7CCSMPRJ Final Year Project
Generative Design in Minecraft - WGAN Road Generation

Final Project Report

Author: Preslav Kisyov

Supervisor: Dr. Thomas Thompson

Programme Title: Artificial Intelligence (MSc)

Email: preslav.kisyov@kcl.ac.uk

Student ID: 1726137

Word Count: 13 153

August 22, 2021

Abstract

This project produces an Artificial Intelligence program that generates a road system and improves a settlement on an unknown Minecraft map.

The final goal of the project is to enhance already existing filters by changing and upgrading their road-network system infrastructure, and thus, improving the overall quality of the generated settlement. The chosen approach uses Computer Vision and Deep Learning algorithms and methods to tackle the aforementioned problem, resulting in a lifelike settlement.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 15,000 words.

Preslav Kisyov

August 22, 2021

Acknowledgements

I would like to acknowledge Dr. Thomas Thompson of the Faculty of Natural and Mathematical Sciences at King's College London for all the support, help, supervision, and guidance that he has given to me throughout the project. I would also like to thank Dr. Thompson for assigning this new and innovative task to me and for believing in my personal skills and abilities to solve the problem at hand.

Contents

List of Figures **3**

Nomenclature **6**

1 Introduction **7**
1.1 Background and Motivation 8
1.2 Report Structure 12

2 Literature Review **13**
2.1 GDMC Competition Methods 13
2.1.1 Bottom-Up Approach 13
2.1.2 Top-Down Approach 15
2.2 Neural Networks 18
2.3 Convolutional Neural Networks 21
2.4 Generative Adversarial Networks 23
2.4.1 Wasserstein GAN 25
2.5 Image Processing in Computer Vision 27
2.5.1 Image Scaling 27
2.5.2 Dilation & Erosion 28
2.5.3 Finding Disconnected Components 29

3 WGAN Road Generation Filter **31**
3.1 Aims & Objectives 31
3.2 Technologies 32
3.3 WGAN - Methodology & Design 32
3.3.1 WGAN's Architecture 33
3.3.2 Preprocessing & Training 36
3.4 Filter - Methodology & Design 38
3.4.1 Road Generation 39
3.4.2 Building the Road 41
3.4.3 Building Houses 46

4 Evaluation & Results **47**
4.1 WGAN Evaluation 47
4.2 Filter Evaluation 49
4.2.1 Road Evaluation 49

4.2.2 Dealing with Obstacles . . . 52
4.2.3 Settlement Evaluation . . . 57
4.3 Runtime Comparison . . . 61

5 Conclusion & Future Work **63**
5.1 Overall Conclusion . . . 63
5.2 Future Work . . . 64

6 Legal, Social, Ethical and Professional Issues **66**
6.1 Plagiarism & Originality . . . 66
6.2 Risks & Software Competence . . . 67
6.3 AI Concerning Ethics . . . 67

References **72**

A Additional Proof Samples **73**
A.1 Road Generation . . . 73
A.2 Settlement Generation . . . 75
A.3 Filter Limitations . . . 77

B User Guide **78**
B.1 MCEdit Download Instructions . . . 78
B.2 Project & MCEdit Installation Instructions . . . 78
B.3 User Instructions . . . 80
B.3.1 Running finalwgan.py . . . 80
B.3.2 Running convert_model.py . . . 81
B.3.3 Running convert_images.py . . . 82
B.3.4 Running convert_images - Conditional.py . . . 82
B.3.5 Running MCEdit . . . 83
B.3.6 Running Filters . . . 83
B.3.7 Reconstructing Results . . . 86

C Source Code **87**
C.1 Originality Avowal . . . 87
C.2 Project Structure . . . 88
C.3 convert_images - Conditional.py . . . 89
C.4 convert_images.py . . . 91
C.5 convert_model.py . . . 92
C.6 finalwgan.py . . . 94
C.7 RoadFilter_WGAN.py . . . 101

List of Figures

1.1 Banished Game Village Generation 8
1.2 GDMC AI-Generated Settlement 8
1.3 Differences Between PGT, EA, and SBPCG 9
1.4 Different Methods for Terrain Generation 10
1.5 Generated Doom Levels 11
1.6 Generated Minecraft Maps 11

