakyReLU** is used as the activation layer, a **sigmoid** function is used for the last layer which output the binary result 1 (*real*) or 0 (*fake*).

```
# example pix2pix discriminator

# define the discriminator model
def define_discriminator(image_shape):
    # weight initialization
    init = RandomNormal(stddev=0.02)
    # source image input
    in_src_image = Input(shape=image_shape)
    # target image input
    in_target_image = Input(shape=image_shape)
    # concatenate images channel-wise
    merged = Concatenate()([in_src_image, in_target_image])
    # C64
    d = Conv2D(64, (4, 4), strides=(2, 2), padding='same',
    ↪  kernel_initializer=init)(merged)
    d = LeakyReLU(alpha=0.2)(d)
    # C128
```

```
d = Conv2D(128, (4, 4), strides=(2, 2), padding='same',
↪  kernel_initializer=init)(d)
d = BatchNormalization()(d)
d = LeakyReLU(alpha=0.2)(d)
# C256
d = Conv2D(256, (4, 4), strides=(2, 2), padding='same',
↪  kernel_initializer=init)(d)
d = BatchNormalization()(d)
d = LeakyReLU(alpha=0.2)(d)
# C512
d = Conv2D(512, (4, 4), strides=(2, 2), padding='same',
↪  kernel_initializer=init)(d)
d = BatchNormalization()(d)
d = LeakyReLU(alpha=0.2)(d)
# second last output layer
d = Conv2D(512, (4, 4), padding='same',
↪  kernel_initializer=init)(d)
d = BatchNormalization()(d)
d = LeakyReLU(alpha=0.2)(d)
# patch output
d = Conv2D(1, (4, 4), padding='same', kernel_initializer=init)(d)
patch_out = Activation('sigmoid')(d)
# define model
model = Model([in_src_image, in_target_image], patch_out)
# compile model
opt = Adam(lr=0.0002, beta_1=0.5)
model.compile(loss='binary_crossentropy', optimizer=opt,
↪  loss_weights=[0.5])
return model
```

4.4.2 The Generator Model

The genertor model for pix2pix is a U-Net model. The U-Net is a encoder-decoder model used for image translation. Skip connections are used to connect encoder layers with the corresponding decoder layer. Just as the PatchGAN, blocks are used form the encoder and decoder layers. In fact the same block/layer configuration that is used by the PatchGAN, a convolution layer, a batch normalization layer, and a LeakyReLU activation layer, is used by the encoder. However the decoder adds a **Dropout** layer to form a convolution layer, a batch normalization layer, Dropout

layer, and a ReLU activation layer as a block. Using the nation established in the preceeding section we can describe the U-Net model as:

- **Encoder** C64-C128-C256-C512-C512-C512-C512-C512
- **Decoder** C512-CD1024-CD1024-C1024-C1024-C512-C256-C128

The activation layer for the output layer is a **tanh** which produces image result in the range of -1 to 1.

```
# define an encoder block
def define_encoder_block(layer_in, n_filters, batchnorm=True):
    # weight initialization
    init = RandomNormal(stddev=0.02)
    # add downsampling layer
    g = Conv2D(n_filters, (4, 4), strides=(2, 2), padding='same',
    ↪  kernel_initializer=init)(layer_in)
    # conditionally add batch normalization
    if batchnorm:
        g = BatchNormalization()(g, training=True)
    # leaky relu activation
    g = LeakyReLU(alpha=0.2)(g)
    return g


# define a decoder block
def decoder_block(layer_in, skip_in, n_filters, dropout=True):
    # weight initialization
    init = RandomNormal(stddev=0.02)
    # add upsampling layer
    g = Conv2DTranspose(n_filters, (4, 4), strides=(2, 2),
    ↪  padding='same', kernel_initializer=init)(layer_in)
    # add batch normalization
    g = BatchNormalization()(g, training=True)
    # conditionally add dropout
    if dropout:
        g = Dropout(0.5)(g, training=True)
    # merge with skip connection
    g = Concatenate()([g, skip_in])
    # relu activation
    g = Activation('relu')(g)
```

```
    return g


# define the standalone generator model
def define_generator(image_shape=(256, 256, 1), output_channels=3):
    # weight initialization
    init = RandomNormal(stddev=0.02)
    # image input
    in_image = Input(shape=image_shape)
    # encoder model
    e1 = define_encoder_block(in_image, 64, batchnorm=False)
    e2 = define_encoder_block(e1, 128)
    e3 = define_encoder_block(e2, 256)
    e4 = define_encoder_block(e3, 512)
    e5 = define_encoder_block(e4, 512)
    e6 = define_encoder_block(e5, 512)
    e7 = define_encoder_block(e6, 512)

    # bottleneck, no batch norm and relu
    b = Conv2D(512, (4, 4), strides=(2, 2), padding='same',
    ↪  kernel_initializer=init)(e7)
    b = Activation('relu')(b)

    # decoder model
    d1 = decoder_block(b, e7, 512)
    d2 = decoder_block(d1, e6, 512)
    d3 = decoder_block(d2, e5, 512)
    d4 = decoder_block(d3, e4, 512, dropout=False)
    d5 = decoder_block(d4, e3, 256, dropout=False)
    d6 = decoder_block(d5, e2, 128, dropout=False)
    d7 = decoder_block(d6, e1, 64, dropout=False)

    # Output
    # After the last layer in the decoder, a Transposed convolution
    # layer (sometimes called Deconvolution) is applied
    # to map to the number of output channels (3 in general,
    # except in colorization, where it is 2), followed by a Tanh
    # function.
    g = Conv2DTranspose(output_channels, (4, 4), strides=(2, 2),
    ↪  padding='same', kernel_initializer=init)(d7)
    out_image = Activation('tanh')(g)
    # define model
    model = Model(in_image, out_image)
    return model
```

4.4.3 GAN Composite Model

Just as was done for the DCGAN, pix2pix in Keras can take advantage of the composite model to update the generator during training steps. With the pix2pix, the weights of the generator will be updated by the total loss calculate as such:

$$\text{loss} = \text{adversarial loss} + \lambda \times \text{L1 loss} \tag{4.2}$$

Where 1 loss is used for regularization and is weight by hyperparameter λ, set to 100. This different loss function is used since generated image is compared to the actual target image for exacting similarity.

```
# define the combined generator and discriminator model, for
↪  updating the generator
def define_gan(g_model, d_model, image_shape):
    # make weights in the discriminator not trainable
    d_model.trainable = False
    # define the source image
    in_src = Input(shape=image_shape)
    # connect the source image to the generator input
    gen_out = g_model(in_src)

    # connect the source input and generator output to the
    ↪  discriminator input
    dis_out = d_model([in_src, gen_out])

    # src image as input, generated image and classification output
    model = Model(in_src, [dis_out, gen_out])

    # compile model
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss=['binary_crossentropy', 'mae'], optimizer=opt,
    ↪  loss_weights=[1, 100])
    return model
```

4.5 Common helper functions

Creating the generator and discriminator for each GAN is only part of the issue. There are requirements for reading in data, providing randomized real an fake data, tracking the training progress. For this helper functions are used.

4.5.1 Reading in Data

The data is read in as a scaled HDFS5 file which contains heightmaps, labeled as x, and texture maps, labeled as y, split between a train, labeled as t and validation, labeled as v. For training the DCGAN, only the training heightmap is required, so the label of xt is used. The following function shows how data is ingested and then return to the training loop as an $(datasetsize, n, n, 1 - channel)$ Numpy array.

```
# example data ingest for the DCGAN
import h5py

# load and prepare training images
def load_real_samples(filename):
    # load image data, Read HDF5 file
    f = h5py.File(filename, 'r')
    dataset = f['xt']
    print('Dataset input shape {}'.format(str(dataset.shape)))

    return dataset
```

The pix2pix GAN uses pairs of images, so both xt and yt image set are used.

```
# example data ingest for pix2pix
import h5py

# load and prepare training images
def load_real_samples(filename):

    # load image data, Read HDF5 file
    f = h5py.File(filename, 'r')
```

```
    # Grab the first 216 random images
    X1 = f['xt']
    X2 = f['yt']
    print('Dataset input shapes {}, {}'.format(str(X1.shape),
    ↪  str(X2.shape)))

    return [X1, X2]
```

4.5.2 Generate Real Training Samples

During the training process, random real images are sent to the discriminator each training batch with a class label of real or 1. The function creates index values from the supplied number of available images. Then, a random n number of those samples are chosen, without replacement, to be used as the training batch.

```
# example of choosing real samples for the DCGAN
from numpy.random import choice
from numpy import ones

# select real samples
def generate_real_samples(dataset, n_samples):
    # create index numbers
    ix = range(dataset.shape[0])

    # choose n_samples of integers at random without replacement
    # sort the result
    ix = sorted(choice(ix, size=n_samples, replace=False))

    # save index values for use with other GANs
    save_idx(ix)

    # retrieve selected images using fancy indexing
    X = dataset[ix]

    # generate 'real' class labels (1)
    y = ones((n_samples, 1))

    return X, y
```

And for the pix2pix GAN, selecting image pairs.

```
# example of choosing real samples for pix2pix
from numpy.random import choice
from numpy import ones

# select a batch of random samples, returns images and target
def generate_real_samples(datasetA, datasetB, n_samples, patch_shape):

    # choose random instances
    ix = range(datasetA.shape[0])

    # choose n_samples of integers at random without replacement
    # sort the result
    ix = sorted(choice(ix, size=n_samples, replace=False))

    # ix = randint(0, datasetA.shape[0], n_samples)

    # retrieve selected images
    X1, X2 = datasetA[ix], datasetB[ix]

    # convert grayscale 1-channel to RGB 3-channel
    X1 = gray2rgb(X1)

    # generate 'real' class labels (1)
    y = ones((n_samples, patch_shape, patch_shape, 1))
    return [X1, X2], y
```

4.5.3 Generate Fake Training Samples

During the training process, the generator attempts to fool the discriminator by providing images that are fake, but are labeled as real. To do this, a batch of synthetic images are created from the generator using the generator's predict function. For the DCGAN, an input vector from the latent space is required, so there is an additional function to generate random latent space vectors from a $\sim N(0, 1)$ Gaussian distribution.

```
# example of generating fake samples using the DCGAN
from numpy.random import normal
from numpy import zeros

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
    # generate points in the latent space
    x_input = normal(0, 1, size=(n_samples, latent_dim))

    # reshape into a batch of inputs for the network
    x_input = x_input.reshape(n_samples, latent_dim)

    return x_input


# use the generator to generate n fake examples, with class labels
def generate_fake_samples(g_model, latent_dim, n_samples):
    # generate points in latent space
    x_input = generate_latent_points(latent_dim, n_samples)

    # predict outputs
    X = g_model.predict(x_input)

    # create 'fake' class labels (1)
    y = zeros((n_samples, 1))

    return X, y
```

For pix2pix, a real heightmap image is used as input to the generator and the generator creates fake texture map. The image pair is sent to the discriminator for evaluation.

```
# example of generating fake samples using the DCGAN
from numpy import zeros

# generate a batch of images, returns images and targets
def generate_fake_samples(g_model, samples, patch_shape):
    # generate fake instance
    X = g_model.predict(samples)
    # create 'fake' class labels (0)
    y = zeros((len(X), patch_shape, patch_shape, 1))
    return X, y
```

4.6 The Training Loop

The training loop controls the retrieval of a batch of real images and generation of a batch of fake images to be assessed by the discriminator. One completion of the training loop is referred as an *epoch*. The training loop also tracks the training loss for the discriminator and the generator, and periodically saves a trained generator model to disk.

```
# example training loop

# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim,
↪  n_epochs=100, n_batch=256):

    # prepare lists for storing stats each iteration
    d1_hist, d2_hist, g_hist, a1_hist, a2_hist = list(), list(),
    ↪  list(), list(), list()

    bat_per_epo = int(dataset.shape[0] / n_batch)
    half_batch = int(n_batch / 2)

    # manually enumerate epochs
    for i in range(n_epochs):

        # enumerate batches over the training set
        for j in range(bat_per_epo):

            # get randomly selected 'real' samples
            X_real, y_real = generate_real_samples(dataset,
            ↪  half_batch)

            # update discriminator model weights
            d_loss1, d_acc1 = d_model.train_on_batch(X_real, y_real)

            # generate 'fake' examples
            X_fake, y_fake = generate_fake_samples(g_model,
            ↪  latent_dim, half_batch)

            # update discriminator model weights
            d_loss2, d_acc2 = d_model.train_on_batch(X_fake, y_fake)
```

```
            # prepare points in latent space as input for the
            ↪  generator
            X_gan = generate_latent_points(latent_dim, n_batch)

            # create inverted labels for the fake samples
            y_gan = ones((n_batch, 1))

            # update the generator via the discriminator's error
            g_loss = gan_model.train_on_batch(X_gan, y_gan)

            # summarize loss on this batch
            #print('>%d, %d/%d, d=%.3f, g=%.3f' % (i + 1, j + 1,
            ↪  bat_per_epo, d_loss, g_loss))

            # summarize loss on this batch
            print('>%d, %d/%d, d1[%.3f], d2[%.3f], g[%.3f], a1[%d],
            ↪  a2[%d]' %
                  (i + 1, j+1, bat_per_epo, d_loss1,  d_loss2, g_loss,
                  ↪  int(100 * d_acc1), int(100 * d_acc2)))

            # record history
            d1_hist.append(d_loss1)
            d2_hist.append(d_loss2)
            g_hist.append(g_loss)
            a1_hist.append(d_acc1)
            a2_hist.append(d_acc2)

        # evaluate the model performance, every 10 epochs
        if (i + 1) % 1 == 0:
            summarize_performance(i, g_model, d_model, dataset,
            ↪  latent_dim)

    # create history plot at the conclusion of training
    plot_history(d1_hist, d2_hist, g_hist, a1_hist, a2_hist)
```

4.7 Project Folder Organization

The project is organized using the following structure for ease of use:

Description of folder contents:

- project: the root of the project, contains all project files and subdirectories.

```
project
|
+-- images/
+-- latex/
+-- misc/
+-- models/
+-- notebooks/
    +-- Create new maps from generator.ipynb
    +-- Exploring the Latent Space.ipynb
    +-- Image Cropping Utils.ipynb
    +-- Image Similarity.ipynb
    +-- Turning Test Image Generator.ipynb
+-- src/
    +-- dcgan.py
    +-- main.py
    +-- pix2pix.py
+-- README.md
+-- Toward Improving Procedural Terrain Generation with GANs.pdf
```

Table 4.1: Project folder structure.

- images: example heightmaps and texture maps generated by GANs created in this project.
- latex: the raw latex code used to generate the thesis PDF.
- misc: contains files to make Jupyter notebook run correctly.
- models: contains trained models that can be used with the Jupyter notebook examples and utilities.
- notebooks: a set of Jupyter notebooks that contain examples and utilities for creating the project.
- src: the Python source code used in this project.

4.8 Files and User Command Line Interface

To train either the DCGAN, the pix2pix GAN, or both, **main.py** contains the driver code. A user can initialize a new driver instance by using the commands available through the command line interface (CLI) as detailed below. **main.py** imports two files: **dcgan.py** and **pix2pix.bf** which contain the architecture and helper functions for each GAN.

To start training and initialize the driver for training, the end user uses the command line. Arguments such as the path to the image training data, training model output, and model hyperparameters. The complete list of arguments can be accessed by typing **python main.py –help** with the resulting output:

```
usage: main.py [-h] --mode {train,other} --images IMAGES --imagefile IMAGEFILE
               --output OUTPUT [--dcgan] [--dcgan_n_iter DCGAN_N_ITER]
               [--dcgan_batch_size DCGAN_BATCH_SIZE]
               [--dcgan_n_epochs DCGAN_N_EPOCHS]
               [--dcgan_latent_dim DCGAN_LATENT_DIM] [--dcgan_incl_noise]
               [--dcgan_noise_val DCGAN_NOISE_VAL] [--pix2pix]
               [--pix2pix_batch_size PIX2PIX_BATCH_SIZE]
               [--pix2pix_n_epochs PIX2PIX_N_EPOCHS]


Stacked GAN for terrain generation.


optional arguments:
  -h, --help            show this help message and exit
  --mode {train,other}  Operational mode
  --images IMAGES       Path to images
  --imagefile IMAGEFILE
```

```
                        Name of .H5 image file
  --output OUTPUT       Path to output folder (must exist)
  --dcgan               inlcude dcgan
  --dcgan_n_iter DCGAN_N_ITER
                        dcgan number of iterations
  --dcgan_batch_size DCGAN_BATCH_SIZE
                        dcgan mini-batch size
  --dcgan_n_epochs DCGAN_N_EPOCHS
                        dcgan number of epochs
  --dcgan_latent_dim DCGAN_LATENT_DIM
                        DCGAN size of latent dim
  --dcgan_incl_noise    DCGAN include noise input layer
  --dcgan_noise_val DCGAN_NOISE_VAL
                        DCGAN the amount of noise used if noise layer is true
  --pix2pix             Include pix2pix GAN
  --pix2pix_batch_size PIX2PIX_BATCH_SIZE
                        pix2pix mini-batch size
  --pix2pix_n_epochs PIX2PIX_N_EPOCHS
                        pix2pix number of epochs
```

Some example training configurations with the assumption the image HDF5 data has been created and named **data_256x256_train_val_float_scaled_neg1_to_1.hd5**:

```
# Train the DCGAN for 1 epoch with a batch size of 10
# and latent vector size of 100
python main.py --mode train --images /home/pathtoimagefolder/
--imagefile data_256x256_train_val_float_scaled_neg1_to_1.hd5
--output /home/outputfolderpath --dcgan --dcgan_n_epochs 1
--dcgan_batch_size 10 --dcgan_latent_dim 100
```

```
# Train the pix2pix GAN for 1 epoch with a batch size of 20
python main.py --mode train --images /home/pathtoimagefolder/
--imagefile data_256x256_train_val_float_scaled_neg1_to_1.hd5
--output /home/outputfolderpath --pix2pix  --pix2pix_n_epochs 1
--pix2pix_batch_size 20

# Train both DCGAN and pix2pix GAN
python main.py --mode train --images /home/pathtoimagefolder/
--imagefile data_256x256_train_val_float_scaled_neg1_to_1.hd5
--output /home/outputfolderpath --dcgan --pix2pix
--dcgan_n_epochs 1 --dcgan_batch_size 20 --dcgan_latent_dim 100
--pix2pix_n_epochs 1 --pix2pix_batch_size 20
```

There are additional files included as Jupyter notebooks. These notebooks assist with some necessary pipeline functions for generating synthetic terrain. Jupyter notebooks are used since they offer interactive output and quick code changes due to the *coding cell* interfaces provided.

- Create new maps from generator.ipynb: Synthetic heightmaps and texture maps can be created using trained models.
- Exploring the Latent Space.ipynb: The latent space of a trained DCGAN model can be explored visually and with vector arithmetic. Using vector arithmetic new heightmaps can be created with desired terrain characteristics. Chapter 6 covers this method in detail.
- Image Cropping Utils.ipynb: This notebook is used to download the NASA images used in the project. This notebook also creates HDF5 data file used for training the GANs and provides some image exploration capabilities.

- Image Similarity.ipynb: The Mean Squared Error (MSE) and Structural Similarity metric (Wang et al., 2004) are included for comparing pairs of images for similarity. This is used for the checking the quality of images generated by the pix2pix GAN.

- Turning Test Image Generator.ipynb: This notebook generates random images that can be used as part of a empirical image quality test.

A complete code listing for all of the code files used in this project can be found in Appendix B: Application Code.

Chapter 5

Development

This chapter discusses the tools and methodologies employed in the code development of the system.

5.1 Development Tools

This project uses a layered approach toward development and operations. At the lowest level, is the Python programming language, followed by Anaconda, GPU utilities/enablers, Tensorflow, and then finally Keras. The remaining layers consist of development tools such as an Integrated Development Environment (IDE), data science and image exploration notebook, and version control. The IDE used was JetBrains PyCharm, the notebook was Jupyter, and the Github for version control. To show the application of appliying the texture map using a heightmap as one would for video game development, Amazon Lumberyard was used.

5.1.1 PyCharm

PyCharm was used for the development and control of model training. Interactivity by a user for the model training is minimal during this step. Each training run can

be thought of as an experiment where initial data and parameter configuration is provided, then the training portion runs, sometimes for over day. At the conclusion of a training run, model performance results that were generated by the process can be reviewed. At the end of a training run, a training generator model is available for making predictions. That is to say a trained generator, given some input from the now learned latent space, can produce images.

5.1.2 Jupyter Notebooks

The images that are created by the generator need to be reviewed for quality and comparison to previous training runs. This part of the process can require higher interactivity from the user and a coding interface that is dynamic without necessitating the need for heavy Graphical User Interface (GUI) support during the process. For this, the open source project Jupyter was used.

5.1.3 Github

Source control for the project was provided by Github, an online host for Git, a version-control system for tracking changes in source code during software development. PyCharm integrates with Github repositories which makes the process of committing and reviewing code changes more efficient. One can manage code changes through the command line interface (CLI) for times when development takes place outside the IDE.

5.1.4 Amazon Lumberyard

Amazon Lumberyard is a free cross-platform game engine developed by Amazon and based on CryEngine, which was licensed from Crytek in 2015. The engine features integration with Amazon Web Services to allow developers to build or host their

games on Amazon's servers, as well as support for livestreaming via Twitch.

5.2 Development Methodologies

The project is architected using Object Oriented (OO) design. A driver is then used to orchestrate the instantiation of the discriminator object, generator object, composite GAN object, and control of the training loop. The use of the object oriented programming paradigm allows for modularity which allows for the instantiation of both the discriminator and generator. The discriminator and generator objects are then reused for the buidling of the composite GAN model. When using the Keras framework, recall the training of the generator is done using the composite GAN model. Weight updates to the discriminator are frozen which prevents overfitting on synthetic images. The object oriented properties allow this training technique to be implemented in a simple manner.

In this project, to create the discriminator and generator, a Keras Sequential model object can be instantiated within separate function calls. The layers of the model are added sequentially within the function. The function returns the completed model object.

