Terrain Modelling with Generative AI

Authors: Ewan Walker, Harry Fletcher, Ed Warneken
Supervisor: Dr Nick Keepfer
MAS3091 Group 8

December 25, 2023

Contents

1 Overview

1.1 Introduction

In this report we will start by looking in detail into the different methods used for terrain generation, explaining the processes of each method and their uses. We will look at cellular automata, Voronoi/Worley noise, diamond-square method, perlin noise and in particular the midpoint displacement method which we will look to recreate in Python ourselves. We want to use these methods to generate our own complex terrain. We will be using Principle Component Analysis to help us reduce the complexity of the data which allows us to use an auto-encoder on this simplified data. We will also be using inverse Principal Component Analysis to invert the data once it's run through the auto-encoder.

Throughout the project, we will reference our code compiled on GitHub [1].

1.2 Objectives

A detailed outline of the objectives are as follows:

Code multiple of these methods for analysis by varying parameters and create plots, colourmaps and 3D projections (where applicable). Implement reproducibility via the use of seeds (an initialisation of pseudorandom number generators to recreate the same 'random' terrain). Utilise noise in heightmaps – fluently blended 'random' normalised values (ranging from 0 to 1) sorted in matrices to create 3D projections using the values as height components. Explore principal component analysis to identify the most important data-points representing the terrain and eliminate any excess for dimensionality reduction, thereby compressing the volume of data. Find out the optimum number of principle components to minimize loss of quality when compared to the original terrain. Build autoencoders, trained on the generated heightmaps, to create a 2D grid in which clusters of points correlate to specific features of the terrain. These are much more efficient than traditional methods. Analyse real-world LiDAR scans via the autoencoders to create realistic, new, predictable terrain from a few select coordinates.

Clearly, that is a lot to unpack! However, the simplistic beauty of the process can be quickly illustrated, and all the fancy concepts and definitions are reduced to an easily understandable format. Firstly, to understand what we mean by 'terrain', take a widely recognised example: Minecraft. In Minecraft, vast, unique worlds are near-instantaneously spawned into existence for the player to explore. They are random; they also contain consistently predictable patterns. One can reproduce infinitely generating, complex random worlds using a short string of numbers, a seed, whilst no

two seeds are the same. How can this be? Below is information to lay a bedrock understanding of all these processes.

1.2.1 Example Applications in Video Games

Minecraft
Terraria
World Anvil

---

2 Terrain Generation Methods

2.1 Midpoint Displacement

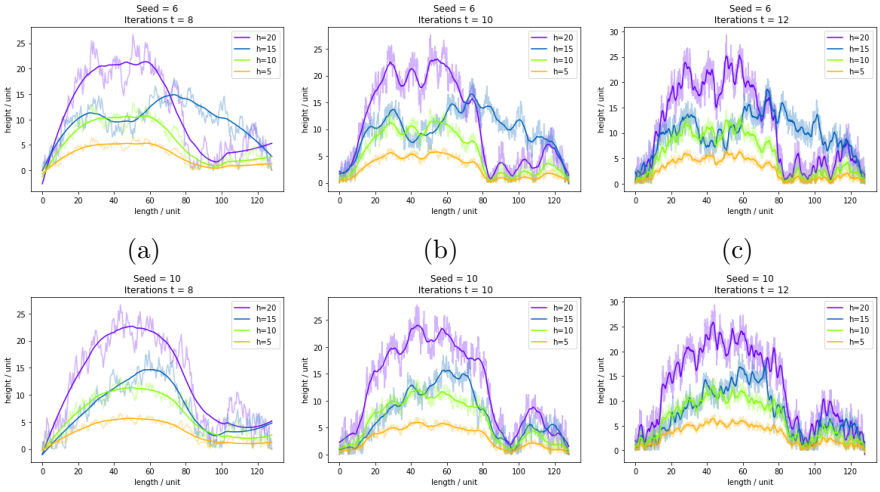
The midpoint displacement [2] method is a recursive algorithm frequently used to create fractal landscapes primarily in the two dimensional space. This method creates complex and aesthetically pleasing terrain by gradually adding controlled randomness to line segments, which are scaled by a h parameter that depends on an iteration number t. The purpose is to create a 2D landscape but it can also be used for graphical applications (i.e. rivers/ravines/paths).