2.1 The A-Star Algorithm Process 14
2.2 The Filip Skwarski Filter Submission 15
2.3 Citygen's Roads Graph 16
2.4 Citygen Road Growth 17
2.5 Citygen City Areas 17
2.6 Gradient Descent 19
2.7 Single Neuron 19
2.8 Activation Functions 20
2.9 CNN Layer 22
2.10 The GAN Process 24
2.11 GAN Generated Images 25
2.12 Morphological Processing Operations 29
2.13 Connected Components Labelling 30

3.1 Comparison of Training Image Sizes 33
3.2 Generator's Architecture 34
3.3 Critic's Architecture 35
3.4 Training Loss History 37
3.5 Training Generated Results 38

3.6 Filter Design . . . 39
3.7 Image Generation Submodule . . . 40
3.8 Image Enhancement Examples . . . 41
3.9 Tree Removal & Generation . . . 43
3.10 Generated Tunnel . . . 44
3.11 Connecting Roads . . . 45

4.1 Enhanced Edge Generation Examples . . . 48
4.2 Road Examples . . . 50
4.3 Road Examples . . . 51
4.4 Tree Removal Examples . . . 53
4.5 David Mason Water Obstacle Example . . . 54
4.6 Water Obstacle Example . . . 55
4.7 Bridge Problem Example . . . 55
4.8 Tunnel Digging Examples . . . 56
4.9 David Mason Mountain Obstacles . . . 57
4.10 David Mason Generated Settlement . . . 58
4.11 Generated Settlement . . . 60
4.12 Selected Region . . . 61

5.1 Conditional WGAN Results . . . 64

A.1 Flat Road Generation . . . 73
A.2 Uneven Surface Road Generation . . . 74
A.3 Combined Filter Settlement Generation Example 1 . . . 75
A.4 Combined Filter Settlement Generation Example 2 . . . 76
A.5 Two Filter Generated Settlements . . . 77

B.1 Activated Virtual Environment . . . 79
B.2 Installation Commands . . . 79
B.3 Model Directory Path Code . . . 81
B.4 Model Conversion Code Example . . . 82
B.5 Conditional Dataset Conversion Code Example . . . 82
B.6 Combined Filter Code Changes . . . 84
B.7 Filter Usage Guide . . . 85

B.8 MCEdit Open Menu 86

Nomenclature

$2D \,\&\, 3D$ Two- and Three-Dimensional

AI Artificial Intelligence

CNN Convolutional Neural Network

DNN Dense Neural Network

EA Evolutionary Algorithms, Subset of Evolutionary Competition

GAN Generative Adversarial Network, a type of Neural Network

$GDMC$ The Generative Design in Minecraft Competition

GPU Graphics Processing Unit

IDE Integrated Development Environment

$MNIST$ Modified National Institute of Standards and Technology Database

NN Neural Network

PGT Procedural Terrain Generation Method

$SBPCG$ Search-based Procedural Content Generation Method

TFD Toronto Face Dataset

$WGAN$ Wasserstein Generative Adversarial Network

Chapter 1

Introduction

Artificial Intelligence in games is a relatively new but well-known research area of AI. It originates from early 1951/1952 when it was first used, as a concept, in the mathematical game Nim [49]. From then on, AI in games has been improving rapidly, becoming more lifelike and, at the same time, better than human players. Nowadays, it is used in almost every game, where players can even interact with the program and influence it. An example of such functionality is the groundbreaking Action Selection concept, where interactions like call or help have been made possible [2].

Therefore, this project creates such an Artificial Intelligence program that is to be used in Minecraft [48]. More specifically, a filter that can modify a selected region of the game's terrain and generate a lifelike settlement on it. The system takes into consideration its surroundings, making it adaptable to the given environment. Programs of this type fall under the Procedural Content Generation method [50]. It is an algorithmic approach to creating new data, usually used in computer graphics. An instance of a game using Procedural Content Generation [50] to create its whole terrain map, and village names, is Banished, by Shining Rock Software [9]. An example of what the game generates can be observed in figure 1.1.

Figure 1.1: This figure shows a generated village and terrain in the game Banished [9]. The showed menus are not generated, thus, they are not to be considered by the reader.

Furthermore, the paper explores and takes inspiration from other famous approaches that are part of the annual GDMC [26] competition. This competition asks participants to create an automated program that can build a lifelike settlement in Minecraft [48], similarly to what this paper describes. Such approaches, as well as the method used for this project, are covered more extensively in future chapters.