Chapter 6

GAN applications

6.1 Principles of Generative Adversarial Networks

GANS are able to learn the input distribution to competing (and cooperating) networks referred to as a generator and a discriminator. The role of the generator in to generate signal, that is essentially fake data, that can fool the discriminator. In turn, the discriminator is trained to determine the different between fake and real signals. As the training progresses, the discriminator will eventually no longer be able to be able to distinguish between synthetic and real data. At this point, the discriminator can be discarded, and the generator can be used to generate synthetic data that is comparable to real data, such as images or music.

Issues with training GANs do exists and can prevent the networks from learning simultaneously. During the training process, the loss function is handled by the discriminator. Accordingly, when the discriminator converges faster than generator, the generator no longer receives sufficient gradient updates for its parameters and fails to converge. GANs can also suffer from either a partial or total modal collapse, a situation wherein the generator is producing almost similar outputs for different latent encodings (Goodfellow, 2016).

As shown in Figure 6.1, a GAN is made up of two networks, a generator and a discriminator. The input to the generator is noise, while the output is synthesized signal. As for the discriminator, the discriminator's input will be synthesized or real. Genuine signals come from the true sampled data, while the fake signals come from the generator. All of the valid signals are labeled 1.0, meaning the probability of the signal being real is 100%. However, synthesized signals are labeled as 0.0, a 0% probability of being real.

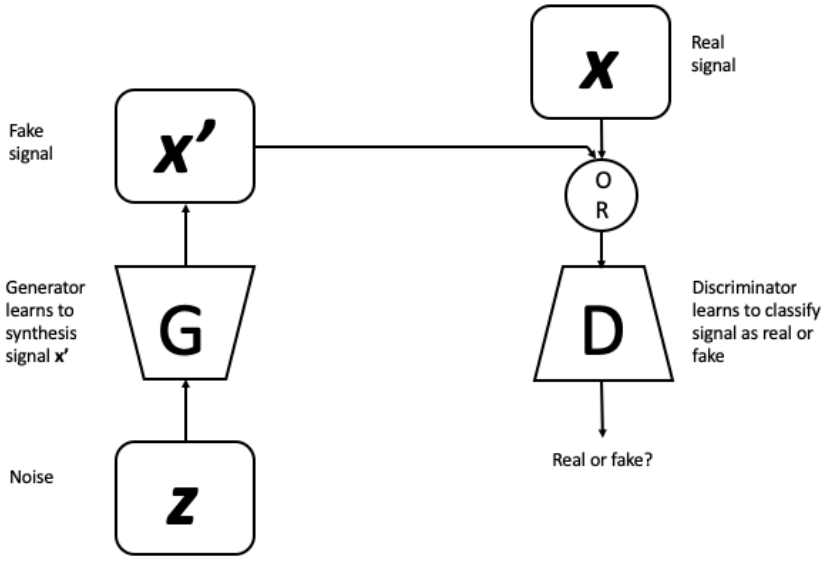

Figure 6.1: A GAN is made up of two networks, a generator, and a discriminator. The discriminator is trained to distinguish between real and fake signals or data. The generator's role is to generate fake signals or data that can eventually fool the discriminator.

The objective of the discriminator is to learn from this supplied data set on how to distinguish real signals from fake signals. During this part of GAN training, only the discriminator parameters will be updated. Like a typical binary classifier, the discriminator is trained to predict on a range of 0.0 to 1.0 in confidence values on how close a given input signal is to the true one. However, this is only one side of the process.

At regular intervals, the generator will pretend that its output is a genuine signal

and will ask the GAN to label it as 1.0 (real data). When the fake signal is then presented to the discriminator, naturally it will be classified as fake with a label close to 0.0. The optimizer computer computes the generator parameter updates based on the presented label (1.0 in this case). The discriminator also takes its own prediction into account when training on this new data sent by the generator. In other words, the discriminator has some doubt about its own prediction, and so, GANs takes that into consideration. This time, GANs will let the gradients backpropagate from the last layer of the discriminator down to the first layer of the generator. However, in most practices, during this phase of training, the discriminator parameters are temporary frozen. The generator will use gradients to update its parameters and improve its ability to synthesize fake signals.

Overall, the whole process is akin to two networks competing with one another while still cooperating at the same time. When the GAN training converges, the end result is a generator that can synthesize signals. The discriminator thinks these synthesized signals are real or with a label near 1.0, which means the discriminator can then be discarded. The generator part will be useful in producing meaningful outputs from arbitrary noise inputs.

As shown in Figure 6.2, the discriminator can be trained by minimizing the loss function in the following equation:

$$\mathcal{L}^{(D)}(\Theta^{(G)}, \Theta^{(D)}) = -\mathbb{E}_x \sim P_{data} \log D(x) - \mathbb{E}_z \log(1 - D(G(z))) \tag{6.1}$$

Equation 6.1: The Discriminator loss function.

The equation is just the standard binary cross-entropy cost function. The loss is the negative sum of the expectation of correctly identifying real data, $D(x)$, and the expectation of 1.0 minus correctly identifying synthetic data, $1 - D(G(z))$. The log does not change the location of the local minima. Two mini-batches of data are supplied to the discriminator during training:

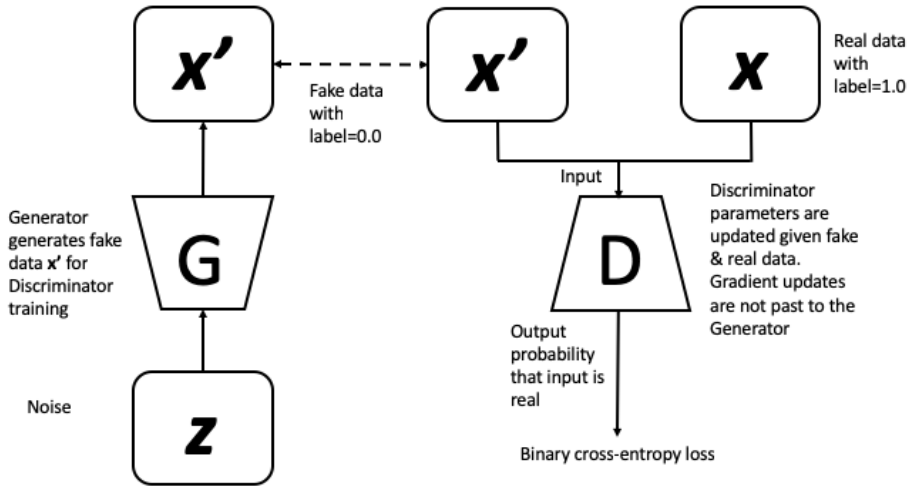

Figure 6.2: Training the discriminator is similar to training a binary classifier network using binary cross-entropy loss. The fake data is supplied by the generator while real data is from true samples.

1. x, real sampled data (that is $x \sim P_{data}$) with label 1.0

2. $x' = G(z)$, fake data from the generator that is labeled 0.0

In order to minimize the loss function, the discriminator parameters, $\Theta^{(D)}$, will be updated through backpropagation by correctly identifying the genuine data, $D(x)$, and synthetic data, $1 - D(G(z))$. Correctly identifying real data is equivalent to $D(x) \rightarrow 1.0$ while correctly classifying fake data is the same as $D(G(Z)) \rightarrow 0.0$ or $(1-D(G(z))) \rightarrow 1.0$. In this equation, z is the arbitrary encoding or noise vector that is used by the generator to synthesize new signals. Both contribute to minimizing the loss function. To train the generator, GAN considers the total of the discriminator and generator losses as a zero-sum game. The generator loss function is simply the negative of the discriminator loss function:

$$\mathcal{L}^{(G)}(\Theta^{(G)}, \Theta^{(D)}) = -\mathcal{L}^{(D)}(\Theta^{(G)}, \Theta^{(D)}) \tag{6.2}$$

Equation 6.2: The Generator loss function.

This can then be rewritten more eloquently as a value function:

$$V^{(G)}(\Theta^{(G)}, \Theta^{(D)} = -\mathcal{L}^{(D)}(\Theta^{(G)}, \Theta^{(D)}) \tag{6.3}$$

Equation 6.3: The Generator loss function as a value function.

From the perspective of the generator, the preceding equation should be minimized. From the point of view of the discriminator, the value function should be maximized. Therefore, the generator training criterion can be written as a minimax problem:

$$\Theta^{(G)*} = arg \min_{\Theta^{(G)}} \max_{\Theta^{(D)}} V^{(D)}(\Theta^{(G)}, \Theta^{(D)}) \tag{6.4}$$

Equation 6.4: The Generator minimax equation.

Randomly, during the training process, fake (synthetic) data will be sent to discriminator with a label of 1.0 (not 0.0), which is normally for real data. By maximizing with respect to $\Theta^{(D)}$, the optimizer sends gradient updates to the discriminator parameters to consider this synthetic data as real. At the same time, by minimizing with respect to $\Theta^{(G)}$, the optimizer will train the generator's parameters on how to outwit the discriminator. However, in practice, the discriminator is confident in its prediction on classifying the synthetic data as fake and will not update its parameters. Furthermore, the gradient updates are small and will diminish significantly as they propagate to the generator layers. As a result, the generator fails to converge:

The solution is to reformulate the loss function of the generator such that:

$$\mathcal{L}(\Theta^{(G)}, \Theta^{(D)}) = -\mathbb{E}_z \log D(G(z)) \tag{6.5}$$

Equation 6.5: Reformulated Generator loss function.

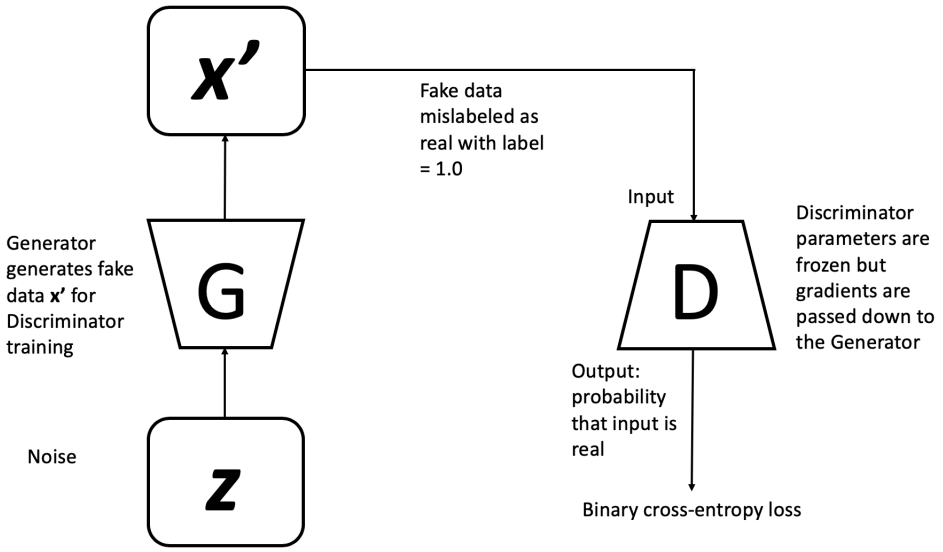

Figure 6.3: Training the generator is basically like any network using a binary cross-entropy loss function. The fake data from the generator is presented as genuine.

The loss function maximizes the chance of the discriminator into believing that the synthetic data is real by training the generator. The new formulation is no longer zero-sum and is purely heuristics-driven. Figure 6.3 shows the generator during training. In this figure, the generator parameters are only updated when the whole adversarial network is trained. This is because the gradients are passed down from the discriminator to the generator. However, in practice, the discriminator weights are only temporarily frozen during adversarial training.

For deep learning methods, both the generator and discriminator can be implemented using a suitable neural network architecture. If the data or signal is an image, both the generator and discriminator networks will use a Convolutional Neural Network (CNN).

As previously stated, as the training progresses, the discriminator will eventually no longer be able to be able to distinguish between synthetic and real data. At which point, the discriminator can be discarded, and the generator can be used to generate synthetic data that is comparable to real data. As related to this project, two GANs are trained resulting in two trained generators. One generator, from the DCGAN, can

be used to generate heightmaps based on values sampled from the latent space the generator was trained on. The second generator, from the pix2pix GAN, is a image translation GAN that translates supplied images, such as heightmaps, to another image, in this case texture maps.

The following sections utilize outputs from trained GANs that were created as part of this project. The code snippets are from a Jupyter notebook that was also created as part of the project. The individual sections demonstrate the utility of using GANs for terrain generation.

6.2 Generating Synthetic Heightmaps

During the implemented training process, Keras training loop will periodically export the generator model in .h5 format. Models that produce desirable results can be used to generate new synthetic images. The trained model is imported using the code shown below.

```
# example of loading a model
from keras.models import load_model


# load model
model = load_model('generator_model_040.h5')
```

Since the method used to generate heightmaps is a DCGAN (Figure 6.4), the input to the generator is a vector of values, or rather points, from the latent space learned by the generator during training. The latent space is a Gaussian distribution of length n.

A function to generate points in the latent space can be done as such.

```
# example of generating latent points
from numpy.random import randn
```

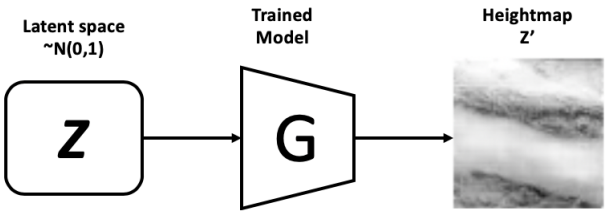

Figure 6.4: Points from n-dimensional latent $\sim N(0, 1)$ space from are sent to the generator to create a heightmap.

```
# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):

    # generate points in the latent space
    x_input = randn(latent_dim * n_samples)

    # reshape into a batch of inputs for the network
    z_input = x_input.reshape(n_samples, latent_dim)

    return z_input
```

To view (or save) images, a function using Python's matplotlib pyplot library can be used. Since the generator was trained on 1-channel grayscale images, an image slice is added to account for the missing dimension that matplotlib pyplot expects. The color cmap is specfied as gray.

```
# example of ploting a single image
import matplotlib.pyplot as plt

# create a plot of generated images
def plot_generated(example):
```

```
    # define subplot
    plt.subplot(1, 1, 1)

    # turn off axis
    plt.axis('off')

    # plot raw pixel 1-channel data
    plt.imshow(examples[ :, :, 0], cmap="gray")

    plt.show()
```

The complete example to generate multiple images as a 5 x 5 image matrix using a latent space point size of 250 is shown below.

```
# example of loading the generator model and generating images
from numpy.random import randn
from keras.models import load_model
import matplotlib.pyplot as plt

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):

    # generate points in the latent space
    x_input = randn(latent_dim * n_samples)

    # reshape into a batch of inputs for the network
    z_input = x_input.reshape(n_samples, latent_dim)
```

```
    return z_input

# create a plot of generated images
def plot_generated(example):

    # define subplot
    plt.subplot(1, 1, 1)

    # turn off axis
    plt.axis('off')

    # plot raw pixel 1-channel data
    plt.imshow(examples[ :, :, 0], cmap="gray")

    plt.show()



# load model
model = load_model('generator_model_040.h5')

# generate images
latent_points = generate_latent_points(250, 25)

# generate images
X = model.predict(latent_points)

# plot the result
```

```
plot_generated(X, 5)
```

Running the example, first loads the saved model. Next, 25 random $\sim N(0, 1)$ points from a 250 dimensional latent space are created and supplied to the generator model to create 25 heightmap images, which are displayed in Figure 6.5 as a 5×5 grid.

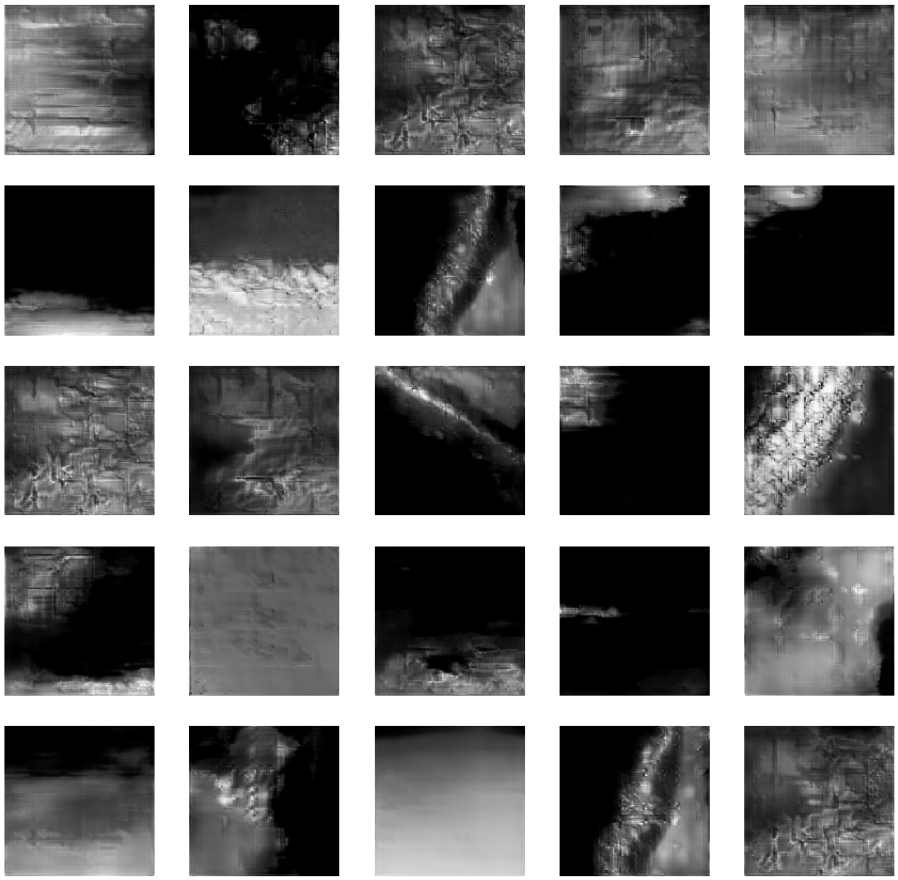

Figure 6.5: Randomly generated heightmap images from a trained DCGAN generator.

6.3 Image Interpolation

Interpolation can be used to transform one heightmap image into another heightmap image. This technique can prove useful when one is looking to create a single image with shared characteristics between two images. An output using linear interpolation

such as the one shown in Figure 6.6 demonstrates the unique characteristics that are potentially created when using this approach. Also this approach can be used blend images to assist in graceful terrain transitions when needed. A interpolation utility is included the Jupyter notebook which uses linear and spherical linear interpolation (Slerp) (Dam et al., 1998) to perform interpolation between two images.

Figure 6.6: Interpolating between two images to produce 'in-between' images with shared characteristics.

6.4 Exploring the latent space of the trained DC-GAN Generator

Generative Adversarial Networks (GANs) are comprised of mainly unsupervised models, although there are exceptions in the broad set of model. Therefore, generating images with certain characteristics or assigned to a particular class can prove to be difficult. As with several GAN model architectures, the generator utilizes a latent space which is high dimensional Gaussian manifold space. As training proceeds, the generator assigns meaning to the latent space and utilizes a vector of points from that learnt latent space to produce images (Figure 6.4). It is often desirable to send different random vectors to the latent space and examine the resulting.

Instead of a purely random trial-and-error approach to exploring the latent space, in the important paper *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*, Radford et al. (2015) introduced two main approaches for systemically exploring the latent space of a generator. The first, is the use of vector arithmetic on the latent space to generate images with certain charac-

teristics or attributes by creating groups of similar images and performing addition or subtraction between groups. The second is using a vector of two points to generate *in-between* images that share a linear path between the two vectors. Producing a similar result to linear interpolation from the proceeding section. Using the first approach provides a path similar to that of classification models by utilizing known labels (*the class*) to obtain desired images in a semi-controlled manner.

6.4.1 Vector Arithmetic in Latent Space

The latent space used by the generator from a GAN, is an n-dimensional hypersphere represent by a Gaussian standard distribution, $\sim N(0, 1)$, with a mean of zero and a standard deviation of one. As stated previously, values from the latent space are sent to the generator to generate images. One can send a vector of n-dimensional points, such as all 0's or all 1's, even all 0.5's for that matter, to in essence perform a query on the latent space to generate an image. Taking the concept further, using the points of two vectors that share a linear path, it is possible to generate a series of images that transition from the starting vector to the ending vector as shown by Radford et al. (2015).

This establishes the idea of using vector arithmetic to construct images with, or without, certain image characteristics. Radford et al. (2015) demonstrates this approach by using a GAN trained on a dataset of celebrity faces. As one example, a three groups of images with particular characteristics were created from the latent space, a face of a smiling woman, the face of a neutral woman (not smiling), and the face of the a neutral man. This was then represented as the following equation:

$$[smilingwoman] - [neutralwoman] + [smilingman] = ??$$

The result, as shown is Figure 6.7, are images of a smiling man.

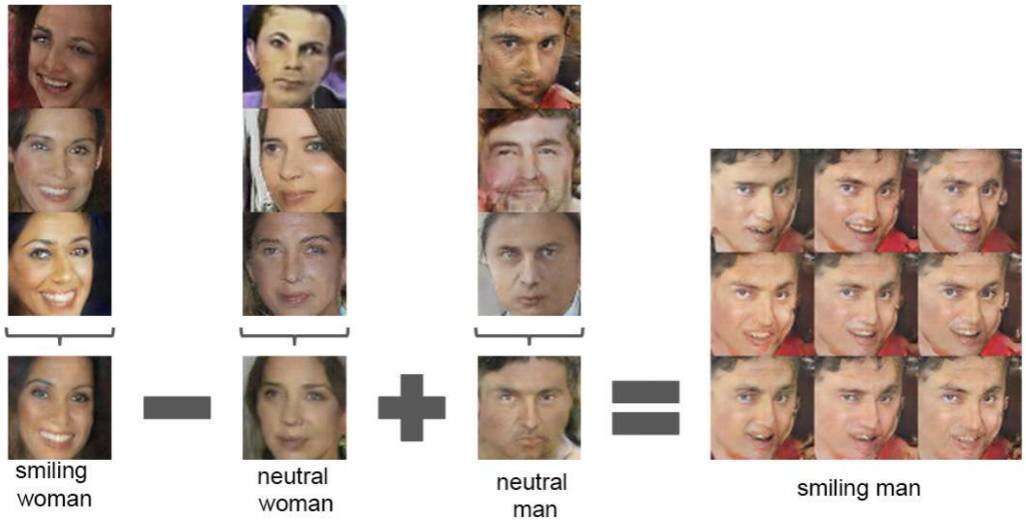

Figure 6.7: Example of Vector Arithmetic on Points in the Latent Space for Generating Faces with a GAN (Radford et al., 2015).