For our code, we simply started from two bounding x values, each at y=0. The elevation never goes below the x-axis - i.e. if the displaced y is less than 0, then reverse the negative sign. The initial midpoint is always at the input displacement height. Input a random integer for variation. Include a seed for the pseudo random number generator to initialise and an input number of iterations. At each repetition of finding midpoints, gradually scale down the y-displacement so that it produces a more smooth, natural result. Remove the large fluctuations of the coordinates (noise) by using the 'Savitzky-Golay filter' [3] method. It basically finds line equations for the least square fit for incremental ranges of x and then records coordinates on the lines in an array. The plots of different displacement values are colour-coordinated for easy visualisation and analysis. We have setup the coding as a module to simply call a function and return some elevation, where input parameters are: first x value, second x value, displacement h, number of iterations, seed. The general function has relevant help if called for - help(function).

Analysis

For analysis, We have created 6 example plots: 2 seeds, 3 variations of iterations per seed, fixing the initial x values.

Looking at 3 plots of the same seed, it's clearly depicted that the code is very sensitive to the number of iterations. Varying by -2 and +2 iterations from an arbitrarily discovered optimum of 10 produces elevation either very uninteresting or too overpowering (even when noise dampened).



(a) (b) (c)

(d) (e) (f)

Figure 1: Midpoint Displacement Plots

Looking at an individual plot, the varied displacement functions similarly to my hypothesis: i.e. descending values produce much flatter, and lower plot.

2.2 Cellular Automata

Purpose: start with a specific order of initial conditions (seed) from which rules are procedurally followed at each given discrete time (ticks) to result in the exact same construction for any given tick value, upon recreation (i.e. each iteration is a pure function of the previous). Useful for cave generation.

The procedure for this method [4] is as follows: First, we use the seed to create a grid containing a unique set of live coordinates/cells. We then iterate the current set through an algorithm to create a new set x times such that: a live cell remains if 2 or 3 adjacent cells are live, a dead cell becomes live if strictly 3 adjacent cells are live. All other live/dead cells are now dead. (Optional) Floodfill - fill all the secondary living areas as appropriate by setting nodes and filling adjacent cells one after another until the classification changes

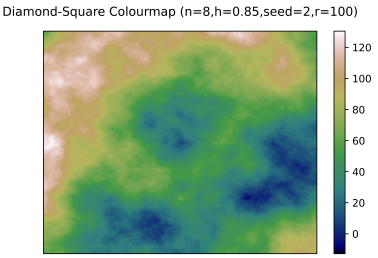
2.3 Voronoi/Worley Noise

The purpose is for biome generation and the procedure [5] is as follows. Randomly distribute a set of feature points, 1 in each cell of a 2D grid of set resolution, usually done as procedural noise to mimic a Poisson disc. Find the direction vector of each given location to its the nearest feature point.

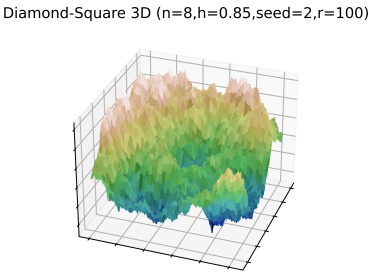
Gradient or colour the 2D space at each coordinate based on the magnitude of its direction vector. The variable is the resolution of the grid. We chose to further investigate other types of noise in this project.

2.4 Diamond-Square

This method begins with a 2D grid (size $2^n + 1$) and createsa matrix of heightmap values [6]. These essentially are the z coordinates used for generating a 3D surface. Starting with a 2D array, initialise the 4 corners with some random value. Firstly, a diamond step takes an average of the 4 nearest corners and adds some random noise - think of this as diagonal. Secondly, a square step takes an average of the nearest horizontal and vertical points (an average of 3 if an edge) with additional noise. These are repeated until the array is full, decreasing the range of randomness with each iteration by multiplying the random value by 2^h (where h is between 0 and 1).