1.1 Background and Motivation

This paper, as stated above, describes an AI program to be used in Minecraft [48] that generates a lifelike settlement given a selected environment. In addition, it produces a functional filter that should be submission-ready for the previously-stated GDMC [26] competition, either individually or as a part of another filter. An example of the results of a similar filter in Minecraft [48] can be seen in figure 1.2.

Figure 1.2: A submission to the GDMC Competition. The image shows an AI-generated settlement in Minecraft [27].

Procedural Content Generation [50] can create new levels, environmental maps, objects (i.e., buildings and weapons), characters, etc. Biologically-Inspired Computing [45] approaches are a famous way for generating the aforementioned entities. These are methods, most commonly known as Evolutionary Algorithms [53], that are built to resemble real-life biology models.

An example of such algorithms being used in games can be seen in the paper "A Survey of Procedural Terrain Generation Techniques using Evolutionary Algorithms" [30]. It covers different Evolutionary Algorithms [53] for Procedural Content Generation [50] in video games. Moreover, they identify three main groups of content generation that can be observed below:

- Procedural Terrain Generation (PGT)
- Evolutionary Algorithms (EA)
- Search-based Procedural Content Generation(SBPCG)

All of these methods share a relationship with each other. That relationship can be seen on graph 1.3. Furthermore, generated terrain samples can be observed in figure 1.4. This project has taken into consideration such methods when creating the filter, however, because of their unnecessary complexity, Artificial Neural Networks have been chosen as the main subject of this paper. Therefore, Evolutionary Algorithms are not further covered, as they are a whole separate field of AI [53].

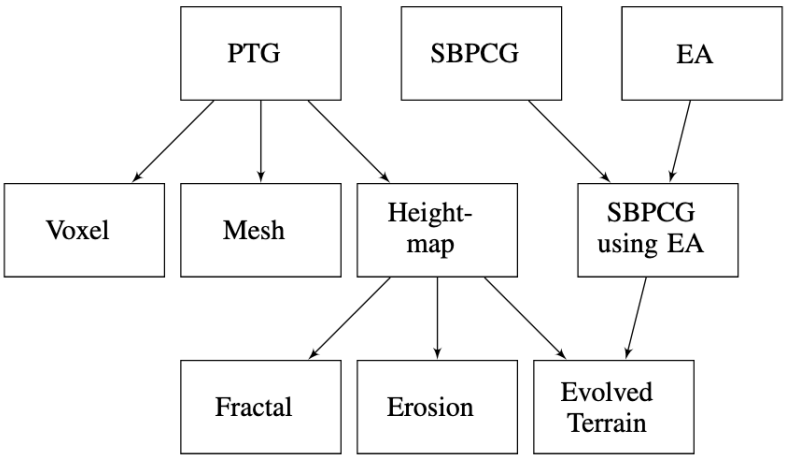
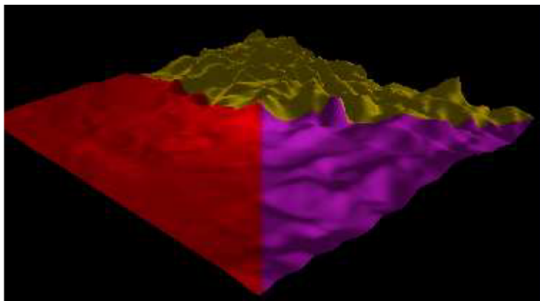


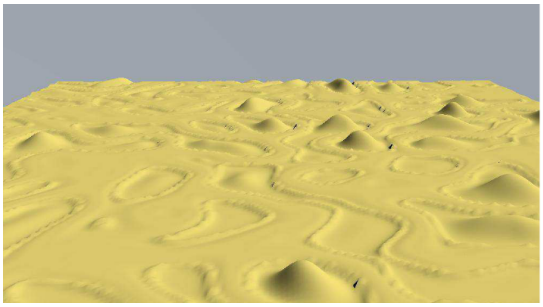
Figure 1.3: The relationship between the three different fields - (PGT), (EA), and (SBPCG). The graph shows different methods that fall under one or more of the three categories. Further details on these methods can be found in the referenced paper [30].