6.4.2 Generate Heightmaps using Vector Arithmetic

Using the refined techniques for exploring the latent space as shown by White (2016), one can perform vector arithmetic on the generator trained on heightmap images to generate heighmaps with desired characteristics. First start by generating a random set of 100 images by sending random vectors to the latent space of the generator. The latent vectors sent to to the generator is stored in an indexed format.

```
# example of storing latent points
from numpy import savez_compressed

...

# generate points in latent points
latent_points = generate_latent_points(250, 100)

# save points
savez_compressed('latent_points.npz', latent_points)

...
```

The latent vectors are saved to a compressed Numpy file. The resulting images

are shown in Figure 6.8.

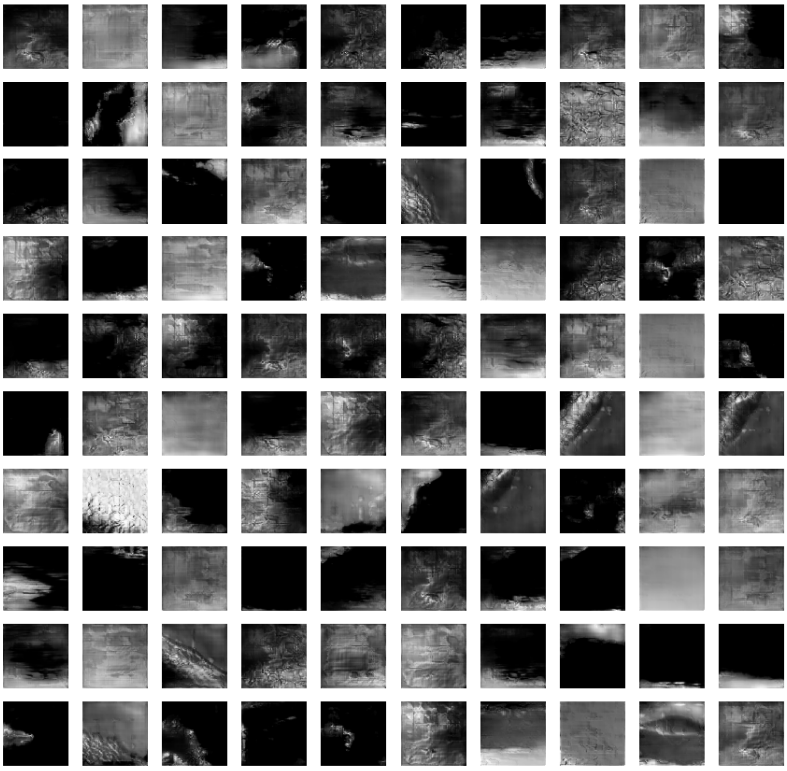

Figure 6.8: 100 heightmaps generated from a trained DCGAN generator.

Each image has an index value which can be retrieved from the compressed Numpy file for reuse. We can explore the images an look for interesting terrain characteristics, such as a ridge. The three images shown in Figure 6.9, have ridge properties.

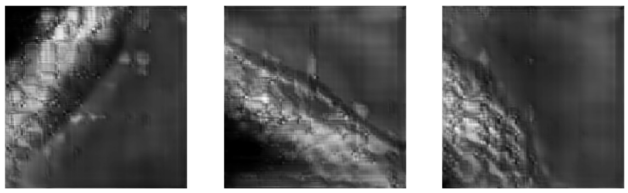

Figure 6.9: Three generated heightmap images showing the terrain characteristics of a ridge with index values *58, 83, 26*.

From this we form a group out of these images and assign an variable to the index of each image in order to retrieve the latent vectors. The values of the latent vectors from the *ridge* group are averaged together, creating a latent vector that represents the three *ridge* images.

```
# example of retrieving latent points and creating a latent vector
↪  using averaged values
from numpy import load
from numpy import mean
from numpy import vstack


# retrieve specific points
ridge_ix = [58, 83, 26]
# load the saved latent points
data = load('latent_points.npz')
points = data['arr_0']

# average list of latent space vectors
def average_points(points, ix):
    # convert to zero offset points
    zero_ix = [i-1 for i in ix]
    # retrieve required points
    vectors = points[zero_ix]
    # average the vectors
    avg_vector = mean(vectors, axis=0)
    # combine original and avg vectors
    all_vectors = vstack((vectors, avg_vector))
    return all_vectors
...

# average vectors
ridge = average_points(points, ridge_ix)
# combine all vectors
all_vectors = vstack((ridge, cliff_edge, high_ground))
# generate images
images = model.predict(all_vectors)
# scale pixel values
images = (images + 1) / 2.0
plot_generated(images, 1, 4)
```

The latent vector created from the latent points is used to represent the ridge

group in the vector arithmetic equation. The resulting image from the average latent vector is shown in Figure 6.10.

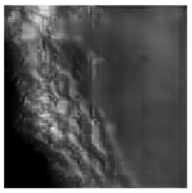

Figure 6.10: Generated heightmap from the average latent vector.

Now suppose we would like to create a latent vector with terrain characteristics of a ridge that transitions to a flat high ground as a plateau. We can formulate an equation such as:

$$[ridge] - [cliffedge] + [highground] \stackrel{?}{=} [plateau]$$

Repeating the steps from above, creating the *cliffedge* and *highground* groups respectively, the following code will yield an image, shown in Figure 6.11 that transitions from lower ground to higher flat ground.

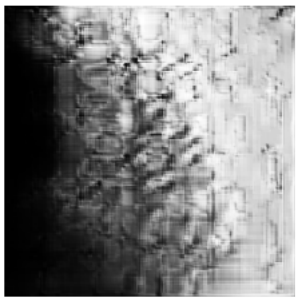

Figure 6.11: Image created from Vector Arithmetic on the Latent Space.

```
from numpy import expand_dims
...
# ridge - cliff_edge + high_ground = ?
result_vector = ridge[-1] - cliff_edge[-1] + high_ground[-1]
# generate image
```

```
result_vector = expand_dims(result_vector, 0)
result_image = model.predict(result_vector)
# scale pixel values
result_image = (result_image + 1) / 2.0
plt.axis('off')
plt.imshow(result_image[0][:,:,0], cmap="gray")
plt.show()
```

6.5 Generating synthetic texture maps

Generating synthetic maps using pix2pix differs slightly from the DCGAN. pix2pix is a known as a conditional GAN, or cGAN, where the image output from the generator is conditional to it's input image (Figure 6.12) rather than input from the latent space, which the pix2pix does not have.

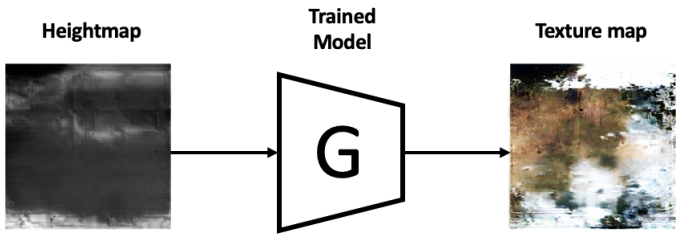

Figure 6.12: Using a heightmap image as input to generate a texture map image from a trained pix2pix generator.

As we did with the DCGAN, we can load a trained model to generate new synthetic images. In this case the generated image is saved as a bitmap image file. The bitmap file format is a common format used in terrain generation software packages. The following code takes heightmap input as a 1-channel Numpy array along with an index value, to correspond with the heightmap's index value as a pairing reference in the terrain generation software.

```
# example of image generation using pix2pix
from keras.models import load_model
from cv2 import merge
```

```
from scipy.misc import imsave
import matplotlib.pyplot as plt

# covert grayscle 1-channel to RGB 3-channel
def gray2rgb(gray_img):

    # convert to 3-channel image
    img = merge((gray_img,gray_img,gray_img))

    return img

# create a plot of generated images
def plot_generated(example, image_ix):

    # retrieve raw pixel data
    arr = example[0, :, :, :]

    # Display image
    plt.axis("off")
    plt.imshow(arr)

    # Save as BMP
    imsave('generated_texture' + str(image_ix) + '.bmp', arr)

# load model
model = load_model('model_000030.h5')

n = X.shape[1]

# Use X array as input and convert to 3-channel
X_3ch = gray2rgb(X).reshape(1, n, n, 3)

# generate image from source
gen_image = model.predict(X_3ch)

# plot all three images
plot_generated(gen_image, image_ix)
```

6.6 Rendering using Amazon Lumberyard

Using a terrain generation software allows for combining the heightmap with the texture map. Generally speaking, the heightmap is imported into the terrain editor

which will generated the polygon wire mesh needed to combine the texture map using a rendering algorithm such as tessellation. As an example shown in Figure 6.13, using a video game creation tool called Amazon Lumberyard, the terrain editor can be used to load the heightmap and create the polygon mesh.

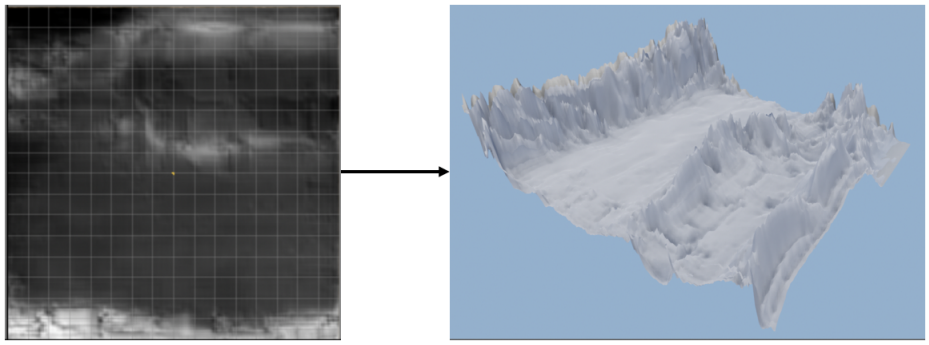

Figure 6.13: Using Amazon's Lumberyard terrain editor to create a polygon mesh from a heightmap generated by a DCGAN.

Next, the texture map is then overlayed (rendered) over the polygon mesh to produce the final outcome shown in Figure 6.14 .

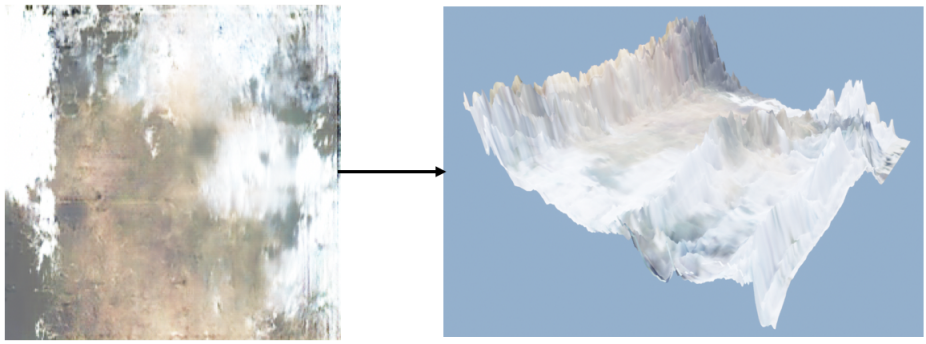

Figure 6.14: Using Amazon's Lumberyard terrain editor to render realistic terrain from a texture map generated by a pix2pix GAN.

6.7 Chapter Summary

In this chapter is has been shown it is possible to create synthetic terrain from Generative Adversarial Networks (GANs). By understanding the relationships between the

input to the DCGAN's generator and resulting output, allows one to manipulate the generator to achieve desirable characteristics in output images in a seemingly random process. Although not covered in this chapter, the sometimes pixelated properties of the images are eventually smoothed away during the rendering process. Flaws in the images can actually be used in inventive ways when it comes to designing worlds for video games. After the initial training phase concludes, a large advantage when using GANs to design synthetic terrains, is the speed at which both realistic heightmaps and texture maps can be created and implemented into a video game production pipeline.

Chapter 7

Summary and Conclusions

In this chapter the success of meeting the proposed project goals as proposed in Chapter 1 are reviewed. The bulk of this chapter is dedicated to Goal #2, where the results and analysis for the different DCGAN architectures and pix2pix GAN are shown. Finally, suggestions for future work and conclusions are made.

7.1 Goal #1 Results

Goal: Port the original project to use Keras and Tensorflow.

The project was recreated using Keras and Tensorflow with one exception which will be discussed below. The original project was written using Keras, Lasagna, and Theano. The use of Theano is limited to only Linux OS systems. With the deep learning landscape changing rapidly, it is important to not only use modern modeling techniques for improved outcomes, but also utilize improved frameworks and toolkits. One method for evaluating the deep learning take up of new toolkits by researchers and developers is to analyze Github activity, Stack Overflow activity, as well as Google search results. As can be seen in Table 7.1, the top five tookits from Thedataincubator (2017), Theano, although widely used, has fallen behind Tensorflow. Lasagne, which

is often paired with Theano, is also down in the rankings.

Library	#	Overall	Github	Stack Overflow	Google Results
tensorflow	1	10.868	4.253	4.372	2.243
keras	2	1.928	0.613	0.8304	0.484
caffe	3	1.855	1.002	0.302	0.552
theano	4	0.757	-0.157	0.3621	0.552
pytorch	5	0.481	-0.198	-0.302	0.982
...	...	...	...	...	...
lasagne	15	-1.106	-0.381	-0.288	-0.437

Table 7.1: Deep Learning Toolkit Rankings (Thedataincubator, 2017).

One feature that was not recreated in the project, was the ability to use image sizes other than 256×256. The original author's paper, although not utilized, did contain this feature which was mainly handled though the lasagne module. Since my project was set to only use image sizes of fixed length, I did not feel the need to include this feature although it could be introduced with only minor architectural changes.

7.2 Goal #2 Results

Goal: Include updated modeling techniques to DCGAN and pix2pix for quality improvements to generated images.

Since the original author's paper was published, updated techniques and architectures have been recommended as reference in the Design chapter. These include using **LeakyReLU** and the hyperbolic tangent or **tanh** function. One technique that was tested to see if there was a difference in the quality of output by the generator of the DCGAN, was additional of the Gaussian Noise layer to the discriminator as the input layer compared to a standard DCGAN input layer architecture.

7.2.1 Evaluation Method

The performance of both the DCGAN and the pix2pix GAN are evaluated by analyzing the training loss and image quality of the output images from the generator. The training loss of the discriminator with real images and the loss with fake images shows the performance of the discriminator detecting the appropriate class at each training step. Likewise the training loss for the generator records the performance the generator's attempts to *fool* the discriminator. It should be noted, improved training loss is not necessarily correlated with image image quality (Arjovsky et al., 2017). Therefore a metric do determine image quality is also necessary for ech GAN.

Since the DCGAN utilizes a learned latent space to generate images, there is no *truth* image to compare the synthetic image against for image similarity. In this case empirical methods must be used. To assess the performance of the DCGAN, a Turing test was completed using humans to classify images as *real* or *fake*.

The pix2pix GAN differs from the DCGAN, whereby image pairs are delivered to the GAN for translation. These means a *truth* image is available to compare with a synthetically generated image when using real training or validation images from the data set. The Mean Squared Error metric and the Structural Similarity metric (Wang et al., 2004) are used to assess image quality.

7.2.2 DCGAN Results

The input to the DCGAN, as implemented in Chapter 4, consists of 256 x 256 pixel one color channel (grayscale) heightmap images. In this experiment, we are mainly evaluating the output of images from the generator when using a discriminator with a Gaussian Layer as the first layer in the discriminator, and without a Gaussian Layer as the first layer.

7.2.3 Baseline DCGAN Results

Total training time: $\sim$ 18 hours

Hyperparameters

Number of epochs: 100

Number of number images per batch: 20

Learning rate: 0.0002

Optimization algorithm: Adam

Loss function(s): binary crossentropy

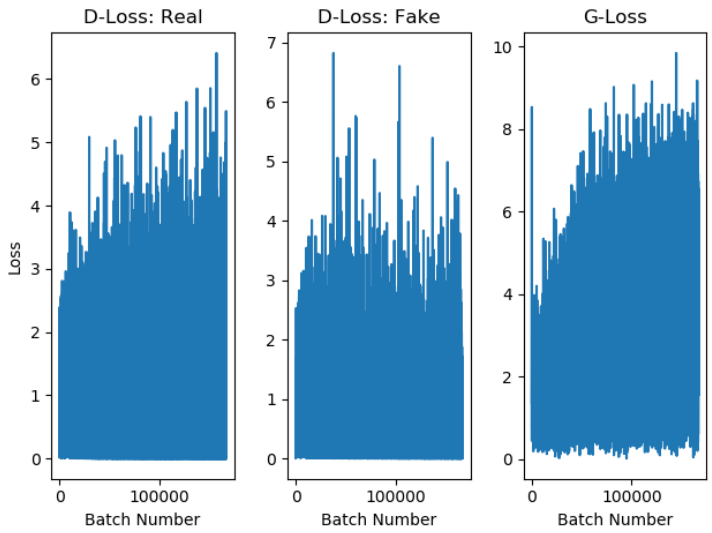

Figure 7.1: The discriminator and generator loss of the baseline DCGAN.

Looking at the loss values for the discriminator in Figure 7.1, losses increase as training continues throughout the training run when the discriminator is evaluating real images. When evaluating fake images, the discriminator seems to stabilize but with high variability. The discriminator is better at detecting fake images vs. real images. As for the generator, training loss values increase throughout the training process and the generator never stabilizes during the training run.

7.2.4 DCGAN with Gaussian Noise Results

Total training time: $\sim$ 21 hours

Hyperparameters

Number of epochs: 100

Number of number images per batch: 20

Learning rate: 0.0002

Optimization algorithm: Adam

Loss function(s): binary crossentropy

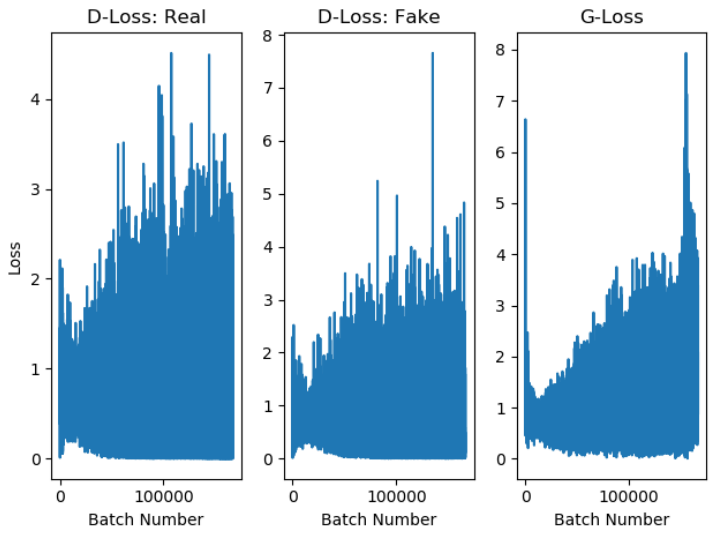

Figure 7.2: The discriminator and generator loss of the DCGAN with Gaussian noise.

Starting with the discriminator, losses seem to decrease early on, but start to increase for the remainder of the training run. The discriminator performs about the same when detecting real images vs. fake images as the losses are less for real images. The generator loss also decreases early in training run but increases for the rest of the training run as it struggles to fool the discriminator. The generator, just as with the baseline generator, does not appear to converge which suggests some training

instability.

7.2.5 DCGAN Analysis

Figure 7.3 shows a random sample of real heightmaps from the satellite data set.

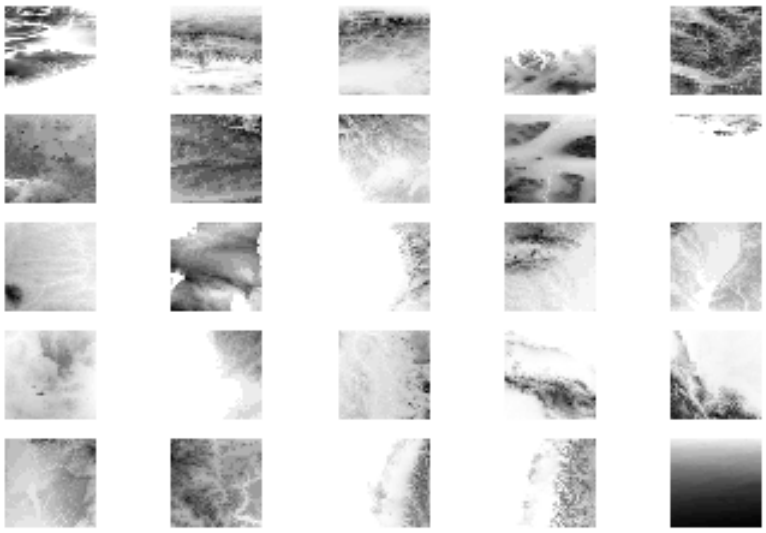

Figure 7.3: Sample of real satellite heighmaps used in DCGAN training data set after 100 epochs.

7.2.6 DCGAN Baseline DCGAN Analysis

Using manual inspection of the images with the baseline DCGAN, after a single epoch as shown in Figure 7.4, shows a lack of diversity in image generation along with some *crosshatching* patterns. At 50 epochs, as shown in Figure 7.8, the generator shows plausible terrain and the image quality continues to improve. There does appear to some diversity with some images having good terrain like characteristics while other images appear to be more smooth. Training concludes at 100 epochs, Figure 7.9 where one can observe plausible terrain with defined features, however some patterns

begin to appear as can be seen in the images in second column starting from the left.