(a) 2D Colourmap of Diamond-Square

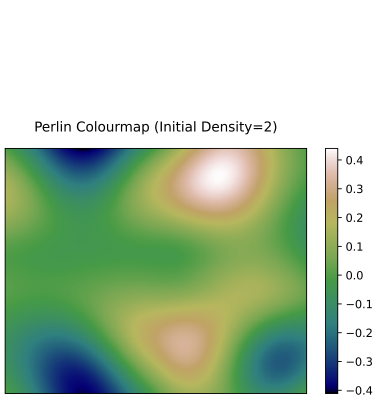


(b) 3D Projection of DIamond-Square

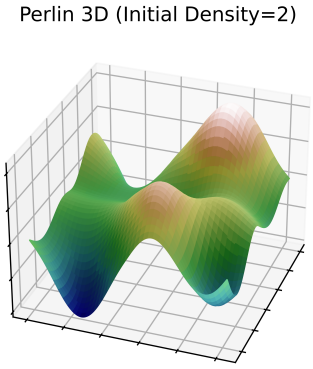
Figure 2: Diamond Square

2.5 Perlin Noise

Ken Perlin [7] developed a method for generating heightmaps in any dimension, with a specific focus here on transitioning from 2D to 3D. The process unfolds via a series of steps denoted as A to B to C. Initially, a 2D grid of cells is created, with each intersection assigned a unit vector of gradient ranging from -1 to 1. Moving forward, for every pixel encompassing the entire grid, the displacement vector (d) is computed relative to its nearest cell corner. The subsequent step involves calculating the dot product between all these displacement vectors and their corresponding corner vectors, as vividly illustrated in image B with color coordination. This process is iterated for the remaining three corners of each cell, resulting in a total of four sets of gradients. These sets are then amalgamated using a "smoothstep function" to yield the heightmap represented by point C. To enhance clarity, numerous versions of these heightmaps are combined, known as octaves. As the density of the initial grid increases, these octaves are added together with a reduced weight or transparency. The cumulative effect of these steps produces a smooth and realistic terrain, as we desire.



(a) 2d Colourmap of Perlin Noise



(b) 3D Projection of Perlin Noise

Figure 3: Perlin Noise Figures

3 Principal Component Analysis

Explanation:

Principal Component Analysis (PCA) [8] is a statistical method used for dimensionality reduction in data by transforming variables into a new set of uncorrelated components (called principal values). The most significant information is captured while minimizing loss of data variance.

Procedure:

In performing dimensionality reduction through Principal Component Analysis (PCA), a systematic approach is followed. Initially, the dataset undergoes standardization to ensure uniform contribution from each data point, facilitating a fair analysis. Subsequently, the covariance matrix is computed, providing a clear summary of correlations within the data and allowing variables that contribute significantly to the explained variance to be identified. Following this, the eigenvalues and eigenvectors of the covariance matrix are determined, representing variance and principal values, respectively. These eigenvalues are then ranked in descending order, with the top eigenvalue selected to form a transformation matrix for dimensionality reduction. The chosen principal components serve as a basis for projecting the standardized data onto a lower-dimensional space, effectively preserving essential information and reducing the overall dimentionality of the dataset.

3.1 Why do we want to use PCA?

With the use of PCA, we are able to greatly reduce the dimensionality of the data so that we can apply the auto-encoder to our data. Not only can we reduce the dimensionality of the data, but once we've run the data via our autoencoder, we can then increase the dimensionality so that it is similar to that of the original data, thus allowing us to produce realistic terrains with a substantial volume of information. Once we've applied the PCA algorithm to this data and ran it through our autoencoder, we can use the inverse PCA algorithm which allows the produced terrain from the autoencoder to have its dimensions increased, providing a higher resolution image similar to that of the original data.

3.2 LiDAR - Plotting Real-World Elevation Data

This can be done using real-world LiDAR scans, by taking the elevation data and converting it to an array within Python so that the data can be plotted, and the PCA algorithm can be applied to this data to reduce the dimensionality.

3.3 Principal Components

Here you can see how the varying number of principal components affects the percentage of variance explained. It becomes evident that after a certain number, adding more becomes futile as such little extra variance is explained by the later principal values. This is why we wanted to find a balance for the number of principal components since if we didn't have enough, the terrains would look unrealistic, but with too many, too much data is needed which our devices were unable to handle.

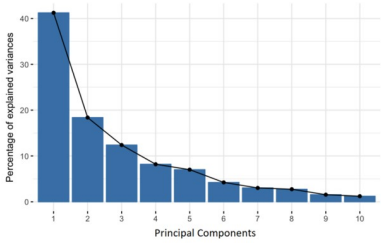


Figure 4: Principal Values and the Variance Explained

---

4 Autoencoders

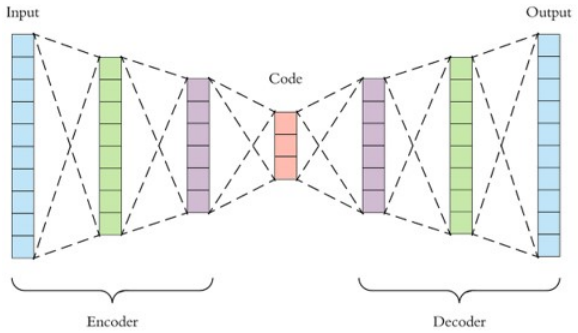


Figure 5: General autoencoding Theory [10]

4.1 Overall Autoenocder Theory

Auto-encoders [11] are useful for un-supervised learning applications for optimizing the reducing of the dimensionality of the given dataset and then for abstraction (return back to the original data); In addition, similar models can be used for de-noising data. Simply, they are particular training methods for Deep Learning (DL) models, in which the mapping from one layer to

the next is not coded by the engineer, but are learned from the data by using a general-purpose learning procedure (the autoencoder). This composes a particular DL called a Deep Neural Network (DNN).