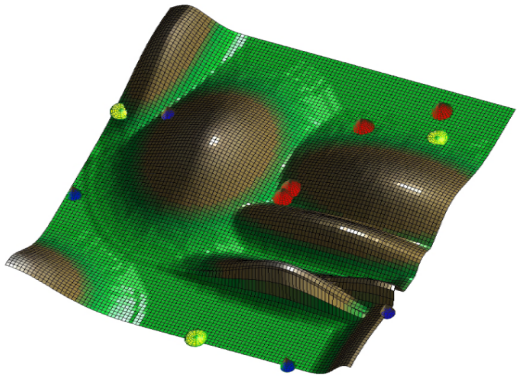
(a) Ong et al. and Saunders

(b) Walsh and Gade

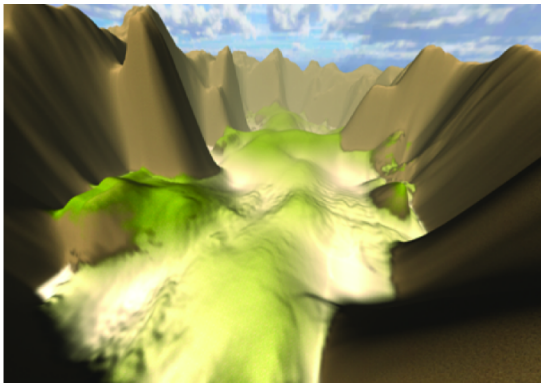
(c) Frade et al.



(d) Frade et al.



(e) Togelius et al.



(f) Raffe et al.

Figure 1.4: The images above show the generation and/or improvement of digital terrains done by various methods (see each respective image). More information on each method can be found in the cited paper [30].

This paper, on the contrary, uses Generative Adversarial Networks (GANs) [17] to generate the road system of the settlement. GANs [17] have been primarily used to generate game levels, maps, weapons, and characters. A prime example of that can be seen in figure 1.5. GANs can also generate 3D content, like Minecraft [48] maps. A famous such approach is GANCraft [18], which utilises both GANs and Convolutional Neural Networks (CNNs) [28] to render a believable image, seen in figure 1.6. All of these methods are extensively covered and explained in the next chapter.

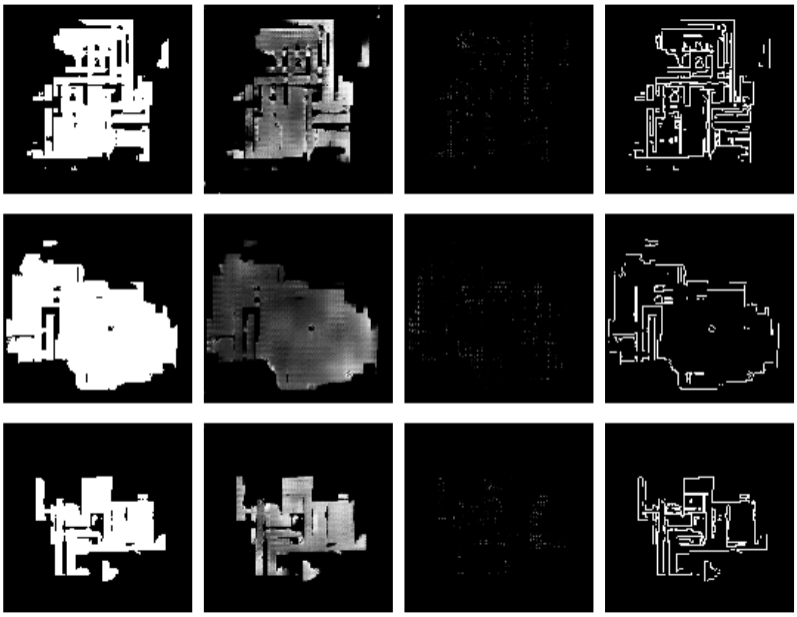


Figure 1.5: Generated levels for the game Doom. A Generative Adversarial Network has generated the images above by being trained on real-level maps. The first two columns show the Floor and Height maps, and the last two show the items and wall maps respectively [16].