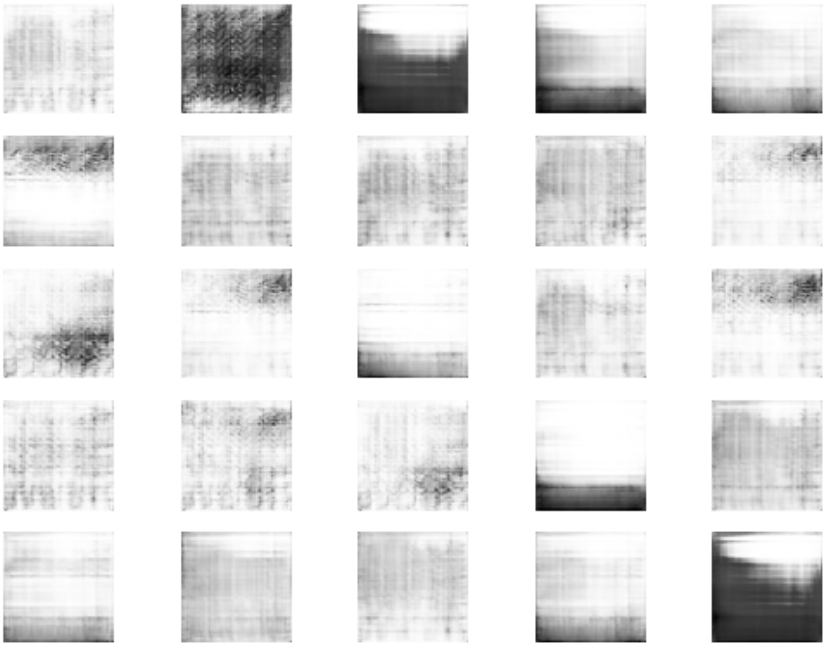

Figure 7.4: Generated heightmap after 1 epoch.

7.2.7 DCGAN with Gaussian Noise Analysis

With manual inspection of the DCGAN with Gaussian noise added as the first layer in the discriminator, after a single epoch as shown in Figure 7.7, shows a lack of diversity in image generation along with some *crosshatching* patterns just as the baseline model. At 50 epochs, as shown in Figure 7.8, the generator starts to develop plausible terrain images when compared to the real images as shown in Figure 7.3. Training concludes at 100 epochs, Figure 7.9 where one can observe plausible terrain with defined features, however some *crosshatching* patterns begin to appear as can be seen in the images in the lower right hand corner. Models from epoch 70 to 100 would generate plausible terrain heightmaps. It appears training after 100 epochs

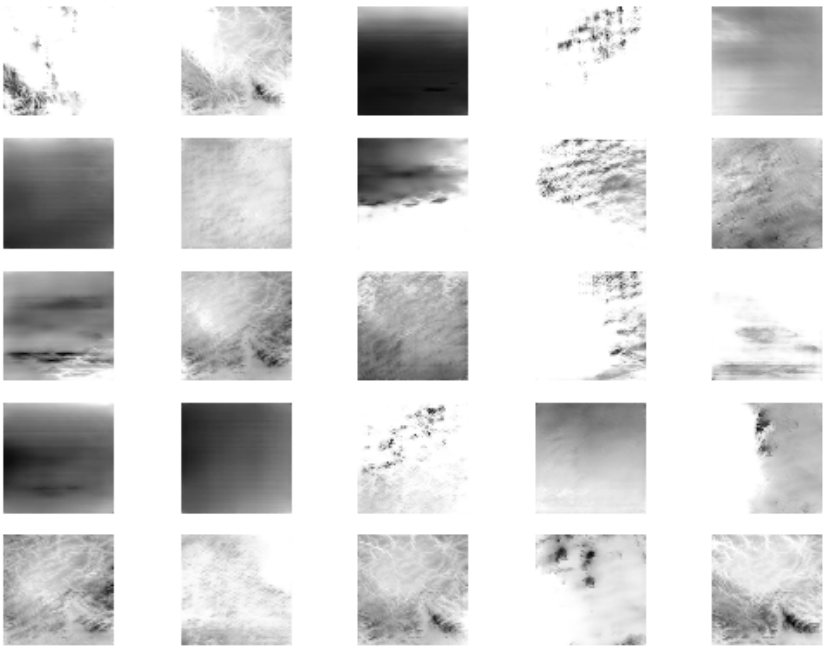

Figure 7.5: Generated heightmap after 50 epochs.

may not be warranted as the image generation begins to degrade.

7.2.8 Turing Test between Baseline DCGAN and Gaussian Noise Layer DCGAN

As discussed previously, the generator of the DCGAN uses a learned latent space to generate images. As such, there is no *truth* image to compare the similarity or quality to a generated synthetic image. Rating the output images therefore is reduced to a few assessment techniques. One is empirical review by humans in the form of a Turning test.

For this project a Turing Test was completed with 7 individuals. Two groups of images were used. Each group consisted of five real images and five synthetically generated images from each GAN type. The first group of images was from the

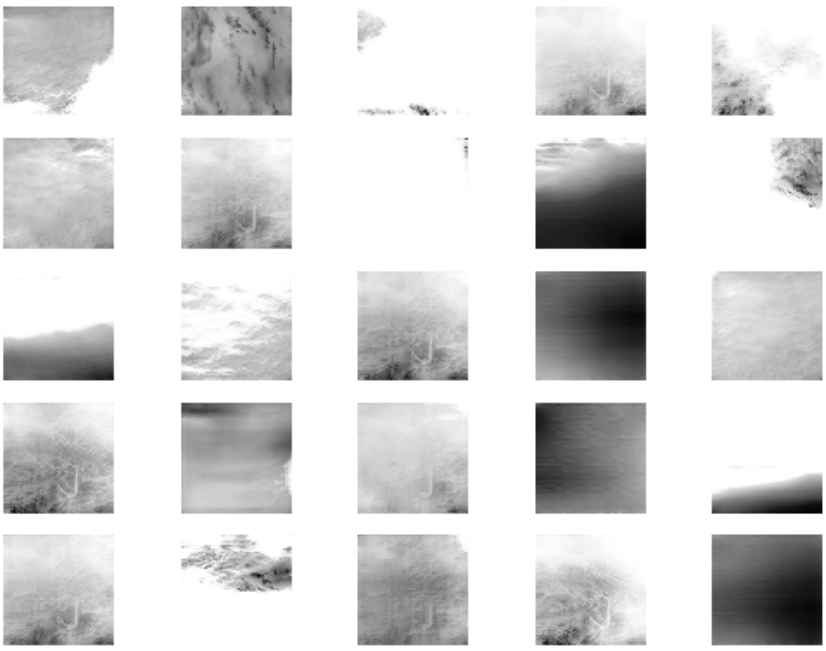

Figure 7.6: Generated heightmap after 100 epochs.

Baseline DCGAN and the second was from the added Gaussian Noise layer DCGAN. The starting group of images for each individual to view was chosen at random. The mean accuracy and sensitivity was used as the performance metric. Mean accuracy was determined by the formula shown in Equation 7.1, where n is the total of human participants.

$$\text{Mean Accuracy} = \frac{\Sigma_{i=1}^{n}(true\ positives + true\ negatives)_i}{n} \tag{7.1}$$

Equation 7.1: Equation for Mean Accuracy.

The sensitivity formula in Equation 7.2 was used to calculate the ratio of synthetic (fake) images classified as fake, where m is the total number of fake images in the group. This number is arguably the best determination of either DCGAN performance as a low sensitivity ratio would show the generator is fooling the human

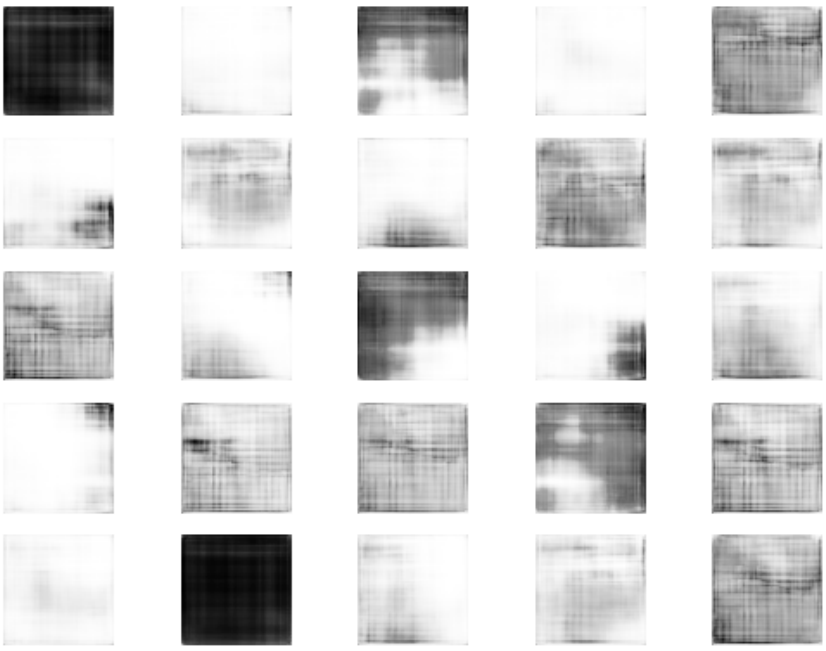

Figure 7.7: Generated heightmap after 1 epoch.

judges, when a fake image is classified as real.

$$\text{Sensitivity} = \frac{\Sigma_{i=1}^{m}(true\ positives)_i}{m} \tag{7.2}$$

Equation 7.2: Equation for Sensitivity.

Table 7.2 shows the results of accuracy, overall correctness whether an image is real or fake, and sensitivity, a fake image being classified as real. As the table shows, both DCGANs performed about the same between the two groups of images. It is also should be noted the relatively high sensitivity scores shows humans where rather adept at classifying actual fake images as fake.

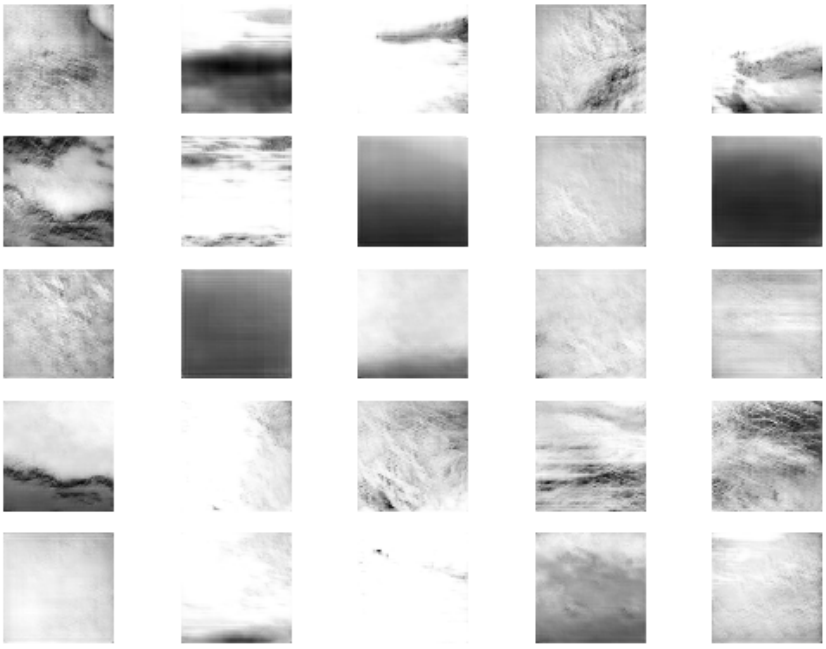

Figure 7.8: Generated heightmap after 50 epochs.

Metric	Baseline DCGAN	Gaussian Noise DCGAN
Mean Acc.	72.86	68.57%
Sensitivity	71.43%	68.57%

Table 7.2: Mean accuracy and sensitivity between two DCGAN architectures, the baseline DCGAN and the DCGAN with added Gaussian Noise input layer.

7.2.9 pix2pix Results

There were no changes between the pix2pix used in the original paper and the one implemented in project.

Total training time: $\sim$ 50 hours

Hyperparameters

Number of epochs: 30

Number of number images per batch: 20

Learning rate: 0.0002

Optimization algorithm: Adam

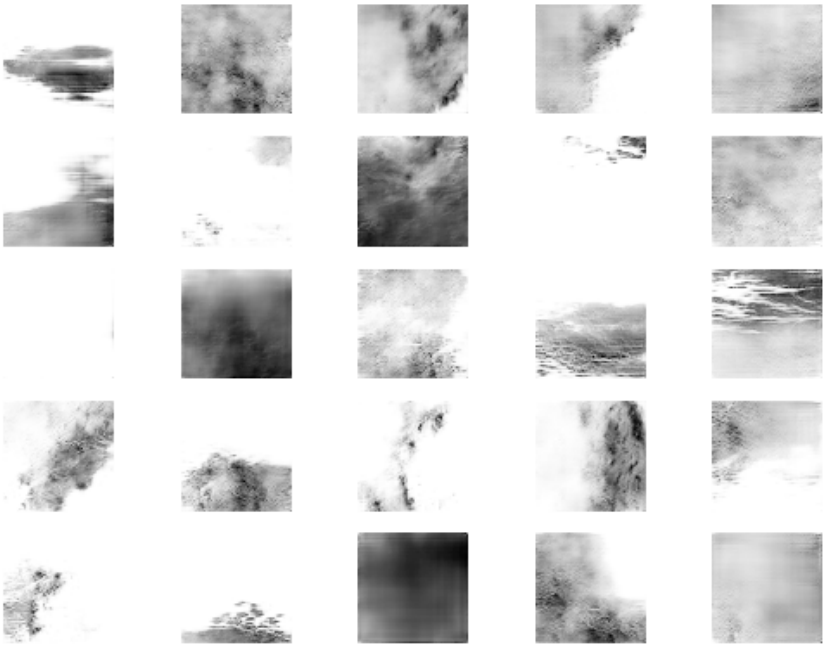

Figure 7.9: Generated heightmap after 100 epochs.

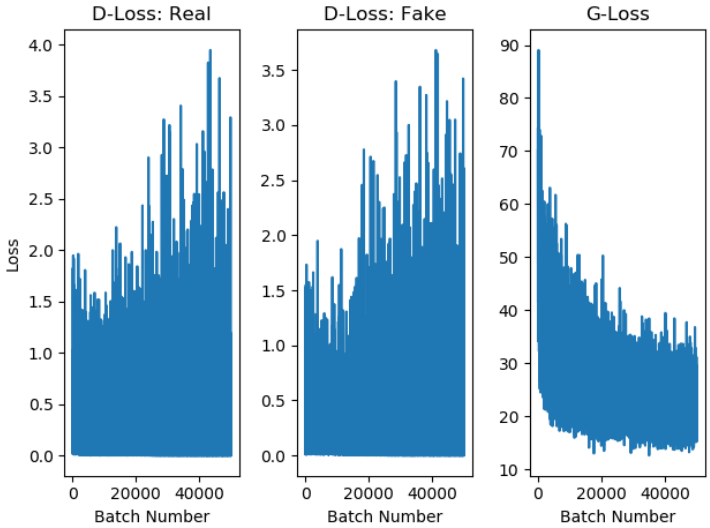

Figure 7.10: The discriminator and generator loss of the pix2pic GAN.

Figure 7.10 shows the training loss for the discriminator, real vs. fake, and the

generator loss. Starting with the discriminator loss, one can see the loss for the discriminator during training increases over time for both real images and synthetic images at about the same rate. This shows the discriminator basically did no better, or worse, at detecting real images from fake. Additionally, as confirmed by the generator loss, the performance of the discriminator at detecting fake images becomes worse over time. As for the generator loss, the generator improved over time as the generator becomes better at fooling the discriminator. It appears the loss starts to level off, indicating perhaps training should continue beyond 30 epochs to see improvements to the generator continue.

7.2.10 pix2pix Analysis

Manual inspection of image quality as translated by the generator at epochs 1, 15, and 30, reveal gradual improvement of the generator during the training process as shown in Figures 7.11 through 7.13.

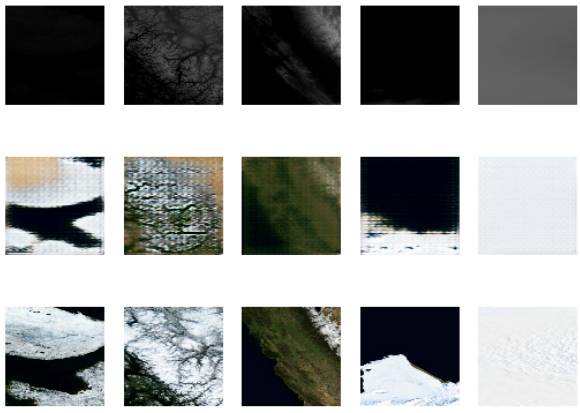

Figure 7.11: After 1 epoch, top row is the heightmaps as input, middle row is the generated image, bottom row is the target image.

For pix2pix, one can compare similarity between the translated image and the target (truth) image by using Mean Squared Error (MSE) and Structural Similarity

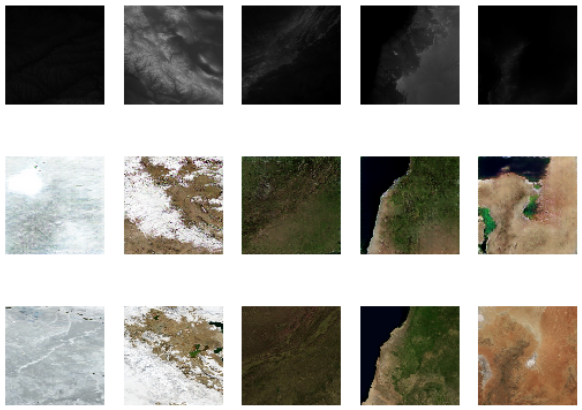

Figure 7.12: After 15 epochs, top row is the heightmaps as input, middle row is the generated image, bottom row is the target image.

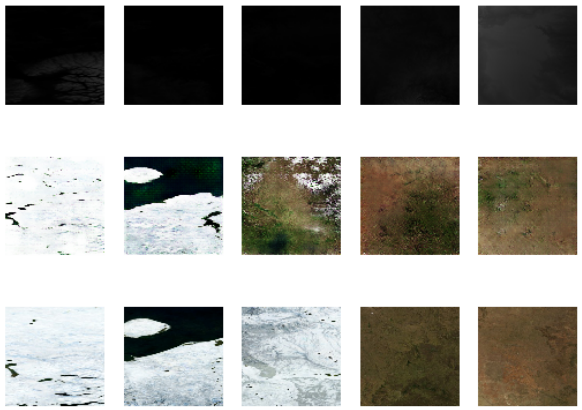

Figure 7.13: After 30 epochs, top row is the heightmaps as input, middle row is the generated image, bottom row is the target image.

Index (Wang et al., 2004). Table 7.3 shows the MSE and Structural Similarity Index values for a set of random images using trained models from epochs 1, 15, and 30. An MSE value of 0 implies an exact copy. The greater the MSE value, the less similar Image A is to Image B. With respect to MSE, Table 7.3 shows mixed results for the different images between epochs. It appears the models trained at different epochs

perform better at certain types of terrain than others as the minimum MSE values fluctuate between a given image and a given model. For the SSIM, the range of output values are between -1 and 1, where 1 implies an exact copy. Reviewing the results in Table 7.3, the values are less mixed, the model at epoch 15 appears to be the best generalized model as is has the highest SSIM values for all images except for one.

	Epoch 1				
Image #	1	2	3	4	5
MSE	0.255	2.052	3.948	2.255	1.298
SSIM	0.648	0.115	-0.0963	-0.100	0.330
	Epoch 15				
Image #	1	2	3	4	5
MSE	0.314	1.926	0.783	0.490	0.582
SSIM	0.732	0.151	0.082	0.429	0.466
	Epoch 30				
Image #	1	2	3	4	5
MSE	0.404	1.319	1.061	0.144	1.349
SSIM	0.657	0.131	0.031	0.572	0.340

Table 7.3: Generated Image vs. Target Image similarity scores using mean squared error (MSE) and Structural Similarity Index (SSIM).

7.3 Goal #3 Results

Goal: Provide utility notebooks in Jupyter that are used to manage image sets and explore the rendered images after training is complete.

This part was mostly covered in Chapter 6, where I demonstrated generating synthetic heightmaps and texture maps from trained GAN generators. Exploring the latent space using a Jupyter notebook demonstrated control over a seemingly random process which allows one to generate heightmaps with certain characteristics. Then in the final section of chapter 6, the application of a using a synthetic heightmap to

generate a synthetic texture map and applying the use of both maps using a terrain editor.

Additionally, a data set creation utility with some image exploration tools were also created and placed into a Jupyter notebook. This allows the user interactive feedback with the image data by allowing the to user the data set and explorer the terrain properties.

7.4 Future Work

As with most projects, there are always room for additional contributions. There are two main areas for exploration. The first, would be to create a data set that is more latitude specific. It's clear from analyzing the resulting images from the pix2pix GAN, latitude plays a role in output. The terrain of mountains at latitudes near the poles are generally covered in snow, whereas mountains near the equator for example, the occurrence of snow is much less, if ever. Therefore, I would construct latitude based data sets and train models specifically on a latitude specific data set. The author's in the original paper do not mention this idea directly, but when reviewing source code you can see attempts made to construct a data set based on the overall color of the terrain.

The second would be determine optimal hyperparameters for training to improve image quality and model stability. This would prove difficult as times for model training can be rather lengthy. Given the proper resources however, it would be possible to train several models on a distributed system such that models are trained in parallel to cut down on the time requirement.