Architecture/construction of the autoecoder

During the training of the autoencoder to structure the system for precise computation, both encoders and decoders are used. This is then reduced to simply just the encoder, in the application of fresh data. The overall architecture functions through a layered neural network, commencing with the input layer and progressing through a series of encoder layers. The initial encoder refines the input data, followed by additional layers, each featuring an undetermined reduced number of neurons. The penultimate encoder layer acts as a bottleneck, compressing the data dimensionally and capturing its maximal signals. On the decoding side, the process unfolds in reverse, starting with the first decoder layer and progressing through layers with an increased number of neurons. The final output layer mirrors the structure of the input layer, maintaining an equivalent number of neurons. This hierarchical structure, from input to output, demonstrates the intricate transformation and representation of the data, with each layer playing a crucial role in the overall functionality of the system.

What does the autoencoder achieve with this process?

As we saw in the PCA process, varying the number of principal components vastly affects the accuracy of the reconstruction. There is quite clearly an optimal number, to be determined. The objective while training the autoencoder is to minimise said reconstruction error between the input and the output of the DNN. In this process, most of the important patterns in the data are recognised effectively enough for precise reconstruction. A Loss Function (LF) calculates this loss during the training of the autoencoder.

An example of a LF for continuous input is the squared error loss, given by:

$$\mathcal{L}\left(x, x'\right) = \left\|x - x'\right\|^2 = \left\|x - W'(Wx + b) + b'\right\|^2$$

where, x is the input, W, b is the weight vector and bias of the encoder, and W0, b are the weight vector and bias of the decoder.

When considered in batches of n samples, the mean square error for the batch could be computed by the formulae

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}\left(x_i, x_i'\right)$$

4.2 MNIST Dataset Training

In essence, autoencoders are mirrored neural networks.

The key idea here is to understand the process of mapping of a representation of something to its conceptual value. For example, when a human views a '9', the brain converts the light input signals and interprets that particular composition of information as a 9. This goes for anything, including the very letters you are reading here. But what if you were to draw a '9' on a piece of paper, use a camera to capture a photograph and ask a computer: what is this? A question we so fortunately deem easy is more complex, when considered on a deeper level. Every individual human has their own unique handwriting, and there exists some variation with each instance; yet we are all able to distinguish the '9' from literally anything else in the entire universe that is not a nine.

"What is this?" one asks a computer.

"Ah ha! That is a 1 followed by a 0 and a 1 and a 1 and a 0 and a 1 and a 0!" the computer proudly pronounces. "I'm glad I can help identifying what you're looking at!"

Our point here is that, in order to digitise the conceptual value of the image, we must write a script that actually reads what it represents, explicitly not a bunch of 1s and 0s. How do we go about this? We train the computer by building an autoencoder model by gathering a large collection of images with one underlying identifiable theme of which we know what they actually are.

One such example of this is the MNIST [12] dataset, consisting of 70000 hand-drawn digit images of 28 pixels by 28 pixels and their values are assigned by a human. These are grouped into 'batches', say 64 randomly selected images from the dataset per batch, to all be fed into the autoencoder once each time the script is ran. Next, stack the batches on top of one another in a Tensor, using the PyTorch library. We create 2 tensors: one for a 'training set' (60,000 of the 70,000) and another for the 'testing set' (10,000 of the 70,000). If the number of batches doesn't exactly divide the dataset, we divide it and then roof it for the tensor size – round it up to the next integer. One solution to this problem is that the last batch just has less images than the others.

Name	Type	Size	Value
batch_size	int	1	64
test_data	datasets.mnist.MNIST	10000	MNIST object of torchvision.datasets.mnist module
test_dataloader	utils.data.dataloader.DataLoader	157	DataLoader object of torch.utils.data.dataloader module
train_dataloader	utils.data.dataloader.DataLoader	938	DataLoader object of torch.utils.data.dataloader module
training_data	datasets.mnist.MNIST	60000	MNIST object of torchvision.datasets.mnist module

Figure 6: MNIST dataset

Note: The number of channels is just the number of different 'themes' we are looking for. Here, the MNIST dataset (and later the Perlin Noise generations) are on a scale of white to black, so only one channel is required.