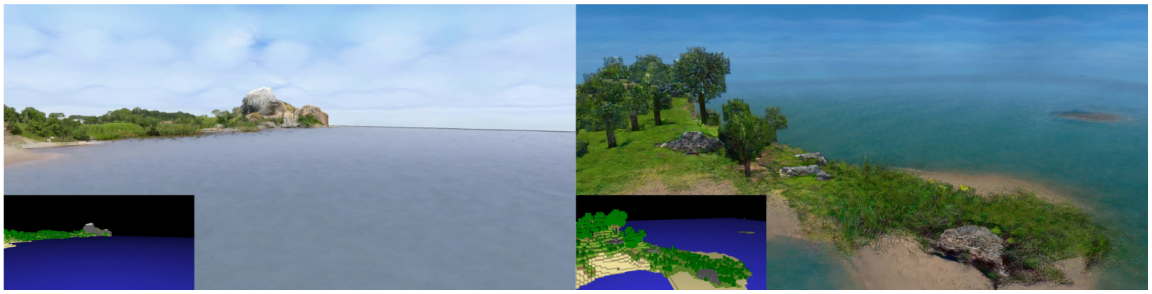


Figure 1.6: Minecraft generated maps, where the real environment images are the samples and the images below are the generated Minecraft maps [18].

The approach taken in this paper trains such networks on real-world road images and generates a new road system that can be later applied to Minecraft [48]. It takes into consideration different environments, obstacles, and terrain, to produce a believable settlement. Overall, it aims to improve upon previous state-of-the-art GDMC [26] submissions. This leads to an overall filter improvement and opens up various possibilities for further research and improvements. The method itself, as well as the technologies used, are described in future chapters. The structure of these chapters can be observed in the next section.

1.2 Report Structure

- Chapter 1:
  - This chapter provides an introduction to AI in games and briefly covers different methods used for content generation.
- Chapter 2:
  - In this chapter, a more detailed description of the methods used in this project is presented. In addition, it dives into a more technical evaluation between the different methods and compares them.
- Chapter 3:
  - The following chapter describes how the proposed Minecraft [48] filter was created. This includes, but is not limited to, how the AI model was trained and how the generated road system was applied to the game.
- Chapter 4:
  - This chapter goes over the evaluation results of both the AI model and the filter itself. It also compares the filter with other approaches.
- Chapter 5:
  - The chapter provides its readers with an overall summary and conclusion of the project described in this paper. Moreover, it reflects on any possible future research works that can be made.
- Chapter 6:
  - The following chapter covers pressing legal, professional, and ethical issues that may or may not arise due to this project.

Chapter 2

Literature Review

The following chapter will go into a detailed analysis of the different technologies used, or explored, in the making of this project. In addition, it will go over any advantages and disadvantages of the different methods, and why they were or were not chosen for this project.

2.1 GDMC Competition Methods

The section focuses on the different approaches taken by various submissions to the GDMC [26] competition. This paper has been highly influenced by the proposed methods and they are the solemn inspiration behind the idea of upgrading the road systems of settlements.

2.1.1 Bottom-Up Approach

With this type of approach, the filter AI program places the buildings first, and then it connects them with roads. Filip Skwarski [35], the winner of the first GDMC [26] edition, is a vivid example of that. Like many other filters, it uses the A-star Algorithm [51] to connect every building. However, to be able to truly evaluate this approach, one must understand what the A-star Algorithm [51] does. Moreover, this very project utilises its search completeness, which will be covered in future chapters.

A-Star

This algorithm is one of the classical and most famous methods of connecting two points in space. It is a complete graph-traversal search method, hence, it will always find a solution if there is one [51]. The idea behind it is to have a tree, with its root being the starting

point, expand one edge at a time until the endpoint is found. There might be many solutions, therefore, the tree with the least distance travelled, the least cost path, is taken. A visual representation can be observed in figure 2.1.

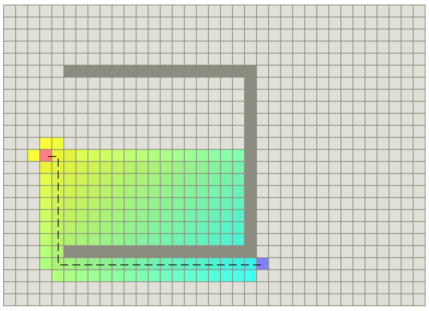


Figure 2.1: A graphical representation of how the A-Star Algorithm works. The image shows a starting point (in red) and a goal point (in purple). The coloured boxes represented the searched area by the method and the dashed lines show the final path taken to the goal [29].