7.5 Conclusion

This project made extensive use of Generative Adversarial Networks (GANs) to generate synthetic terrain. Two DCGANs with different architectures were used to generate heightmaps from training images. One DCGAN received a change to the input layer by adding Gaussian Noise as a potential improvement to output image quality. The results showed the addition of the Gaussian Noise layer made no difference in training loss or training stability when compared with the baseline DCGAN. A Turning test was performed to assess image quality. The Turning test results showed humans guessed the correct image label, *real* or *fake*, 72.66% percent of the time with the baseline GAN and 68.57% of the time with the DCGAN with Gaussian Noise as the input layer. Humans were also able to determine that an image was indeed *fake* 71.43% of the time with the baseline DCGAN and 68.57% of the time with DCGAN with Gaussian Noise as the input layer. It can be concluded the addition of Gaussian Noise did not make a significant difference when compared to the baseline DCGAN.

The image results from the pix2pix GAN were of mixed quality depending on the number of epochs performed and the type of terrain to be generated. Training for the pix2pix generator model was stable and the image results were of significant quality. The Mean Squared Error and Structural Similarity measures were used to assess image quality. Models trained between fifteen and thirty epochs proved to be of acceptable quality for texture map generation. Further experiments with latitude specific data sets would most likely improve the performance of pix2pix GAN, which would eliminate some unusual learned anomalies such as including snow in terrain where it clearly does not belong. Also the elimination of the ocean images as input, except in coastal areas, would also prove beneficial, since the Earth is mostly covered in water, which biases the pix2pix model for low level areas.

Although procedural terrain generation techniques using fractals and noise techniques will undoubtedly continue to dominate the video game graphics generation

space, Artificial Intelligence (AI) based techniques in video games will continue to make inroads beyond Non-Player Character (NPC) interaction. Game development pipelines will need to integrate AI based techniques to improve wider adoption. The application of Generative Adversarial Networks will also need to mature beyond canonical examples in order to expose GANs to a wider domain of potential applications. Wider exposure will indeed generate new ideas and research topics that ultimately improve GANs as a proven image generation technique. Overall, the project demonstrated the use of terrain created by a GAN as a replacement, or at least augment, terrain created by procedural generation techniques. I believe the big advantage to using this method is the speed at which terrain can be created once trained models become available. Also, through making use of exploring the latent space, one can add uniqueness to game play by generating maps that are indeed different every time the game is played.

Bibliography

A3r0 (2006a). *A heightmap created with Terragen.*

A3r0 (2006b). *A version of the heightmap rendered with Anim8or and no textures.*

Arjovsky, M., Chintala, S., and Bottou, L. (2017). Wasserstein gan.

Beckham, C. and Pal, C. (2017). A step towards procedural terrain generation with gans.

Capasso, G. (2001). *Mechanical and Hydraulic Behavior of a Rock Fracture in Relation to Surface Roughness*. PhD thesis.

Dam, E. B., Koch, M., and Lillholm, M. (1998). Quaternions, interpolation and animation.

Desai, U. (2018). Keep calm and train a gan. pitfalls and tips on training generative adversarial networks.

Ewin, C. (2015). *Visualization of the Diamond Square Algorithm.*

Fournier, A., Fussell, D., and Carpenter, L. (1982). Computer rendering of stochastic models. *Communications of the ACM*, 25(6):371–384.

Goodfellow, I. (2016). Nips 2016 tutorial: Generative adversarial networks.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In Ghahramani,

Z., Welling, M., Cortes, C., Lawrence, N. D., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc.

Hvidsten, M. (2004). The opengl texture mapping survival guide.

Isola, P., Zhu, J.-Y., Zhou, T., and Efros, A. A. (2016). Image-to-image translation with conditional adversarial networks.

Kirkby, M. J. (1983). The fractal geometry of nature. benoit b. mandelbrot. w. h. freeman and co. *Earth Surface Processes and Landforms*, 8(4):406–406.

Mao, X., Li, Q., Xie, H., Lau, R. Y. K., Wang, Z., and Smolley, S. P. (2016). Least squares generative adversarial networks.

Perlin, K. (1985). An image synthesizer. *ACM SIGGRAPH Computer Graphics*, 19(3):287–296.

Perlin, K. (2001). Noise hardware. In M. Olano, editor, Real-Time Shading SIGGRAPH Course Notes, chapter 9.

Radford, A., Metz, L., and Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks.

Ronneberger, O., Fischer, P., and Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation.

Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., and Chen, X. (2016). Improved techniques for training gans.

Smith, G. (2015). An analog history of procedural content generation. In *Proceedings of the 2015 Conference on the Foundations of Digital Games*. FDG 2015.

Sund, F. (2014). *Water confined in nanoporous silica*. PhD thesis.

Sønderby, C. K., Caballero, J., Theis, L., Shi, W., and Huszár, F. (2016). Amortised map inference for image super-resolution.

Thedataincubator (2017). thedataincubator/data-science-blogs.

Wang, Z., Bovik, A., Sheikh, H., and Simoncelli, E. (2004). Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612.

White, T. (2016). Sampling generative networks.

Witten, T. A. and Sander, L. M. (1981). Diffusion-limited aggregation, a kinetic critical phenomenon. *Phys. Rev. Lett.*, 47:1400–1403.

Glossary

bitmap A bitmap is a type of memory organization or image file format used to store digital images. 2

class The output category of a data set. Often use in a group of machine learning models called classifiers, or supervised learning. 86

convolution In mathematics (in particular, functional analysis) convolution is a mathematical operation on two functions, f and g, that produces a third function expressing how the shape of one is modified by the other. The term *convolution* refers to both the result function and to the process of computing it. It is defined as the integral of the product of the two functions after one is reversed and shifted. 29

convolutional neural network In deep learning, a convolutional neural network (CNN, or ConvNet) is a class of deep neural networks, most commonly applied to analyzing visual imagery. 19

deep neural network An artificial neural network (ANN) with multiple layers between the input and output layers. The DNN finds the correct mathematical manipulation to turn the input into the output, whether it be a linear relationship or a non-linear relationship. DNNs can model complex non-linear relationships. DNN architectures generate compositional models where the object is expressed as a layered composition of primitives. The extra layers enable

composition of features from lower layers, potentially modeling complex data with fewer units than a similarly performing shallow network. 14

graphics processing unit A programmable logic chip (processor) specialized for display functions. The GPU renders images, animations and video for the computer's screen. Since GPUs perform parallel operations on multiple sets of data, they are increasingly used as vector processors for non-graphics applications that require repetitive computations. 14

heightmap In computer graphics, a heightmap is a raster image used mainly in elevation modeling. Each pixel stores a value, such as surface elevation data, for display in 3D computer graphics. For the purpose of synthetic terrain generation, the heightmap is combined with a texture map to produce a rendered terrain element. 2

interpolation In mathematics, linear interpolation is a method of curve fitting using linear polynomials to construct new data points within the range of a discrete set of known data points. 85

latent space In mathematics and machine learning, a collection of input used by a function that maps input to a projection of output. In other words, the latent space is the space where your features lie. 10

raster In computer graphics, a raster graphics or bitmap image is a dot matrix data structure that represents a generally rectangular grid of pixels (points of color). Raster images are stored in image files with varying formats. 2

synthetic image In terms of data creation, synthetic data can be any method used that is not direct measurement of a data source. As applied to imagery, the creation of 1-channel grayscale or 3-channel RGB values used in image data. 1

unsupervised model Unsupervised model used in machine learning that is a type of self-organized learning that helps find previously unknown patterns in data set without pre-existing labels or class values. 86

vector A vector is an object that has both a magnitude and a direction. Geometrically, we can picture a vector as a directed line segment, whose length is the magnitude of the vector and with an arrow indicating the direction. The direction of the vector is from its tail to its head. 86

Appendix A

Example GAN Using Keras

Below is the complete example code for creating a Generative Adversarial Network (GAN) using the Keras framework as shown in Chapter 4.

```
# train a generative adversarial network on a one-dimensional
↪ function
from numpy import hstack
from numpy import zeros
from numpy import ones
from numpy.random import rand
from numpy.random import randn
from keras.models import Sequential
from keras.layers import Dense

# define the standalone discriminator model
def create_discriminator(num_inputs=2):
    discriminator = Sequential()
    discriminator.add(Dense(25, activation='relu',
    ↪ kernel_initializer='he_uniform', input_dim=num_inputs))
```

```
    discriminator.add(Dense(1, activation='sigmoid'))
    # compile the model
    discriminator.compile(loss='binary_crossentrophy',
    ↪  optimizer='adam', metrics=['accuracy'])

    return discriminator

# define the standalone generator model
def create_generator(latent_dim, num_outputs=2):
    generator = Sequential()
    generator.add(Dense(15, activation='relu',
    ↪  kernel_initializer='he_uniform', input_dim=latent_dim))
    generator.add(Dense(num_outputs, activation='linear'))
    return generator

# define the composite generator and discriminator model, for
↪  updating the generator
def create_gan(generator, discriminator):

    # make the discriminator not trainable when updating the
    ↪  generator
    discriminator.trainable = False

    # combine the two models and compile
    gan = Sequential()

    # add generator to the composite model
```

```
    gan.add(generator)

    # add generator to the composite model
    gan.add(discriminator)

    # compile the composite model
    ganl.compile(loss='binary_crossentrophy', optimizer='adam')

    return gan

# create real samples helper function
def create_real_samples(num):

    # create inputs in [-0.5, 0.5]
    X1 = rand(num) - 0.5

    # create outputs for X^2
    X2 = X1 * X1

    # stack resulting arrays
    X1 = X1.reshape(num, 1)
    X2 = X2.reshape(num, 1)
    X = hstack((X1, X2))

    # create class labels, real = 1
    y = ones((n, 1))
```

```
    return X, y

# create points in latent space as input for the generator helper
↪ function
def create_latent_points(latent_dim, num):

    # create points in the latent space
    x_in = randn(latent_dim * num)

    # reshape into batch of inputs for the network
    x_in = x_in.reshape(num, latent_dim)

    return x_in

# use the standalone generator to create n fake samples, with class
↪ label fake = 0 helper function
def create_fake_samples(generator, latent_dim, num):

    # generate points in latent space
    x_in = create_latent_points(latent_dim, num)

    # predict output
    X = generator.predict(x_in)

    # create class labels, fake = 0
    y = zeros((num, 1))
```

```
    return X, y

# train the generator and discriminator
def train(generator, discriminator, gan, latent_dim, num_epochs=10000,
↪  num_batch=128)

    # calculate half the size of one batch, for training the
    ↪  discriminator
    half_batch = int(num_batch / 2)

    # loop through the number of epochs
    for i in range(num_epochs):

        # retrieve real samples
        x_real, y_real = generate_real_samples(half_batch)

        # retrieve synthetic samples
        x_fake, y_fake = generate_fake_samples(gen_model, latent_dim,
        ↪  half_batch)

        # train discriminator for one epoch
        discriminator.train_on_batch(x_real, y_real)
        discriminator.train_on_batch(x_fake, y_fake)

        # retrieve points in the latent space as input for the
        ↪  generator
        x_gan = generate_latent_points(latent_dim, num_batch)
```

```
        # create inverted labels for the fake samples
        y_gan = numpy.one((n_batch, 1))

        # train the generator using the discriminator's error
        gan.train_on_batch(x_gan, y_gan)

# size of latent space
latent_dim = 5

# create the discriminator model
discriminator = create_discriminator()

# create the generator model
generator = create_generator(latent_dim)

# create the composite gan model
gan = create_gan(generator, discriminator)

# train the model
train(generator, discriminator, gan, latent_dim)
```

Appendix B

Application Code

B.1 Project Folder Organization

The project is organized using the following structure:

```
project
|
+-- images/
+-- latex/
+-- misc/
+-- models/
+-- notebooks/
    +-- Create new maps from generator.ipynb
    +-- Exploring the Latent Space.ipynb
    +-- Image Cropping Utils.ipynb
    +-- Image Similarity.ipynb
    +-- Turning Test Image Generator.ipynb
+-- src/
    +-- dcgan.py
    +-- main.py
    +-- pix2pix.py
+-- README.md
+-- Toward Improving Procedural Terrain Generation with GANs.pdf
```

Table B.1: Project folder structure.

B.2 Source Files

main.py

```
import os
import sys
import argparse
import dcgan
import pix2pix

def main(**kwargs):

    print(kwargs)

    mode=kwargs['mode']
    img_path=kwargs['img_path']
    img_file=kwargs['img_file']
    output_path=kwargs['output_path']
    incl_gan=kwargs['incl_gans']
    dcgan_n_epochs=kwargs['dcgan_n_epochs']
    dcgan_batch_size=kwargs['dcgan_batch_size']
    dcgan_latent_dim=kwargs['dcgan_latent_dim']
    dcgan_incl_noise = kwargs['dcgan_incl_noise']
    dcgan_noise_val = kwargs['dcgan_noise_val']
    pix2pix_batch_size=kwargs['pix2pix_batch_size']
    pix2pix_n_epochs=kwargs['pix2pix_n_epochs']


    # change working directory to output_path
    os.chdir(output_path)
    print("Output path changed to {}".format(output_path))

    # set data set path
    imgs_dataset = os.path.join(img_path, img_file)

    print(imgs_dataset)

    ## DCGAN

    while incl_gans[0]:

        print("Training DCGAN.")
```

```
        # define discriminator
        dcgan_d_model =
        ↪  dcgan.define_discriminator(incl_noise=dcgan_incl_noise,
        ↪  noise_val=dcgan_noise_val)
        # save a summary of the model to disk
        dcgan.summary2file(dcgan_d_model,
        ↪  'dcgan_discriminator_modelsummary.txt')

        # define generator
        dcgan_g_model = dcgan.define_generator(dcgan_latent_dim)
        # save a summary of the model to disk
        dcgan.summary2file(dcgan_g_model,
        ↪  'dcgan_generator_modelsummary.txt')

        # create the gan
        dcgan_gan_model = dcgan.define_gan(dcgan_g_model,
                                           dcgan_d_model)
        # save a summary of the model to disk
        dcgan.summary2file(dcgan_gan_model,
        ↪  'dcgan_gan_modelsummary.txt')

        # load image data
        dataset = dcgan.load_real_samples(imgs_dataset)

        # train model
        dcgan.train(dcgan_g_model,
                    dcgan_d_model,
                    dcgan_gan_model,
                    dataset,
                    dcgan_latent_dim,
                    dcgan_n_epochs,
                    dcgan_batch_size)

        # Exit after training completes
        incl_gans[0] = False

    ## pix2pix
    while incl_gans[1]:

        print('Training pix2pix')

        ## Hyper params
        # width, height of images to work with. Assumes images are
        ↪  square
        img_width = img_height = 256
```

```
            # input/output channels in image
            input_channels = 3
            output_channels = 3

            # image dims
            input_img_dim = (img_width, img_height, input_channels)
            output_img_dim = (img_width, img_height, output_channels)

            # define generator
            pix2pix_g_model = pix2pix.define_generator(input_img_dim,
            ↪  output_channels)
            # save a summary of the model to disk
            pix2pix.summary2file(pix2pix_g_model,
            ↪  'pix2pix_generator_modelsummary.txt')

            # define discriminator
            pix2pix_d_model = pix2pix.define_discriminator(output_img_dim)
            # save a summary of the model to disk
            pix2pix.summary2file(pix2pix_g_model,
            ↪  'pix2pix_discriminator_modelsummary.txt')

            # define the composite model
            pix2pix_gan_model = pix2pix.define_gan(pix2pix_g_model,
            ↪  pix2pix_d_model, input_img_dim)
            # save a summary of the model to disk
            pix2pix.summary2file(pix2pix_g_model,
            ↪  'pix2pix_GAN_modelsummary.txt')

            # train model
            pix2pix.train(pix2pix_d_model,
                          pix2pix_g_model,
                          pix2pix_gan_model,
                          imgs_dataset,
                          pix2pix_n_epochs,
                          pix2pix_batch_size)

            # Exit after training completes
            incl_gans[1] = False

if __name__ == "__main__":

    # Run configuration examples:
    #
```

```
# --mode train --images ..\gan-heightmaps\ --imagefile
↪  data_256x256_train_val_float_scaled_neg1_to_1.hd5 --output
↪  GAN_output --dcgan --dcgan_n_epochs 100 --dcgan_latent_dim
↪  250
#
#--mode train --images ..\gan-heightmaps\ --imagefile
↪  data_256x256_train_val_float_scaled_neg1_to_1.hd5 --output
↪  GAN_output --dcgan --dcgan_n_epochs 100 --dcgan_batch_size
↪  20 --dcgan_latent_dim 250
#
# --mode train


# instantiate CLI parser
parser = argparse.ArgumentParser(description='Stacked GAN for
↪  terrain generation.')
parser.add_argument("--mode",
                    choices=["train", "other"],
                    required=True,
                    type=str,
                    help="Operational mode")
parser.add_argument("--images",
                    required=True,
                    type=str,
                    help="Path to images")
parser.add_argument("--imagefile",
                    required=True,
                    type=str,
                    help="Name of .H5 image file")
parser.add_argument("--output",
                    required=True,
                    type=str,
                    help="Path to output folder (must exist)")
parser.add_argument("--dcgan",
                    action='store_true',
                    default=False,
                    help="inlcude dcgan")
parser.add_argument("--dcgan_n_iter",
                    type=int,
                    default=100,
                    help="dcgan number of iterations")
parser.add_argument("--dcgan_batch_size",
                    type=int,
                    default=40,
                    help="dcgan mini-batch size")
```

```
parser.add_argument("--dcgan_n_epochs",
                    type=int,
                    default=100,
                    help="dcgan number of epochs")
parser.add_argument("--dcgan_latent_dim",
                    type=int,
                    default=250,
                    help="DCGAN size of latent dim")
parser.add_argument("--dcgan_incl_noise",
                    action='store_false',
                    default=False,
                    help="DCGAN include noise input layer")
parser.add_argument("--dcgan_noise_val",
                    type=float,
                    default=0.2,
                    help="DCGAN the amount of noise used if noise
                    ↪  layer is true")
parser.add_argument("--pix2pix",
                    action='store_true',
                    default=False,
                    help="Include pix2pix GAN")
parser.add_argument("--pix2pix_batch_size",
                    type=int,
                    default=40,
                    help="pix2pix mini-batch size")
parser.add_argument("--pix2pix_n_epochs",
                    type=int,
                    default=100,
                    help="pix2pix number of epochs")

# parse CLI options
args = parser.parse_args()
mode = args.mode
img_path = args.images
img_file = args.imagefile
output_path = args.output
incl_dcgan = args.dcgan
incl_pix2pix = args.pix2pix
dcgan_n_iter = args.dcgan_n_iter
dcgan_batch_size = args.dcgan_batch_size
dcgan_n_epochs = args.dcgan_n_epochs
dcgan_latent_dim = args.dcgan_latent_dim
dcgan_incl_noise = args.dcgan_incl_noise,
dcgan_noise_val = args.dcgan_noise_val,
pix2pix_batch_size = args.pix2pix_batch_size
```

```
pix2pix_n_epochs = args.pix2pix_n_epochs

# configure mode
if mode == 'train':

    # Check if given image path exists
    if os.path.isdir(img_path) is False:
        print('Unable to locate: {}'.format(img_path))
        sys.exit(0)

    if os.path.isfile(img_path + img_file) is False:
        print('Error with image file: {}'.format(img_file))
        sys.exit(0)

    # Check if given output path exists
    if os.path.isdir(output_path) is False:
        print('Unable to locate: {}'.format(output_path))
        sys.exit(0)


    # Check if at least GAN is set to true
    if incl_dcgan is False and incl_pix2pix is False:
        print("At least one GAN must be included for training.")
        sys.exit()

    # Set GANs to include for training
    incl_gans = [incl_dcgan, incl_pix2pix]
    print(incl_gans)


    # run driver
    main(mode=mode,
         img_path=img_path,
         img_file=img_file,
         output_path=output_path,
         incl_gans=incl_gans,
         dcgan_n_iter=dcgan_n_iter,
         dcgan_n_epochs=dcgan_n_epochs,
         dcgan_batch_size=dcgan_batch_size,
         dcgan_latent_dim=dcgan_latent_dim,
         dcgan_incl_noise=dcgan_incl_noise,
         dcgan_noise_val=dcgan_noise_val,
         pix2pix_batch_size=pix2pix_batch_size,
         pix2pix_n_epochs=pix2pix_n_epochs)
```