Compare that with a coloured image of red, green and blue, which would require 3 channels for the 3 different gradients.

To create our model autoencoder on the MNIST dataset, we train it using the specified training data, for a number of epochs – the amount of times the batches are randomly selected and input into the autoencoder for training. But what exactly happens during training? We must define our model layers explicitly. In our project, we have chosen to use dense layers. These dense layers are sequenced as linear functions between one another, each looking at every single dataset and changing the number of neurons (datapoints) that represent each individual image. With each subsequent layer, additional time is taken but more data is preserved through the process. One alternative type of layer is a convolutional layer, whereby a filter is used (imagine a small radius of pixels neighbouring a specific centre pixel, changing the centre pixel as appropriate) and the weights are shared for all the pixels. Each layer is fully connected to another, i.e. every neuron has some weighted connection to each neuron in the next layer. It's is important to note that the input of a layer must match the size of the output from the previous. Between the layers, we have used a 'ReLU' (rectified linear unit) which is a non-linear function as shown in the figure, increasing the model's predictive capabilities; it basically just zeros the negative values shown below.

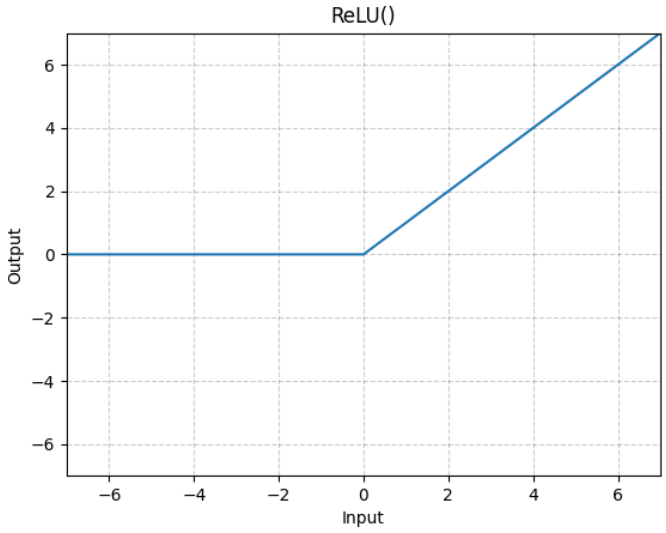


Figure 7: Graph showing the output and input using ReLU inbetween the layers [13]

As explained previously, the model completes a forward pass per batch, calculating the loss (MSE) compared to original batch data. Each image is

first flattened, simply converting the 28px by 28px into a single 784 string. We use the 'Adam' optimizer from the PyTorch library which adjusts the layer weights (and bias, as applicable) during back-propagation and set the 'learning rate' parameter to be 0.001; this essentially changes how much we update the weights with each batch passed through. Upon periodically printing the current loss at sequenced times in the console, we observe that the mean loss decreases as the number of batches and epochs are used in training, as expected. Once trained, the test dataset is used to see how good our model is. You can then show the model a new hand-written digit and it can predict what it represents very accurately.

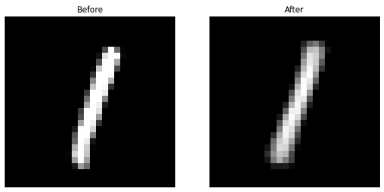


Figure 8: Before and After Pytorch's 'Adam'

To analyse the inputs vs the outputs, we plot a figure of various corresponding subplots due to time and restricted processing power capabilities towards the end of the project, we have only done this for the first epoch, shown below. We have modified our code to display our passed images, rather than emit the corresponding numerical values. As you can see, even after one epoch, our weights have been adjusted to compress the data to just 2 neurons and extrapolate that back out to resemble the original image. Given more time and less restrictions, this would be repeated for many epochs and the results will significantly increase, as reported in the next section in which we train a model on Perlin Noise heightmaps.