The algorithm uses heuristics and they need to be admissible to find the optimal solution. This means that the estimated cost must be equal or lower to the actual cost, so it never overestimates [51]. For example, the estimated cost would be the straight line from two points in space, however, that might be deceiving, as this straight-line path might not be reachable. The condition for having a heuristic admissible, or consistent, is given in equation 2.1. The h is the heuristics of a given point, and d is the length of edge (x, y).

$$h(x) \leq d(x, y) + h(y) \tag{2.1}$$

The A-Star Algorithm [51] needs to determine which path, part of the tree, to expand. Therefore, it tries to minimize the following equation 2.2, where $f(n)$ is the sum of the actual path cost $g(n)$ and the estimated cost $h(n)$, that was previously defined.

$$f(n) = g(n) + h(n) \tag{2.2}$$

However, as it was previously stated, the algorithm will find the optimal solution if its heuristics are admissible. Having translated that condition into Minecraft [48], it would mean that the estimated cost of connecting any two buildings needs to be lower than the actual cost. However, there are many obstacles, like water, lava, different elevation, and unreachable points, which may cause to have disconnected buildings, as the algorithm will not find either the optimal path or any path at all. This can be seen in figure 2.2.

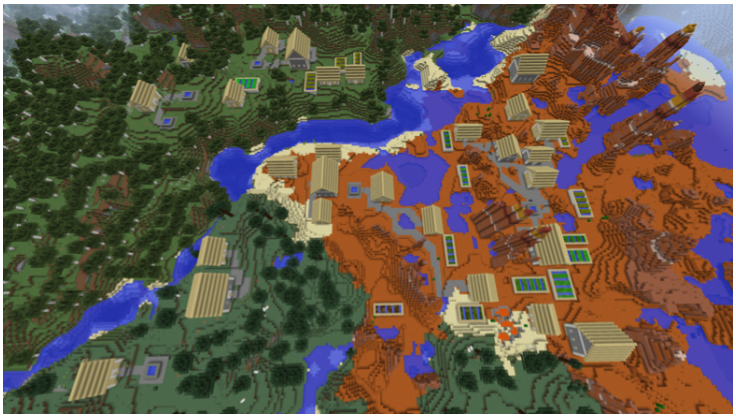


Figure 2.2: An aerial view of the Filip Skwarski submission. It can be seen that there are disconnected buildings, as for example, because there is a water body [35].

In contrast, the proposed method by this paper produces a more flexible and adjustable, to the terrain, road. It creates the road first, thus, it can be adapted to the environment and it is not building dependent. Such a similar approach is covered in the following subsection.

2.1.2 Top-Down Approach

On the contrary to the previously-covered approach, this one builds the road first. However, this particular method lacks autonomy. This means that the user or AI needs to decide where that road should be placed, and then, put the buildings on the road itself. That would make the generation of the road vital for the overall quality of the filter, as it needs to take into consideration house slots, as well as obstacles, and the elevation of the terrain. All of these requirements make it difficult for a classical algorithm to generate such a road system unless the user specifies the parameters after observing the environment. A famously proposed approach is Citygen [20], which uses a method applied to Procedural City Generation.

Citygen

This method, as stated before, is an iterative ranked Top-Down graph model. It has three layers of stages that are executed sequentially, as per [20]. These processes can be observed below:

1. Primary Road Generation
2. Secondary Road Generation
3. Building Generation

The first stage allows the user to define control nodes, vertices of the Primary Road graph. By doing so, the primary road can be adjusted to the environment and act as a starting point of the whole road system and settlement generation. This is the most crucial step, as this main road serves as the backbone of any future-generated city or settlement [20]. The Primary Road graph is comprised of two separate graph nodes:

1. High Level Graph
2. Low Level Graph

The high-level graph stores the topology of the primary road. This includes the control points and the edges between them, indicating that they have been connected. On the contrary, the low-level graph stores the actual path taken by each road, called adaptive roads [20]. Both graphs use an Adjacency List, which is a list of entries for each node in a graph, and each entry has a list of the nodes directly connected to it [39]. Having both graphs utilising the Adjacency List Data Structure, and being separated from one another, makes the extraction of information more efficient than storing everything in one graph [20].

The second stage is automatically executed after the user has defined the main road. However, it lacks adaptability because it generates a secondary road grid from already existing presets. Furthermore, roads are adapted to their environment and terrain by a process called Sampling [20]. The system starts plotting edges on an undirected graph simultaneously from both source and destination node towards one another. The Sampling process finishes once a sample is within a defined constant distance from the destination point, taking into consideration a possible deviation angle of the road [20]. Both primary and secondary roads can be seen in figure 2.3.