dcgan.py

```
import h5py

from numpy import ones
from numpy import zeros
from numpy import vstack
from numpy.random import rand
from numpy.random import choice
from numpy.random import randn
from numpy.random import normal
from numpy.random import uniform

from contextlib import redirect_stdout

from keras.models import Sequential
from keras.optimizers import Adam
from keras.layers import GaussianNoise
from keras.layers import BatchNormalization
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import Activation
from keras.layers import LeakyReLU
from keras.layers import Dropout
from keras.layers import AveragePooling2D
from keras.layers import UpSampling2D

import matplotlib.pyplot as plt


## GAN components: discriminator, generator, composite GAN

# define the standlone discrimintor model
def define_discriminator(in_shape=(256, 256, 1), incl_noise=False,
↪  noise_val=0.2):
    model = Sequential()

    if incl_noise:
        # add Gaussian noise to prevent Discriminator overfitting
        model.add(GaussianNoise(noise_val, input_shape=in_shape))
```

```
    # 256x256x1 Image
    model.add(Conv2D(filters=8, kernel_size=3, padding='same'))
    model.add(LeakyReLU(0.2))
    model.add(Dropout(0.25))
    model.add(AveragePooling2D())

else:
    # standard input layer
    # 256x256x1 Image
    model.add(Conv2D(filters=8, kernel_size=3, padding='same',
    ↪  input_shape=in_shape))
    model.add(LeakyReLU(0.2))
    model.add(Dropout(0.25))
    model.add(AveragePooling2D())

# 128x128x8
model.add(Conv2D(filters=16, kernel_size=3, padding='same'))
model.add(BatchNormalization(momentum=0.7))
model.add(LeakyReLU(0.2))
model.add(Dropout(0.25))
model.add(AveragePooling2D())

# 64x64x16
model.add(Conv2D(filters=32, kernel_size=3, padding='same'))
model.add(BatchNormalization(momentum=0.7))
model.add(LeakyReLU(0.2))
model.add(Dropout(0.25))
model.add(AveragePooling2D())

# 32x32x32
model.add(Conv2D(filters=64, kernel_size=3, padding='same'))
model.add(BatchNormalization(momentum=0.7))
model.add(LeakyReLU(0.2))
model.add(Dropout(0.25))
model.add(AveragePooling2D())

# 16x16x64
model.add(Conv2D(filters=128, kernel_size=3, padding='same'))
model.add(BatchNormalization(momentum=0.7))
model.add(LeakyReLU(0.2))
model.add(Dropout(0.25))
model.add(AveragePooling2D())

# 8x8x128
model.add(Conv2D(filters=256, kernel_size=3, padding='same'))
```

```
    model.add(BatchNormalization(momentum=0.7))
    model.add(LeakyReLU(0.2))
    model.add(Dropout(0.25))
    model.add(AveragePooling2D())

    # 4x4x256
    model.add(Flatten())

    # 256
    model.add(Dense(128))
    model.add(LeakyReLU(0.2))

    model.add(Dense(1, activation='sigmoid'))

    # compile model
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt,
    ↪  metrics=['accuracy'])

    return model


# define the standalone generator model
def define_generator(latent_dim):
    model = Sequential()

    model.add(Reshape(target_shape=[1, 1, latent_dim],
    ↪  input_shape=[latent_dim]))

    # 1x1x250
    model.add(Conv2DTranspose(filters=256, kernel_size=4))
    model.add(LeakyReLU(alpha=0.2))

    # 4x4x256
    model.add(Conv2D(filters=256, kernel_size=4, padding='same'))
    model.add(BatchNormalization(momentum=0.7))
    model.add(LeakyReLU(alpha=0.2))
    model.add(UpSampling2D())

    # 8x8x256
    model.add(Conv2D(filters=128, kernel_size=4, padding='same'))
    model.add(BatchNormalization(momentum=0.7))
    model.add(LeakyReLU(alpha=0.2))
    model.add(UpSampling2D())
```

```
    # 16x16x128
    model.add(Conv2D(filters=64, kernel_size=3, padding='same'))
    model.add(BatchNormalization(momentum=0.7))
    model.add(LeakyReLU(alpha=0.2))
    model.add(UpSampling2D())

    # 32x32x64
    model.add(Conv2D(filters=32, kernel_size=3, padding='same'))
    model.add(BatchNormalization(momentum=0.7))
    model.add(LeakyReLU(alpha=0.2))
    model.add(UpSampling2D())

    # 64x64x32
    model.add(Conv2D(filters=16, kernel_size=3, padding='same'))
    model.add(BatchNormalization(momentum=0.7))
    model.add(LeakyReLU(alpha=0.2))
    model.add(UpSampling2D())

    # 128x128x16
    model.add(Conv2D(filters=8, kernel_size=3, padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(UpSampling2D())

    # 256x256x8
    model.add(Conv2D(filters=1, kernel_size=3, padding='same'))
    model.add(Activation('tanh'))

    return model


# define the combined generator and discriminator model, for
 ↪  updating the generator
def define_gan(g_model, d_model):
    # make weights in the discriminator not trainable
    d_model.trainable = False

    # connect them
    model = Sequential()

    # add generator
    model.add(g_model)

    # add the discriminator
    model.add(d_model)
```

```
    # compile model
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt)

    return model

## Helper Functions
# load and prepare training images
def load_real_samples(filename):
    # load image data, Read HDF5 file
    f = h5py.File(filename, 'r')
    dataset = f['xt']
    print('Dataset input shape {}'.format(str(dataset.shape)))

    return dataset


# select real samples
def generate_real_samples(dataset, n_samples):
    # create index numbers
    ix = range(dataset.shape[0])

    # choose n_samples of integers at random without replacement
    # sort the result
    ix = sorted(choice(ix, size=n_samples, replace=False))

    # save index values for use with other GANs
    save_idx(ix)

    # retrieve selected images using fancy indexing
    X = dataset[ix]

    # generate 'real' class labels (1)
    y = ones((n_samples, 1))

    return X, y

def save_idx(ix_vals):

    # create row
    line = ','.join([str(i) for i in ix_vals])
    line = line + '\n'

    # write row to file
    with open('img_index.csv', 'a') as f:
```

```
        f.write(line)


# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
    # generate points in the latent space
    #x_input = randn(latent_dim * n_samples)

    #x_input = uniform(-1.0, 1.0, size=[n_samples, latent_dim])

    x_input = normal(0, 1, size=(n_samples, latent_dim))


    #print(x_input)

    # reshape into a batch of inputs for the network
    x_input = x_input.reshape(n_samples, latent_dim)

    return x_input


# use the generator to generate n fake examples, with class labels
def generate_fake_samples(g_model, latent_dim, n_samples):
    # generate points in latent space
    x_input = generate_latent_points(latent_dim, n_samples)

    # predict outputs
    X = g_model.predict(x_input)

    # create 'fake' class labels (1)
    y = zeros((n_samples, 1))

    return X, y


## Training and performance functions

# save a summary of the model to disk
def summary2file(model, filename):
    with open(filename, 'w') as f:
        with redirect_stdout(f):
            model.summary()
```

```
# create and save a plot of generated images (reversed grayscale)
def save_plot(examples, epoch, n=5):


    # plot image
    for i in range(n * n):
        # define subplot
        plt.subplot(n, n, 1 + i)

        # turn off axis
        plt.axis('off')

        # plot raw pixel data
        plt.imshow(examples[i, :, :, 0], cmap='gray_r')


    # save plot to file
    filename = 'dcgan_generated_plot_e%03d.png' % (epoch + 1)
    plt.savefig(filename)
    plt.close()


'''
# create a line plot of loss for the gan and save to file
def plot_history(d1_hist, d2_hist, g_hist, a1_hist, a2_hist):
    # plot loss
    plt.subplot(2, 1, 1)
    plt.plot(d1_hist, label='d-real')
    plt.plot(d2_hist, label='d-fake')
    plt.plot(g_hist, label='gen')
    plt.legend()

    # plot discriminator accuracy
    plt.subplot(2, 1, 2)
    plt.plot(a1_hist, label='acc-real')
    plt.plot(a2_hist, label='acc-fake')
    plt.legend()

        # save plot to file
    filename = 'plot_line_plot_loss.png'
    plt.savefig(filename)
    plt.close()
'''

def plot_history(d1_hist, d2_hist, g_hist, a1_hist, a2_hist):
```

```
    # plot loss
    plt.subplot(1, 3, 1)
    plt.plot(d1_hist, label='d-real')
    plt.xlabel('Batch Number')
    plt.ylabel('Loss')
    plt.title('D-Loss: Real')

    plt.subplot(1, 3, 2)
    plt.plot(d2_hist, label='d-fake')
    plt.xlabel('Batch Number')
    plt.title('D-Loss: Fake')

    plt.subplot(1, 3, 3)
    plt.plot(g_hist, label='gen')
    plt.xlabel('Batch Number')
    plt.title('G-Loss')

    # plot layout
    plt.tight_layout()

    # save plot to file
    filename = 'dcgan_plot_line_plot_loss.png'
    plt.savefig(filename)
    plt.close()


# evaluate the discriminator, plot generted images, save genertor
 ↪  model
def summarize_performance(epoch, g_model, d_model, dataset,
 ↪  latent_dim, n_samples=25):
    # prepare real samples
    X_real, y_real = generate_real_samples(dataset, n_samples)

    # evaluate discriminator on real examples
    _, acc_real = d_model.evaluate(X_real, y_real, verbose=0)

    # prepare fake examples
    x_fake, y_fake = generate_fake_samples(g_model, latent_dim,
     ↪  n_samples)

    # evaluate discriminator on fake examples
    _, acc_fake = d_model.evaluate(x_fake, y_fake, verbose=0)

    # summarize discriminator performance
```

```
    print('>Discriminator Accuracy real: %.0f%%, fake: %.0f%%' %
    ↪  (acc_real * 100, acc_fake * 100))

    # save plot
    save_plot(x_fake, epoch)

    # save the generator model tile file
    filename = 'dcgan_generator_model_%03d.h5' % (epoch + 1)
    g_model.save(filename)


'''
# train the composite model
def train_gan(gan_model, latent_dim, n_epochs=100, n_batches=256):
    # manually enumerate epochs
    for i in range(n_epochs):
        # prepare points in latent space as input for the generator
        x_gan = generate_latent_point(latent_dim, n_batch)

        # create inverted labels for the fake samples
        y_gan = one((n_batch, 1))

        # update the generator via the discriminator's error
        gan_model.train_on_batch(x_gan, y_gan)
'''


# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim,
↪  n_epochs=100, n_batch=256):

    # prepare lists for storing stats each iteration
    d1_hist, d2_hist, g_hist, a1_hist, a2_hist = list(), list(),
    ↪  list(), list(), list()

    bat_per_epo = int(dataset.shape[0] / n_batch)
    half_batch = int(n_batch / 2)

    # manually enumerate epochs
    for i in range(n_epochs):

        # enumerate batches over the training set
        for j in range(bat_per_epo):

            # get randomly selected 'real' samples
```

```
        X_real, y_real = generate_real_samples(dataset,
        ↪  half_batch)

        # update discriminator model weights
        d_loss1, d_acc1 = d_model.train_on_batch(X_real, y_real)

        # generate 'fake' examples
        X_fake, y_fake = generate_fake_samples(g_model,
        ↪  latent_dim, half_batch)

        # update discriminator model weights
        d_loss2, d_acc2 = d_model.train_on_batch(X_fake, y_fake)

        # prepare points in latent space as input for the
        ↪  generator
        X_gan = generate_latent_points(latent_dim, n_batch)

        # create inverted labels for the fake samples
        y_gan = ones((n_batch, 1))

        # update the generator via the discriminator's error
        g_loss = gan_model.train_on_batch(X_gan, y_gan)

        # summarize loss on this batch
        #print('>%d, %d/%d, d=%.3f, g=%.3f' % (i + 1, j + 1,
        ↪  bat_per_epo, d_loss, g_loss))

        # summarize loss on this batch
        print('>%d, %d/%d, d1[%.3f], d2[%.3f], g[%.3f], a1[%d],
        ↪  a2[%d]' %
              (i + 1, j+1, bat_per_epo, d_loss1,  d_loss2, g_loss,
              ↪  int(100 * d_acc1), int(100 * d_acc2)))

        # record history
        d1_hist.append(d_loss1)
        d2_hist.append(d_loss2)
        g_hist.append(g_loss)
        a1_hist.append(d_acc1)
        a2_hist.append(d_acc2)

    # evaluate the model performance, every 10 epochs
    if (i + 1) % 10 == 0:
        summarize_performance(i, g_model, d_model, dataset,
        ↪  latent_dim)
```

```
    # create history plot at the conclusion of training
    plot_history(d1_hist, d2_hist, g_hist, a1_hist, a2_hist)
```

pix2pix.py

```
import os

import h5py

from numpy import asarray
from numpy import zeros
from numpy import ones
from numpy.random import choice
#from numpy.random import randint

from cv2 import merge

import matplotlib.pyplot as plt

from contextlib import redirect_stdout

from keras.optimizers import Adam
from keras.initializers import RandomNormal
from keras.models import Model
from keras.models import Input
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import Activation
from keras.layers import Concatenate
from keras.layers import Dropout
from keras.layers import BatchNormalization
from keras.layers import LeakyReLU

# define the discriminator model
def define_discriminator(image_shape):
    # weight initialization
    init = RandomNormal(stddev=0.02)
    # source image input
    in_src_image = Input(shape=image_shape)
    # target image input
    in_target_image = Input(shape=image_shape)
    # concatenate images channel-wise
    merged = Concatenate()([in_src_image, in_target_image])
```

```
    # C64
    d = Conv2D(64, (4, 4), strides=(2, 2), padding='same',
    ↪  kernel_initializer=init)(merged)
    d = LeakyReLU(alpha=0.2)(d)
    # C128
    d = Conv2D(128, (4, 4), strides=(2, 2), padding='same',
    ↪  kernel_initializer=init)(d)
    d = BatchNormalization()(d)
    d = LeakyReLU(alpha=0.2)(d)
    # C256
    d = Conv2D(256, (4, 4), strides=(2, 2), padding='same',
    ↪  kernel_initializer=init)(d)
    d = BatchNormalization()(d)
    d = LeakyReLU(alpha=0.2)(d)
    # C512
    d = Conv2D(512, (4, 4), strides=(2, 2), padding='same',
    ↪  kernel_initializer=init)(d)
    d = BatchNormalization()(d)
    d = LeakyReLU(alpha=0.2)(d)
    # second last output layer
    d = Conv2D(512, (4, 4), padding='same',
    ↪  kernel_initializer=init)(d)
    d = BatchNormalization()(d)
    d = LeakyReLU(alpha=0.2)(d)
    # patch output
    d = Conv2D(1, (4, 4), padding='same', kernel_initializer=init)(d)
    patch_out = Activation('sigmoid')(d)
    # define model
    model = Model([in_src_image, in_target_image], patch_out)
    # compile model
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt,
    ↪  loss_weights=[0.5])
    return model


# define an encoder block
def define_encoder_block(layer_in, n_filters, batchnorm=True):
    # weight initialization
    init = RandomNormal(stddev=0.02)
    # add downsampling layer
    g = Conv2D(n_filters, (4, 4), strides=(2, 2), padding='same',
    ↪  kernel_initializer=init)(layer_in)
    # conditionally add batch normalization
    if batchnorm:
```

```
        g = BatchNormalization()(g, training=True)
    # leaky relu activation
    g = LeakyReLU(alpha=0.2)(g)
    return g


# define a decoder block
def decoder_block(layer_in, skip_in, n_filters, dropout=True):
    # weight initialization
    init = RandomNormal(stddev=0.02)
    # add upsampling layer
    g = Conv2DTranspose(n_filters, (4, 4), strides=(2, 2),
    ↪  padding='same', kernel_initializer=init)(layer_in)
    # add batch normalization
    g = BatchNormalization()(g, training=True)
    # conditionally add dropout
    if dropout:
        g = Dropout(0.5)(g, training=True)
    # merge with skip connection
    g = Concatenate()([g, skip_in])
    # relu activation
    g = Activation('relu')(g)
    return g


# define the standalone generator model
def define_generator(image_shape=(256, 256, 1), output_channels=3):
    # weight initialization
    init = RandomNormal(stddev=0.02)
    # image input
    in_image = Input(shape=image_shape)
    # encoder model
    e1 = define_encoder_block(in_image, 64, batchnorm=False)
    e2 = define_encoder_block(e1, 128)
    e3 = define_encoder_block(e2, 256)
    e4 = define_encoder_block(e3, 512)
    e5 = define_encoder_block(e4, 512)
    e6 = define_encoder_block(e5, 512)
    e7 = define_encoder_block(e6, 512)

    # bottleneck, no batch norm and relu
    b = Conv2D(512, (4, 4), strides=(2, 2), padding='same',
    ↪  kernel_initializer=init)(e7)
    b = Activation('relu')(b)
```

```
    # decoder model
    d1 = decoder_block(b, e7, 512)
    d2 = decoder_block(d1, e6, 512)
    d3 = decoder_block(d2, e5, 512)
    d4 = decoder_block(d3, e4, 512, dropout=False)
    d5 = decoder_block(d4, e3, 256, dropout=False)
    d6 = decoder_block(d5, e2, 128, dropout=False)
    d7 = decoder_block(d6, e1, 64, dropout=False)

    # Output
    # After the last layer in the decoder, a Transposed convolution
    # layer (sometimes called Deconvolution) is applied
    # to map to the number of output channels (3 in general,
    # except in colorization, where it is 2), followed by a Tanh
    # function.
    g = Conv2DTranspose(output_channels, (4, 4), strides=(2, 2),
    ↪  padding='same', kernel_initializer=init)(d7)
    out_image = Activation('tanh')(g)
    # define model
    model = Model(in_image, out_image)
    return model



# define the combined generator and discriminator model, for
↪  updating the generator
def define_gan(g_model, d_model, image_shape):
    # make weights in the discriminator not trainable
    d_model.trainable = False
    # define the source image
    in_src = Input(shape=image_shape)
    # connect the source image to the generator input
    gen_out = g_model(in_src)

    # connect the source input and generator output to the
    ↪  discriminator input
    dis_out = d_model([in_src, gen_out])

    # src image as input, generated image and classification output
    model = Model(in_src, [dis_out, gen_out])

    # compile model
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss=['binary_crossentropy', 'mae'], optimizer=opt,
    ↪  loss_weights=[1, 100])
```

```
    return model


# load and prepare training images
def load_real_samples(filename):

    # load image data, Read HDF5 file
    f = h5py.File(filename, 'r')

    # Grab the first 216 random images
    X1 = f['xt']
    X2 = f['yt']
    print('Dataset input shapes {}, {}'.format(str(X1.shape),
    ↪  str(X2.shape)))

    '''
    # This is done in the datafile.
    # scale from [0,1] to [-1,1]
    print('Scaling X1 data to [-1,1]')
    X1 = (X1[:] - .5) / .5
    print('Scaling X2 data to [-1,1]')
    X2 = (X2[:] - .5) / .5
    '''

    return [X1, X2]


# covert grayscle 1-channel to RGB 3-channel
def gray2rgb(gray_imgs):
    ''' Convert on 1-channel image to 3-channel image
        using CV

    :param gray_imgs:
    :return: three_channel_imgs
    '''
    three_channel_imgs = []

    idx, height, length, channel = gray_imgs.shape

    for i in range(idx):

        # 1-channel image
        img = gray_imgs[i]

        # convert to 3-channel image
```

```
            img2 = merge((img,img,img))

            # add image to collection
            three_channel_imgs.append(img2)

        three_channel_imgs = asarray(three_channel_imgs)

        return three_channel_imgs


# select a batch of random samples, returns images and target
def generate_real_samples(datasetA, datasetB, n_samples, patch_shape):

    # choose random instances
    ix = range(datasetA.shape[0])

    # choose n_samples of integers at random without replacement
    # sort the result
    ix = sorted(choice(ix, size=n_samples, replace=False))

    # ix = randint(0, datasetA.shape[0], n_samples)


    # retrieve selected images
    X1, X2 = datasetA[ix], datasetB[ix]

    # convert grayscale 1-channel to RGB 3-channel
    X1 = gray2rgb(X1)

    # generate 'real' class labels (1)
    y = ones((n_samples, patch_shape, patch_shape, 1))
    return [X1, X2], y


# generate a batch of images, returns images and targets
def generate_fake_samples(g_model, samples, patch_shape):
    # generate fake instance
    X = g_model.predict(samples)
    # create 'fake' class labels (0)
    y = zeros((len(X), patch_shape, patch_shape, 1))
    return X, y

## Training and performance functions

# save a summary of the model to disk
```

```
def summary2file(model, filename):
    with open(filename, 'w') as f:
        with redirect_stdout(f):
            model.summary()

# create and save a plot of generated images (reversed grayscale)
def save_plot(examples, epoch, n=5):


    # plot image
    for i in range(n * n):
        # define subplot
        plt.subplot(n, n, 1 + i)

        # turn off axis
        plt.axis('off')

        # plot raw pixel data
        plt.imshow(examples[i, :, :, 0], cmap='gray_r')


    # save plot to file
    filename = 'pix2pix_generated_plot_e%03d.png' % (epoch + 1)
    plt.savefig(filename)
    plt.close()


def plot_history(d1_hist, d2_hist, g_hist):
    # plot loss
    plt.subplot(1, 3, 1)
    plt.plot(d1_hist, label='d-real')
    plt.xlabel('Batch Number')
    plt.ylabel('Loss')
    plt.title('D-Loss: Real')

    plt.subplot(1, 3, 2)
    plt.plot(d2_hist, label='d-fake')
    plt.xlabel('Batch Number')
    plt.title('D-Loss: Fake')

    plt.subplot(1, 3, 3)
    plt.plot(g_hist, label='gen')
    plt.xlabel('Batch Number')
    plt.title('G-Loss')
```

```
    # plot layout
    plt.tight_layout()

    # save plot to file
    filename = 'pix2pix_plot_line_plot_loss.png'
    plt.savefig(filename)
    plt.close()


# generate samples and save as a plot and save the model
def summarize_performance(epoch, g_model, datasetA, datasetB,
↪   n_samples=5):
    # select a sample of input images
    [X_realA, X_realB], _ = generate_real_samples(datasetA, datasetB,
    ↪   n_samples, 1)
    # generate a batch of fake samples
    X_fakeB, _ = generate_fake_samples(g_model, X_realA, 1)
    # scale all pixels from [-1,1] to [0,1]
    X_realA = (X_realA + 1) / 2.0
    X_realB = (X_realB + 1) / 2.0
    X_fakeB = (X_fakeB + 1) / 2.0
    # plot real source images
    for i in range(n_samples):
        plt.subplot(3, n_samples, 1 + i)
        plt.axis('off')
        plt.imshow(X_realA[i])
    # plot generated target image
    for i in range(n_samples):
        plt.subplot(3, n_samples, 1 + n_samples + i)
        plt.axis('off')
        plt.imshow(X_fakeB[i])
    # plot real target image
    for i in range(n_samples):
        plt.subplot(3, n_samples, 1 + n_samples * 2 + i)
        plt.axis('off')
        plt.imshow(X_realB[i])
    # save plot to file
    filename1 = 'pix2pix_plot_%06d.png' % (epoch + 1)
    plt.savefig(filename1)
    plt.close()

    # save the generator model
    filename2 = 'pix2pix_model_%06d.h5' % (epoch + 1)
    g_model.save(filename2)
    print('>Saved: %s and %s' % (filename1, filename2))
```

```
# train pix2pix models
def train(d_model, g_model, gan_model, dataset, n_epochs=100,
↪  n_batch=256):

    # prepare lists for storing stats each iteration
    d1_hist, d2_hist, g_hist = list(), list(), list()

    # determine the output square shape of the discriminator
    n_patch = d_model.output_shape[1]
    # unpack dataset
    #trainA, trainB = dataset

    trainA, trainB = load_real_samples(dataset)

    # calculate the number of batches per training epoch
    bat_per_epo = int(trainA.shape[0] / n_batch)

    # calculate the number of training iterations
    #n_steps = bat_per_epo * n_epochs

    # manually enumerate epochs
    for i in range(n_epochs):

        # enumerate batches over the training set
        for j in range(bat_per_epo):

            # select a batch of real samples
            #print("generate real samples")
            [X_realA, X_realB], y_real = generate_real_samples(trainA,
            ↪  trainB, n_batch, n_patch)

            # generate a batch of fake samples
            #print("generate fake samples")
            X_fakeB, y_fake = generate_fake_samples(g_model, X_realA,
            ↪  n_patch)

            # update discriminator for real samples
            #print("train on batch")
            d_loss1 = d_model.train_on_batch([X_realA, X_realB],
            ↪  y_real)

            # update discriminator for generated samples
```

```
            d_loss2 = d_model.train_on_batch([X_realA, X_fakeB],
            ↪  y_fake)

            # update the generator
            g_loss, _, _ = gan_model.train_on_batch(X_realA, [y_real,
            ↪  X_realB])

            # summarize performance
            print('>%d, %d/%d, d1[%.3f] d2[%.3f] g[%.3f]' %
                    (i + 1, j+1, bat_per_epo, d_loss1, d_loss2,
                    ↪  g_loss))

            # record history
            d1_hist.append(d_loss1)
            d2_hist.append(d_loss2)
            g_hist.append(g_loss)


        # evaluate the model performance, every 10 epochs
        if (i + 1) % 10 == 0:
            summarize_performance(i, g_model, trainA, trainB)

    # create history plot at the conclusion of training
    plot_history(d1_hist, d2_hist, g_hist)
```

B.3 Source Files

The following are Jupyter notebooks exported as a Python file.