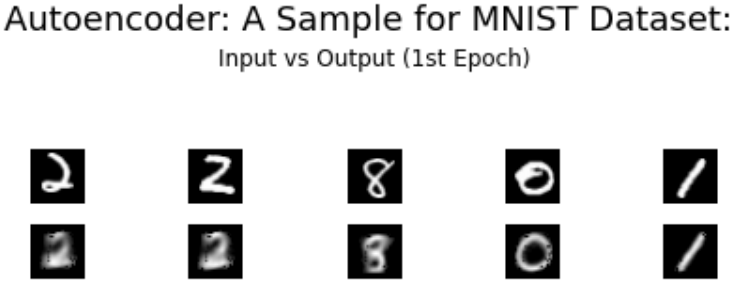


Figure 9: Sample of MNIST Dataset

4.3 Perlin Noise Heightmap Data Training

Referring to our objectives set out at the beginning of the project, our intention is to be able to generate new realistic terrain from a small (2D) latent space. Since we have had issues with correctly accumulating the LiDAR scans after the PCA dimensionality reduction, we decided to utilise our Perlin Noise code to prove the effectiveness of the model. Attached is a figure proving just how efficient our autoencoder is.

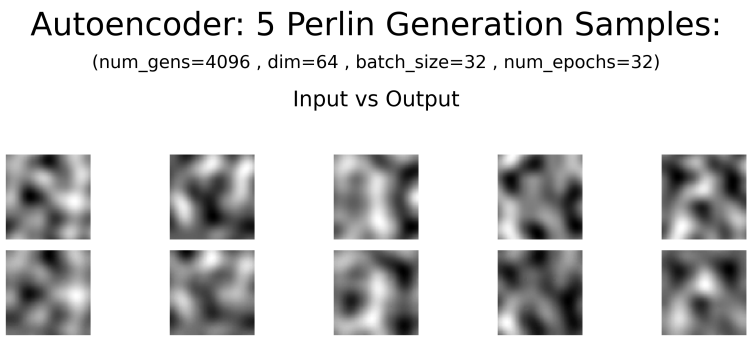


Figure 10: Autoencdoer generated terrain using Perlin

In our prime example above, we generated 4096 Perlin Noise images, each of dimension 64 pixels by 64 pixels; these were batched into 32s. The model was modified with an additional layer, since we have more pixels, and the bottleneck was changed to 8 neurons, rather than the 2 previously. Run this training for 32 epochs and the outputs tend increasingly towards the inputs with impeccable precision. With more time and processing power, we could increase these parameters for even better effect. In addition, colour could be returned to the images, as in the Perlin Noise section of the report, for a more aesthetically pleasing colourmap. We unfortunately didn't have time to use 2 central neurons to generate new plots, but with even a slight increase in time we could have modified the code for this purpose.

4.4 Variational Autoencoders

Before concluding the report, it's worth briefly discussing variational autoencoders (VAEs) [14]. This is the name for the type of network referred to when writing about mapping from the latent space, used in unsupervised learning, semi-supervised learning and supervised learning. We would make this modification to our model to be able to generate those new images, given extra time.

Basically, the model receives the input (i.e. our LiDAR scan/s) and compresses it into the latent space. It is here that a Gaussian distribution represents datapoints and we can sample for new generative terrain. The

point is to maximise the likelihood of the data given a mixture of Normal distributions for it.

References

[1] Code: https://github.com/ewanmwalker/Terrain-Generation/tree/main

[2] Midpoint Displacement information https://bitesofcode.wordpress.com/2016/12/23/landscape-generation-using-midpoint-displacement/

[3] Savitzky-Golay Filter https://www.mathworks.com/help/ signal/ref/sgolayfilt.html

[4] Cellular Automata Conway https://en.wikipedia.org/wiki/Conway's $_{G}ame_{o}f_{L}ife$

[5] Worley noise Worley noise - Wikipedia

[6] Diamond Square Theory https://en.wikipedia.org/wiki/Diamond-square_algorithm

[7] Ken Perlin's Paper on Perlin Noise [*An Image Synthesiser – ACM SIGGRAOH Computer Graphics, 19(3):287-296, 1985*]

[8] Principal Component Analysis Theory https://www.inf.ed.ac.uk/teaching/courses/iaml/2011/s

[9] Principal Components Graph https://link.springer.com/chapter/10.1007/978-3-031-04420-5_{4}

[10] Autoencoder diagram: https://medium.com/@visheshtaposthali/exploring-autoencoders-understanding-the-math-and-implementing-with-code-6ce80db835a0

[11] Autoencoder Theory https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

[12] MNIST dataset https://www.kaggle.com/datasets/hojjatk/mnist-dataset/

[13] ReLU information and graph https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html

[14] VAE https://en.wikipedia.org/wiki/Variational$_{a}utoencoder$