Create new maps from generator.ipynb

```
#!/usr/bin/env python
# coding: utf-8

# # Create new maps from generator

# ## Create Heightmaps

# ## Create 100 random images and save latent values

# In[13]:
```

```
# example of loading the generator model and generating images
from numpy.random import randn
from numpy.random import normal
from numpy import load
from numpy import mean
from numpy import vstack
from numpy import expand_dims
from keras.models import load_model
import matplotlib.pyplot as plt
from numpy import savez_compressed

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
    # generate points in the latent space
    #x_input = randn(latent_dim * n_samples)
    x_input = normal(0, 1, size=(n_samples, latent_dim))

    # reshape into a batch of inputs for the network
    z_input = x_input.reshape(n_samples, latent_dim)

    return z_input

# create a plot of generated images
def plot_generated(examples, n):
    # plot images
    plt.figure(figsize=(15,15))

    for i in range(n * n):
        # define subplot
        plt.subplot(n, n, 1 + i)
        # turn off axis
        plt.axis('off')
        # plot raw pixel data
        plt.imshow(examples[i, :, :,0], cmap="gray")

    plt.savefig('..\\images\\generated_heightmaps.png')
    plt.close()

# load model
model =
↪  load_model('..\\models\dcgan\w_gaussian_layer\\generator_model_075.h5')
# generate points in latent space
latent_points = generate_latent_points(250, 100)
```

```
# save points
savez_compressed('..\\misc\\latent_points.npz', latent_points)
# generate images
X = model.predict(latent_points)
# scale from [-1,1] to [0,1]
#X = (X + 1) / 2.0
# save plot
plot_generated(X, 10)


# ## Read in file with latent values

# In[2]:


# example of reading in data file with latent points
from numpy import load
from keras.models import load_model
import matplotlib.pyplot as plt

# create a plot of generated images
def plot_generated(examples, n):
    # plot images
    plt.figure(figsize=(15,15))

    for i in range(n * n):
        # define subplot
        plt.subplot(n, n, 1 + i)
        # turn off axis
        plt.axis('off')
        # plot raw pixel data
        plt.imshow(examples[i, :, :,0], cmap="gray")

# load saved model
model =
↪  load_model('..\\models\dcgan\w_gaussian_layer\\generator_model_075.h5')

# load the saved latent points
data = load('..\\misc\\latent_points.npz')
latent_points = data['arr_0']

# generate images
X = model.predict(latent_points)
plot_generated(X, 10)
```

```
# ## Retrieve image and export as tiff file

# In[3]:


# example of plotting an image from a latent vector
from keras.models import load_model
import matplotlib.pyplot as plt
from PIL import Image

# create a plot of generated images
def plot_generated(example, image_ix):

    # retrieve raw pixel data
    arr = example[0, :, :,0]

    # Display image
    plt.axis("off")
    plt.imshow(arr, cmap="gray")

    # Save as TIFF
    img = Image.fromarray(arr)
    img.save('..\\images\\generated_heightmap' + str(image_ix) +
    ↪  '.tif')

# substract 1 from image matrix location
#image_ix = 42
image_ix = 34
#image_ix = 14
specific_points = latent_points[image_ix]
specific_points = specific_points.reshape(1, specific_points.shape[0])

# load saved model
model =
↪  load_model('..\\models\dcgan\w_gaussian_layer\\generator_model_075.h5')

# Send latent point to generator and plot image
X = model.predict(specific_points)
plot_generated(X, image_ix)


# ## Translate Images - Create Texture maps and save as .bmp file
# Run the 'Read in file with latent values' section for heightmaps
↪  sources
```

```
# In[4]:


from keras.models import load_model
from cv2 import merge
from scipy.misc import imsave
import matplotlib.pyplot as plt

# covert grayscle 1-channel to RGB 3-channel
def gray2rgb(gray_img):

    # convert to 3-channel image
    img = merge((gray_img,gray_img,gray_img))

    return img

# create a plot of generated images
def plot_generated(example, image_ix):

    # retrieve raw pixel data
    arr = example[0, :, :, :]

    # Display image
    plt.axis("off")
    plt.imshow(arr)

    # Save as BMP
    #imsave('..\\Textures\\generated_texture' + str(image_ix) +
    ↪  '.bmp', arr)

# load model
model = load_model('..\\models\pix2pix\\model_000030.h5')

n = X.shape[1]

# Use X array as input and convert to 3-channel
X_3ch = gray2rgb(X[0,:,:,:]).reshape(1, n, n, 3)

# generate image from source
gen_image = model.predict(X_3ch)

# plot all three images
plot_generated(gen_image, image_ix)
```

```
# ## Compare Translate Images - Texture maps
# Send a random heightmap, texture pair and copare the result

# In[11]:


from numpy.random import randint
import h5py
from keras.models import load_model
from cv2 import merge
import matplotlib.pyplot as plt

# load and prepare training images
def load_real_samples(filename):
    # load image data, Read HDF5 file
    f = h5py.File(filename, 'r')
    X1 = f['xt']
    X2 = f['yt']
    print('Dataset input shapes {}, {}'.format(str(X1.shape),
    ↪  str(X2.shape)))

    return [X1, X2]

# covert grayscle 1-channel to RGB 3-channel
def gray2rgb(gray_img):
    ''' Convert on 1-channel image to 3-channel image
        using CV

    '''

    # convert to 3-channel image
    img = merge((gray_img,gray_img,gray_img))

    return img

# plot source, generated and target images
def plot_images(src_img_1ch, gen_img, tar_img):

    # scale from [-1, 1] to [0, 1]
    images = (images + 1) / 2.0

    # plot images row by row
    # define subplot
    plt.subplot(1, 3, 1)
```

```
        # turn off axis
        plt.axis("off")
        # plot raw pixel data
        plt.imshow(src_img_1ch[:,:,0], cmap="gray")
        # show title
        plt.title('Source')

        # define subplot
        plt.subplot(1, 3, 2)
        # turn off axis
        plt.axis("off")
        # plot raw pixel data
        plt.imshow(gen_img[0,:,:,:])
        # show title
        plt.title('Generated')

        # define subplot
        plt.subplot(1, 3, 3)
        # turn off axis
        plt.axis("off")
        # plot raw pixel data
        plt.imshow(tar_img[0,:,:,:])
        # show title
        plt.title('Expected')

        # plot layout
        plt.tight_layout()

        plt.savefig('...\\images\\translated_img_5.png')
        plt.show()
        plt.close()

# load dataset
dataset = 'data_256x256_train_val_float_scaled_neg1_to_1.hd5'
X1, X2 = load_real_samples(dataset)
print('Loaded', X1.shape, X2.shape)

# load model
model = load_model('..\\models\pix2pix\\model_000030.h5')

# select random example, change the 1-channel image to 3-channel
ix = randint(0, len(X1), 1)[0]
src_image_1ch = X1[ix]
tar_image = X2[ix]
```

```
n = tar_image.shape[0]

src_image_3ch = gray2rgb(X1[ix]).reshape(1, n, n, 3)
tar_image = X2[ix].reshape(1,n, n, 3)

# generate image from source
gen_image = model.predict(src_image_3ch)

# plot all three images
plot_images(src_image_1ch, gen_image, tar_image)
```

Exploring the Latent Space.ipynb

```
#!/usr/bin/env python
# coding: utf-8

# # Explore the Latent Space for Generated Heightmaps

# ## Load the Model and Generate Heightmaps

# In[7]:


# example of loading the generator model and generating images
from numpy.random import randn
from numpy.random import normal
from keras.models import load_model
import matplotlib.pyplot as plt

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
    # generate points in the latent space
    #x_input = randn(latent_dim * n_samples)

    x_input = normal(0, 1, size=(n_samples, latent_dim))

    # reshape into a batch of inputs for the network
    z_input = x_input.reshape(n_samples, latent_dim)
    return z_input

# create a plot of generated images
def plot_generated(examples, n):
    # plot images
```

```
    plt.figure(figsize=(15,15))

    for i in range(n * n):
        # define subplot
        plt.subplot(n, n, 1 + i)
        # turn off axis
        plt.axis('off')
        # plot raw pixel data
        plt.imshow(examples[i, :, :,0], cmap="gray")
    plt.show()

# load model
model =
↪  load_model('..\\models\dcgan\w_gaussian_layer\\generator_model_075.h5')
# generate images
latent_points = generate_latent_points(250, 25)
# generate images
X = model.predict(latent_points)
# scale from [-1,1] to [0,1]
#X = (X + 1) / 2.0
# plot the result
plot_generated(X, 5)


# ## Interpolate Between Generated Heightmaps Using Linear
↪  Interpolation

# In[6]:


# example of interpolating between generated heightmaps
from numpy import asarray
from numpy.random import randn
from numpy.random import normal
from numpy import linspace
from keras.models import load_model
import matplotlib.pyplot as plt

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
    # generate points in the latent space
    #x_input = randn(latent_dim * n_samples)
    x_input = normal(0, 1, size=(n_samples, latent_dim))

    # reshape into a batch of inputs for the network
```

```
    z_input = x_input.reshape(n_samples, latent_dim)

    return z_input

# uniform interpolation between two points in latent space
def interpolate_points(p1, p2, n_steps=10):
    # interpolate ratios between the points
    ratios = linspace(0, 1, num=n_steps)

    # linear interpolate vectors
    vectors = list()
    for ratio in ratios:
        v = (1.0 - ratio) * p1 + ratio * p2
        vectors.append(v)

    return asarray(vectors)

# create a plot of generated images
def plot_generated(examples, n):
    # plot images
    plt.figure(figsize=(15,15))

    for i in range(n):
        # define subplot
        plt.subplot(1, n, 1 + i)
        # turn off axis
        plt.axis('off')
        # plot raw pixel data
        plt.imshow(examples[i, :, :,0], cmap="gray")

# load model
model =
↪  load_model('..\\models\dcgan\w_gaussian_layer\\generator_model_075.h5')
# generate points in latent space
pts = generate_latent_points(250, 2)
# interpolate points in latent space
interpolated = interpolate_points(pts[0], pts[1])
# generate images
X = model.predict(interpolated)
# scale from [-1,1] to [0,1]
#X = (X + 1) / 2.0
# plot the result
plot_generated(X, len(interpolated))
```

```
# ## Interpolate Between Generated Heightmaps Using Spherical Linear
↪  Interpolation (slerp)

# In[18]:


# example of interpolating between generated heightmaps
from numpy import asarray
from numpy import vstack
from numpy.random import randn
from numpy.random import normal
from numpy import arccos
from numpy import clip
from numpy import dot
from numpy import sin
from numpy import linspace
from numpy.linalg import norm
from keras.models import load_model
import matplotlib.pyplot as plt

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
    # generate points in the latent space
    #x_input = randn(latent_dim * n_samples)
    x_input = normal(0, 1, size=(n_samples, latent_dim))

    # reshape into a batch of inputs for the network
    z_input = x_input.reshape(n_samples, latent_dim)

    return z_input

# spherical linear interpolation (slerp)
def slerp(val, low, high):
    omega = arccos(clip(dot(low/norm(low), high/norm(high)), -1, 1))
    so = sin(omega)
    if so == 0:
        # L'Hopital's rule/LERP
        return (1.0-val) * low + val * high
    return sin((1.0-val)*omega) / so * low + sin(val*omega) / so *
    ↪  high

# uniform interpolation between two points in latent space
def interpolate_points(p1, p2, n_steps=10):

    # interpolate ratios between the points
```

```
    ratios = linspace(0, 1, num=n_steps)

    # linear interpolate vectors
    vectors = list()
    for ratio in ratios:
        v = slerp(ratio, p1, p2)
        vectors.append(v)

    return asarray(vectors)

# create a plot of generated images
def plot_generated(examples, n):
    # plot images
    plt.figure(figsize=(15,15))

    for i in range(n * n):
        # define subplot
        plt.subplot(n, n, 1 + i)
        # turn off axis
        plt.axis('off')
        # plot raw pixel data
        plt.imshow(examples[i, :, :,0], cmap="gray")

    plt.show()


# load model
model =
↪  load_model('..\\models\dcgan\w_gaussian_layer\\generator_model_075.h5')
# generate points in latent space
n = 20
pts = generate_latent_points(250, n)
# interpolate pairs
results = None
for i in range(0, n, 2):
    # interpolate points in latent space
    interpolated = interpolate_points(pts[i], pts[i+1])
    # generate images
    X = model.predict(interpolated)
    # scale from [-1,1] to [0,1]
    #X = (X + 1) / 2.0
    if results is None:
        results = X
    else: results = vstack((results, X))
```

```
# plot the result
plot_generated(results, 10)


# ## Explore the Latent Space for Heightmaps

# In[8]:


# example of loading the generator model and generating images
from numpy.random import randn
from numpy.random import normal
from numpy import load
from numpy import mean
from numpy import vstack
from numpy import expand_dims
from keras.models import load_model
import matplotlib.pyplot as plt
from numpy import savez_compressed

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
    # generate points in the latent space
    #x_input = randn(latent_dim * n_samples)
    x_input = normal(0, 1, size=(n_samples, latent_dim))

    # reshape into a batch of inputs for the network
    z_input = x_input.reshape(n_samples, latent_dim)

    return z_input

# create a plot of generated images
def plot_generated(examples, n):
    # plot images
    plt.figure(figsize=(15,15))

    for i in range(n * n):
        # define subplot
        plt.subplot(n, n, 1 + i)
        # turn off axis
        plt.axis('off')
        # plot raw pixel data
        plt.imshow(examples[i, :, :,0], cmap="gray")

    plt.savefig('..\\images\\generated_heightmaps.png')
```

```
    plt.close()

# load model
model =
↪  load_model('..\\models\dcgan\w_gaussian_layer\\generator_model_075.h5')
# generate points in latent space
latent_points = generate_latent_points(250, 100)
# save points
#savez_compressed('..\\latent_points.npz', latent_points)
# generate images
X = model.predict(latent_points)
# scale from [-1,1] to [0,1]
#X = (X + 1) / 2.0
# save plot
#plot_generated(X, 10)


# ### Retrieve specific points

# In[9]:


# retrieve specific points
ridge_ix = [58, 83, 26]
cliff_edge_ix = [21, 75, 90]
high_ground_ix = [33, 62, 2]
# load the saved latent points
data = load('..\\misc\\latent_points.npz')
points = data['arr_0']


# In[10]:


# average list of latent space vectors
def average_points(points, ix):
    # convert to zero offset points
    zero_ix = [i-1 for i in ix]
    # retrieve required points
    vectors = points[zero_ix]
    # average the vectors
    avg_vector = mean(vectors, axis=0)
    # combine original and avg vectors
    all_vectors = vstack((vectors, avg_vector))
    return all_vectors
```

```
# In[11]:


# create a plot of generated images
def plot_generated(examples, rows, cols):
    # plot images
    plt.figure(figsize=(15,15))

    for i in range(rows * cols):
        # define subplot
        plt.subplot(rows, cols, 1 + i)
        # turn off axis
        plt.axis('off')
        # plot raw pixel data
        plt.imshow(examples[i, :, :,0], cmap='gray')
    plt.show()


# In[37]:


# average vectors
ridge = average_points(points, ridge_ix)
cliff_edge = average_points(points, cliff_edge_ix)
high_ground = average_points(points, high_ground_ix)
# combine all vectors
all_vectors = vstack((ridge, cliff_edge, high_ground))
# generate images
images = model.predict(all_vectors)
# scale pixel values
images = (images + 1) / 2.0
plot_generated(images, 3, 4)


# In[44]:


# ridge - cliff_edge + high_ground = ?
result_vector = ridge[-1] - cliff_edge[-1] + high_ground[-1]
# generate image
result_vector = expand_dims(result_vector, 0)
result_image = model.predict(result_vector)
# scale pixel values
```

```
result_image = (result_image + 1) / 2.0
plt.axis('off')
plt.imshow(result_image[0][:,:,0], cmap="gray")
plt.show()
```

Image Cropping Utils.ipynb

```
#!/usr/bin/env python
# coding: utf-8

# ### Here we create HDF5 files using the NASA blue marble high-res
↪  images, for pix2pix translation with GANs.

# Download the images.

# In[ ]:


import certifi
import urllib3
from PIL import Image
from PIL import ImageDraw
from PIL import ImageFont
import io

http = urllib3.PoolManager(cert_reqs='CERT_REQUIRED',
                           ca_certs=certifi.where())

# To prevent a decompression bomb with large files, a file designed
↪  to crash or render useless the program
# or system reading it, i.e. a denial of service. PIL has a limit on
↪  the number of pixels in an image to
# prevent a decompression bomb.

# WARNING!!: Only run this commnd on known trusted files
# Decompression Bomb protection override
Image.MAX_IMAGE_PIXELS = None


# In[ ]:


# download
↪  https://eoimages.gsfc.nasa.gov/images/imagerecords/74000/74218/world.200412.
```

```
# output should be 200
r = http.request('GET',
↪  'https://eoimages.gsfc.nasa.gov/images/imagerecords/74000/74218/world.200412.3
print(r.status)
stream = io.BytesIO(r.data)
img = Image.open(stream)
img.save("data/world.200412.3x21600x10800.jpg", "JPEG", quality=95)


# In[ ]:


# download
↪  https://eoimages.gsfc.nasa.gov/images/imagerecords/73000/73934/gebco_08_rev_
# output should be 200
r = http.request('GET',
↪  'https://eoimages.gsfc.nasa.gov/images/imagerecords/73000/73934/gebco_08_rev_e
print(r.status)
stream = io.BytesIO(r.data)
img = Image.open(stream)
img.save("data/gebco_08_rev_elev_21600x10800.png", "PNG")


# Code to extract chunks.

# In[ ]:


from skimage.io import imread
import numpy as np
from IPython.display import Image
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
import time

# To prevent a decompression bomb with large files, a file designed
↪  to crash or render useless the program
# or system reading it, i.e. a denial of service. PIL has a limit on
↪  the number of pixels in an image to
# prevent a decompression bomb.

# WARNING!!: Only run this commnd on known trusted files
# Decompression Bomb protection override
Image.MAX_IMAGE_PIXELS = None
```

```
# In[ ]:


texture = imread("data/world.200412.3x21600x10800.jpg")


# In[ ]:


texture.shape


# In[ ]:


heightmap = imread("data/gebco_08_rev_elev_21600x10800.png")
heightmap.shape


# ### Test thresholding code for heightmaps

# In[ ]:


# generate random offset
xoff, yoff = np.random.randint(0, heightmap.shape[1]),
↪  np.random.randint(0, heightmap.shape[0])

# crop texture image
texture_crp = texture[ yoff:yoff+512, xoff:xoff+512 ]

# crop heightmap image
heightmap_crp = heightmap[ yoff:yoff+512, xoff:xoff+512 ]

# show plots
plt.subplot(1,2,1)
plt.imshow(texture_crp)
plt.subplot(1,2,2)
plt.imshow(heightmap_crp, cmap="gray")

# show ratio of white to black
crp_sum = (heightmap_crp==0).sum()*1.0
crp_size = np.prod(heightmap_crp.shape)
```

```
print(crp_sum / float(crp_size))


# In[ ]:


def comparator(heightmap_chunk, texture_chunk):
    """
    returns True if we should keep the chunk, otherwise
    False to discard it
    """
    # count how much black there is in the heightmap
    thresh = ((heightmap_chunk==0).sum()*1.0) /
    ↪  float(np.prod(heightmap_chunk.shape))
    # if more than 90% black, skip it
    if thresh > 0.9:
        return False
    return True


# In[ ]:


def get_chunks(img, img2, crop_size=256, stride=1, max_n=None,
↪  debug=False, dry_run=False):
    """
    img: texture map
    img2: height map

    """
    assert img.shape[0] == img2.shape[0] and img.shape[1] ==
    ↪  img2.shape[1]
    ctr = 0
    is_done = False
    for y in range(0, img.shape[0], stride):
        #print(y)
        for x in range(0, img.shape[1], stride):
            chunk = img[ y:y+crop_size, x:x+crop_size ] # texture
            chunk2 = img2[ y:y+crop_size, x:x+crop_size
            ↪  ][:,:,np.newaxis] # heightmap
            if chunk.shape != (crop_size, crop_size, 3):
                continue
            # if the comparator doesn't like the chunk,
            # discard it
            if not comparator(chunk2, chunk):
```

```
                continue
            if dry_run:
                yield None
            else:
                yield chunk, chunk2
                ctr += 1
            if max_n != None and ctr == max_n:
                return


# Create index, to see how many images we have to save

# In[ ]:


t0 = time.time()
ctr = 0
for chunk in get_chunks(texture, heightmap, crop_size=256, stride=50,
↪   max_n=None):
    ctr += 1
    #pass
print("time taken to process: %f sec" % (time.time()-t0))
print("number of patches detected: %i" % ctr)


# Ok now we can write an H5 file out

# In[ ]:


OUT_FILE = 'data_256x256.hd5'


# In[ ]:


'''
import h5py
#if f != None:
#    f.close()
f = h5py.File(OUT_FILE, 'w')
num_train = int(ctr*0.9)
num_valid = int(ctr*0.1)
f.create_dataset('xt', (num_train, 512, 512, 1), dtype='uint8')
f.create_dataset('yt', (num_train, 512, 512, 3), dtype='uint8')
```

```
f.create_dataset('xv', (num_valid, 512, 512, 1), dtype='uint8')
f.create_dataset('yv', (num_valid, 512, 512, 3), dtype='uint8')
'''


# In[ ]:


idxs = [x for x in range(ctr)]
rnd_state = np.random.RandomState(0)
rnd_state.shuffle(idxs)


# In[ ]:


import h5py

f = h5py.File(OUT_FILE, "w")
f.create_dataset("textures", (ctr, 256, 256, 3), dtype='uint8')
f.create_dataset("heightmaps", (ctr, 256, 256, 1), dtype='uint8')

t0 = time.time()
ctr = 0
for chunk1, chunk2 in get_chunks(texture, heightmap, crop_size=256,
↪   stride=50, max_n=None):

    #print(chunk1.shape)
    #print(chunk2.shape)

    #print(chunk2)

    #print(type(chunk1))

    # by passing hd5 storage for now

    f["textures"][idxs[ctr]] = chunk1
    f["heightmaps"][idxs[ctr]] = chunk2

    ctr += 1
print(time.time()-t0)
print("number of patches detected:", ctr)


# In[ ]:
```

```
#good_value = 4660
#idx = np.random.randint(0, f['textures'].shape[0])
idx = 10000
plt.figure(figsize=(15,15))
plt.subplot(1,2,1)
plt.imshow( f['textures'][idx] ) #10000
plt.subplot(1,2,2)
plt.imshow( f['heightmaps'][idx][:,:,0],cmap="gray" ); #1000


# In[ ]:


f.close()


# ### Greate a n x n grid of real images

# In[ ]:


from numpy.random import choice
from numpy import ones
import h5py
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')


# In[ ]:


def load_real_samples(filename):
    # load image data, Read HDF5 file
    f = h5py.File(filename, 'r')
    dataset = f['xt']
    print('Dataset input shape {}'.format(str(dataset.shape)))

    return dataset


# In[ ]:
```

```
# select real samples
def generate_real_samples(dataset, n_samples):

    # create index numbers
    ix = range(dataset.shape[0])

    # choose n_samples of integers at random without replacement
    # sort the result
    ix = sorted(choice(ix, size=n_samples, replace=False))

    # retrieve selected images using fancy indexing
    X = dataset[ix]

    # generate 'real' class labels (1)
    y = ones((n_samples, 1))

    return X, y


# In[ ]:


def save_plot(examples, epoch, n=5):


    # plot image
    for i in range(n * n):
        # define subplot
        plt.subplot(n, n, 1 + i)

        # turn off axis
        plt.axis('off')

        # plot raw pixel data
        plt.imshow(examples[i, :, :, 0], cmap='gray_r')


    # save plot to file
    filename = '..\\images\\real_plot_e%03d.png' % (epoch + 1)
    plt.savefig(filename)
    plt.show()
    plt.close()


# In[ ]:
```

```
num_samples = 25
sample_num = 1

#dataset =
↪  load_real_samples('data_256x256_train_val_float_scaled.hd5')
dataset =
↪  load_real_samples('data_256x256_train_val_float_scaled_neg1_to_1.hd5')
dataset.shape
X_real, y_real = generate_real_samples(dataset, num_samples)
#save_plot(X_real, sample_num)
f.close()


# ### Convert 1 channel image (grayscale) 3 channel (RGB)

# In[ ]:


import numpy as np
from numpy.random import choice
from numpy.random import randint
from numpy import ones
import matplotlib.pyplot as plt
import cv2
import h5py


# In[ ]:


cv2.__version__


# In[ ]:


def load_real_samples(filename):
    # load image data, Read HDF5 file
    f = h5py.File(filename, 'r')
    dataset = f['xt']
    print('Dataset input shape {}'.format(str(dataset.shape)))

    return dataset
```

```
# In[ ]:


# select real samples
def generate_real_samples(dataset, n_samples):

    # create index numbers
    ix = range(dataset.shape[0])

    # choose n_samples of integers at random without replacement
    # sort the result
    ix = sorted(choice(ix, size=n_samples, replace=False))

    # retrieve selected images using fancy indexing
    X = dataset[ix]

    return X


# In[ ]:


num_samples = 1 # retrieve one image

dataset = load_real_samples('data_256x256_train_val_float_scaled.hd5')
X_real= generate_real_samples(dataset, num_samples)

img = X_real[0]

img.shape


# In[ ]:


plt.figure()
plt.imshow(img[:,:,0],cmap="gray")
plt.clim((0, 1))

plt.show()


# In[ ]:
```

```
img2 = cv2.merge((img,img,img))


# In[ ]:


img2.shape


# In[ ]:


type(img2)


# In[ ]:


plt.figure()
plt.imshow(img2, cmap="gray_r")
plt.clim((0, 1))

plt.show()


# ### Split dataset into train and validtion and scale data
# Also remove images that contain mostly black pixels

# In[1]:


import h5py

input_file = 'data_256x256.hd5'

f1 = h5py.File(input_file, "r")


# In[2]:


# Show the dataset names
for key in f1.keys():
```

```
    print(key)


# In[3]:


# Load datasets
heightmaps_data = f1['heightmaps']
textures_data = f1['textures']

# Output dataset shapes
print(heightmaps_data.shape)
print(textures_data.shape)

# Count the number of images in data set
ctr = heightmaps_data.shape[0]


# In[4]:


import matplotlib.pyplot as plt

image = textures_data[10500]
print(image.min())
print(image.max())

plt.figure(figsize=(10,10))

# plot image
# define subplot
plt.subplot(1, 1, 1)

# turn off axis
plt.axis('off')

# plot raw pixel data
plt.imshow(image);


# In[5]:


scaled_img = (textures_data[10500] - 127.5)/127.5
```

```
print(scaled_img.min())
print(scaled_img.max())


# In[6]:


import matplotlib.pyplot as plt

plt.figure(figsize=(10,10))

# plot image
# define subplot
plt.subplot(1, 1, 1)

# turn off axis
plt.axis('off')

# plot raw pixel data
plt.imshow((scaled_img + 1) / 2.0);


# In[7]:


def comparator(heightmap_chunk, texture_chunk):
    """
    returns True if we should keep the chunk, otherwise
    False to discard it
    """
    # count how much black there is in the heightmap
    thresh = ((heightmap_chunk==0).sum()*1.0) /
    ↪  float(np.prod(heightmap_chunk.shape))
    # if more than 90% black, skip it
    if thresh > 0.9:
        return False
    return True


# In[11]:


import numpy as np

# Portion of dataset for test and validation
```

```
num_train = int(ctr*0.9)
num_valid = ctr - num_train

# Create random index for subsetting data for test and train
idxs = [x for x in range(ctr)]

rnd_state = np.random.RandomState(0)
rnd_state.shuffle(idxs)

train_idxs = idxs[:num_train]
val_idxs = idxs[num_train:]


# Create HDF5 file with test and validation
f2 = h5py.File('data_256x256_train_val_float_scaled_neg1_to_1.hd5',
↪  'w')
#f2 = h5py.File('data_test_val_float_scaled.hd5', 'w')
f2.create_dataset('xt', (num_train, 256, 256, 1), dtype='float32')
f2.create_dataset('yt', (num_train, 256, 256, 3), dtype='float32')
f2.create_dataset('xv', (num_valid, 256, 256, 1), dtype='float32')
f2.create_dataset('yv', (num_valid, 256, 256, 3), dtype='float32')


# iterate though train idxs and select images from heightmap and
↪  texture
for idx in train_idxs:

    # retreive data from source data
    heightmap_chunk = heightmaps_data[idx]
    texture_chunk = textures_data[idx]

    # check if image is mostly black pixels
    if comparator(heightmap_chunk, texture_chunk):

        # scale 0 to 1
        #heightmap_chunk = heightmap_chunk /255.0
        #texture_chunk = texture_chunk / 255.0

        # scale -1 to 1
        heightmap_chunk = (heightmap_chunk - 127.5) / 127.5
        texture_chunk = (texture_chunk - 127.5) / 127.5

        # swap index values from old index to new index
        train_idx = train_idxs.index(idx)
```

```
            # write data
            f2['xt'][train_idx] = heightmap_chunk
            f2['yt'][train_idx] = texture_chunk


    # iterate though validation idxs and select images from heightmap
    ↪  and texture
    for idx in val_idxs:

        # retreive data from source data
        heightmap_chunk = heightmaps_data[idx]
        texture_chunk = textures_data[idx]

        # check if image is mostly black pixels
        if comparator(heightmap_chunk, texture_chunk):

            # scale 0 to 1
            #heightmap_chunk = heightmap_chunk/255.0
            #texture_chunk = texture_chunk/255.0

            # scale -1 to 1
            heightmap_chunk = (heightmap_chunk - 127.5) / 127.5
            texture_chunk = (texture_chunk - 127.5) / 127.5

            # swap index values from old index to new index
            val_idx = val_idxs.index(idx)

            # write data
            f2['xv'][val_idx] = heightmap_chunk
            f2['yv'][val_idx] = texture_chunk

    f1.close()
    f2.close()


    # ### View sampled image from train and validation datasets

    # In[ ]:


    import h5py
    import matplotlib.pyplot as plt
    get_ipython().run_line_magic('matplotlib', 'inline')
```

```
f2 = h5py.File('data_256x256_train_val_float_scaled_neg1_to_1.hd5',
↪  'r')

idx = 42
plt.figure(figsize=(15,15))
plt.subplot(1,2,1)
plt.imshow( f2['xt'][idx][:,:,0],cmap="gray" )
plt.subplot(1,2,2)
plt.imshow( f2['yt'][idx] )

plt.figure(figsize=(15,15))
plt.subplot(1,2,1)
plt.imshow( f2['xv'][idx][:,:,0],cmap="gray" )
plt.subplot(1,2,2)
plt.imshow( f2['yv'][idx] );
f2.close()


# ## Explore n x n image matrix

# In[ ]:


# example of reading in data file with latent points
import h5py
import matplotlib.pyplot as plt

# create a plot of generated images
def plot_generated(examples, n):
    # plot images
    plt.figure(figsize=(15,15))

    for i in range(n * n):
        # define subplot
        plt.subplot(n, n, 1 + i)
        # turn off axis
        plt.axis('off')
        # plot raw pixel data
        plt.imshow((examples[i,:,:,:] + 1) / 2.0)



# load images
input_file = 'data_256x256_train_val_float_scaled_neg1_to_1.hd5'
```

```
f = h5py.File(input_file, "r")
Y = f['yt'][0:225]

# generate images
plot_generated(Y, 15)
f.close()
```

Image Similarity.ipynb

```
#!/usr/bin/env python
# coding: utf-8

# # Image Similarity
# This notebook compares the similarity of two images using Mean
↪  Squared Error (MSE) and Structural Similarity Index

# In[20]:


from skimage.measure import compare_ssim as ssim
import matplotlib.pyplot as plt
import numpy as np
import cv2


# In[37]:


def mse(imageA, imageB):
    # the 'Mean Squared Error' between the two images is the
    # sum of the squared difference between the two images;
    # NOTE: the two images must have the same dimension
    err = np.sum((imageA.astype("float") - imageB.astype("float")) **
    ↪  2)
    err /= float(imageA.shape[0] * imageA.shape[1])

    # return the MSE, the lower the error, the more "similar"
    # the two images are
    return err

def compare_images(imageA, imageB, title):
    # compute the mean squared error and structural similarity
    # index for the images
```

```
    m = mse(imageA, imageB)
    s = ssim(imageA, imageB, multichannel=True)

    # setup the figure
    fig = plt.figure(title)
    plt.suptitle("MSE: %.5f, SSIM: %.5f" % (m, s))

    # show first image
    ax = fig.add_subplot(1, 2, 1)
    #plt.imshow(imageA, cmap = plt.cm.gray)
    plt.imshow(imageA)
    plt.title('Target')
    plt.axis("off")

    # show the second image
    ax = fig.add_subplot(1, 2, 2)
    #plt.imshow(imageB, cmap = plt.cm.gray)
    plt.imshow(imageB)
    plt.title('Generated')
    plt.axis("off")

    # show the images
    plt.show()


# ## Generate two images and compare similarity

# In[26]:


from numpy.random import randint
import h5py
from keras.models import load_model
from cv2 import merge
import matplotlib.pyplot as plt

# load and prepare training images
def load_real_samples(filename):
    # load image data, Read HDF5 file
    f = h5py.File(filename, 'r')
    X1 = f['xt']
    X2 = f['yt']
    print('Dataset input shapes {}, {}'.format(str(X1.shape),
    ↪  str(X2.shape)))
```

```
    return [X1, X2]

# covert grayscle 1-channel to RGB 3-channel
def gray2rgb(gray_img):
    ''' Convert on 1-channel image to 3-channel image
        using CV

    '''

    # convert to 3-channel image
    img = merge((gray_img,gray_img,gray_img))

    return img

# plot source, generated and target images
def plot_images(src_img_1ch, gen_img, tar_img):

    # scale from [-1, 1] to [0, 1]
    gen_img = (gen_img + 1) / 2.0
    tar_img = (tar_img + 1) / 2.0

    # plot images row by row
    # define subplot
    plt.subplot(1, 3, 1)
    # turn off axis
    plt.axis("off")
    # plot raw pixel data
    plt.imshow(src_img_1ch[:,:,0], cmap="gray")
    # show title
    plt.title('Source')

    # define subplot
    plt.subplot(1, 3, 2)
    # turn off axis
    plt.axis("off")
    # plot raw pixel data
    plt.imshow(gen_img[0,:,:,:])
    # show title
    plt.title('Generated')

    # define subplot
    plt.subplot(1, 3, 3)
    # turn off axis
    plt.axis("off")
```

```
    # plot raw pixel data
    plt.imshow(tar_img[0,:,:,:])
    # show title
    plt.title('Expected')

    # plot layout
    plt.tight_layout()

# load data set
dataset =
↪  '..\\gan-heightmaps\\data_256x256_train_val_float_scaled_neg1_to_1.hd5'
X1, X2 = load_real_samples(dataset)

# load model
model = load_model('..\\models\pix2pix\\model_000030.h5')

# select random example, change the 1-channel image to 3-channel
ix = randint(0, len(X1), 1)[0]
src_image_1ch = X1[ix]
tar_image = X2[ix]

n = tar_image.shape[0]

src_image_3ch = gray2rgb(X1[ix]).reshape(1, n, n, 3)
tar_image = X2[ix].reshape(1, n, n, 3)

# generate image from source
gen_image = model.predict(src_image_3ch)

# plot all three images
plot_images(src_image_1ch, gen_image, tar_image)


# In[38]:


compare_images(tar_image[0,:,:,:], gen_image[0,:,:,:], 'Compare
↪  Images')


# ## Generate five images and compare similarity

# In[2]:
```

```
import h5py
from numpy.random import choice
from numpy import asarray
from keras.models import load_model
from cv2 import merge


# In[33]:


# load and prepare training images
def load_real_samples(filename):
    # load image data, Read HDF5 file
    f = h5py.File(filename, 'r')
    X1 = f['xt']
    X2 = f['yt']
    print('Dataset input shapes {}, {}'.format(str(X1.shape),
    ↪  str(X2.shape)))

    return [X1, X2]


def generate_real_samples(datasetA, datasetB, n_samples):

    # choose random instances
    ix = range(datasetA.shape[0])

    # choose n_samples of integers at random without replacement
    # sort the result
    ix = sorted(choice(ix, size=n_samples, replace=False))

    # retrieve selected images
    X1, X2 = datasetA[ix], datasetB[ix]

    return [X1, X2]


# generate a batch of images, returns images and targets
def generate_fake_samples(g_model, samples):

    # generate fake instance
    X = g_model.predict(samples)

    return X
```

```
# covert grayscle 1-channel to RGB 3-channel
def gray2rgb(gray_img):
    ''' Convert on 1-channel image to 3-channel image
        using CV

    '''

    # convert to 3-channel image
    img = merge((gray_img,gray_img,gray_img))

    return img

def create_dataset(g_model, datasetA, datasetB, n_samples=5):

    # select a sample of input images
    [X_realA, X_realB] = generate_real_samples(datasetA, datasetB,
    ↪  n_samples)

    X_realA_3ch = []

    for i in range(X_realA.shape[0]):

        X_realA_3ch.append(gray2rgb(X_realA[i]))

    X_realA_3ch = asarray(X_realA_3ch)

    # generate a batch of fake samples
    X_fakeB = generate_fake_samples(g_model, X_realA_3ch)



def summarize_similarity(datasetA, datasetB):

    # compare images and append to array
    imgs_mse = []
    imgs_ssim = []

    for i in range(datasetA.shape[0]):

        imageA = datasetA[i]
        imageB = datasetB[i]

        compare_images(imageA, imageB, 'Compare Images')
```

```
        imgs_mse.append(mse(imageA, imageB))
        imgs_ssim.append(ssim(imageA, imageB, multichannel=True))

    print('MSE')
    print(imgs_mse)
    print('SSIM')
    print(imgs_ssim)


# In[34]:


# Load data set
trainA, trainB =
↪  load_real_samples('..\\gan-heightmaps\\data_256x256_train_val_float_scaled_neg

# select a sample of 5 input and target images
[X_realA, X_realB] = generate_real_samples(trainA, trainB, 5)

X_realA_3ch = []

for i in range(X_realA.shape[0]):

    X_realA_3ch.append(gray2rgb(X_realA[i]))

X_realA_3ch = asarray(X_realA_3ch)


# In[42]:


# Change the model at different saved epochs to compare quality
# then rerun this cell
# load model
g_model = load_model('..\\models\pix2pix\\model_000001.h5')

# generate a batch of fake samples
X_fakeB = generate_fake_samples(g_model, X_realA_3ch)


# In[43]:


# Run similarity
summarize_similarity(X_realB, X_fakeB)
```

```
# ## References
#
# Wang, Z., Bovik, A., Sheikh, H., & Simoncelli, E. (2004). Image
↪  Quality Assessment: From Error Visibility to Structural
↪  Similarity. IEEE Transactions on Image Processing, 13(4),
↪  600{612. doi: 10.1109/tip.2003.819861
```

Turning Test Image Generator.ipynb

```
#!/usr/bin/env python
# coding: utf-8

# # Create images to be used in turning test

# ## Retreive real images

# In[1]:


import h5py
from numpy.random import choice
from numpy.random import normal
from numpy import vstack
from keras.models import load_model
import matplotlib.pyplot as plt


# In[3]:


# load and prepare training images
def load_real_samples(filename):
    # load image data, Read HDF5 file
    f = h5py.File(filename, 'r')
    X1 = f['xt']
    X2 = f['yt']
    print('Dataset input shapes {}, {}'.format(str(X1.shape),
    ↪  str(X2.shape)))

    return [X1, X2]
```

```
def generate_real_samples(datasetA, datasetB, n_samples):

    # choose random instances
    ix = range(datasetA.shape[0])

    # choose n_samples of integers at random without replacement
    # sort the result
    ix = sorted(choice(ix, size=n_samples, replace=False))

    # retrieve selected images
    X1, X2 = datasetA[ix], datasetB[ix]

    return [X1, X2]


# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
    # generate points in the latent space
    #x_input = randn(latent_dim * n_samples)

    #x_input = uniform(-1.0, 1.0, size=[n_samples, latent_dim])

    x_input = normal(0, 1, size=(n_samples, latent_dim))


    #print(x_input)

    # reshape into a batch of inputs for the network
    x_input = x_input.reshape(n_samples, latent_dim)

    return x_input


# use the generator to generate n fake examples, with class labels
def generate_fake_samples(g_model, latent_dim, n_samples):
    # generate points in latent space
    x_input = generate_latent_points(latent_dim, n_samples)

    # predict outputs
    X = g_model.predict(x_input)

    return X


# create a plot of generated images
```

```
def plot_generated(examples, n):
    # plot images
    plt.figure(figsize=(15,15))

    for i in range(n):
        # define subplot
        plt.subplot(1, n, 1 + i)
        # turn off axis
        plt.axis('off')
        # plot raw pixel data
        plt.imshow(examples[i, :, :,0], cmap="gray")


# In[4]:


# Load data set
trainA, trainB =
↪  load_real_samples('..\\gan-heightmaps\\data_256x256_train_val_float_scaled_neg

# Retrieve real images
realA, _ = generate_real_samples(trainA, trainB, 5)

# Plot images
plot_generated(realA, 5)


# ## Generate fake samples

# In[ ]:


# load saved model
model =
↪  load_model('..\\models\dcgan\w_gaussian_layer\\generator_model_100.h5')

# prepare fake examples
x_fake = generate_fake_samples(model, 250, 5)

# Plot images
plot_generated(x_fake, 5)


# ## Create index and shuffle images
```

```
# In[29]:


# create a plot of generated images
def plot_generated(examples, key, n):
    # plot images
    plt.figure(figsize=(15,15))

    for i in range(n):
        # define subplot
        plt.subplot(1, n, 1 + i)
        # turn off axis
        plt.axis('off')
        # plot raw pixel data
        plt.imshow(examples[key[i], :, :,0], cmap="gray")


# In[2]:


# Stack real and fake images
combined_imgs = vstack((realA, x_fake))

# Create index values
ix = list(range(combined.shape[0]))

# Shuffle
key = list(choice(ix, size=10, replace=False))
print(key)

# In[32]:


# Output images
plot_generated(combined_imgs, key, 10)
```