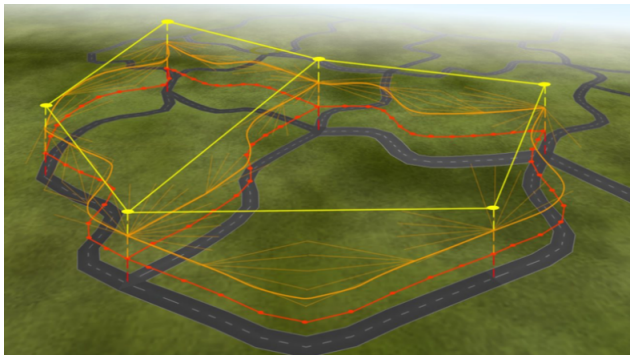


Figure 2.3: The main road network, where the yellow graph represents the high-level topology, and the red graph shows the low-level graph. The orange graph shows the plotted samples [20].

The generated secondary roads are connected with the primary road, creating loops. These

loops define the city cells, which are populated with more secondary roads. Such an image showing the road growth in a city cell can be observed in figure 2.4.

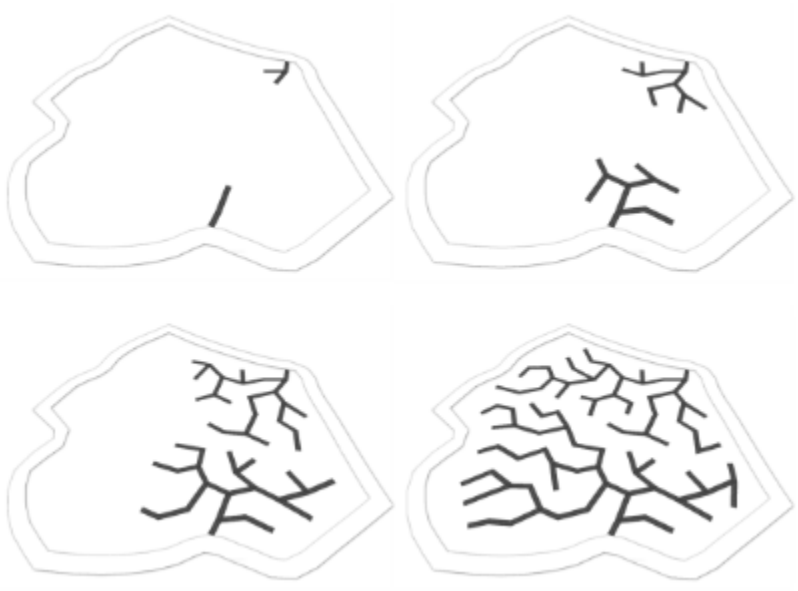


Figure 2.4: The road growth within a cell for 10, 100, 300, and 1000 iterations [20].

The final stage of Citygen [20] is Building Generation. Buildings are created with preset geometry depending on whether the plot that they are first put on is part of a suburban, industrial, or downtown area. Depending on that, buildings are also put more or less closer together. Different city areas can be seen in figure 2.5.

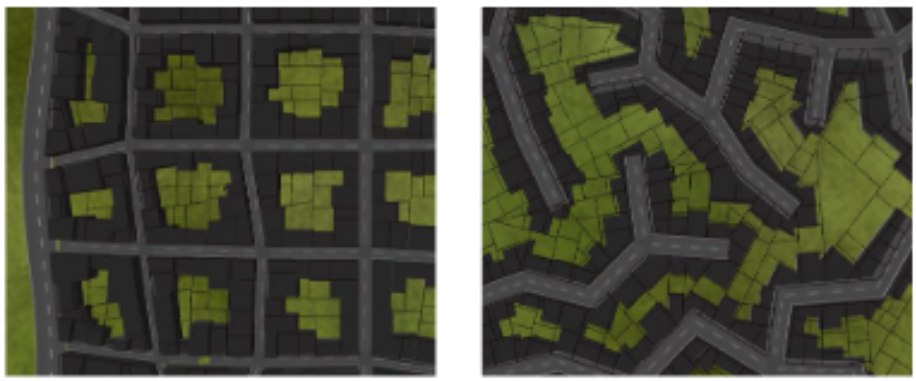


Figure 2.5: The left image shows a downtown area and the right image is a suburban area [20].

To conclude, Citygen [20] is a powerful road generation tool, however, it is very parameter-dependant. Thus, it cannot be fully utilised as an automated AI program and applied to Minecraft [48], unless constant user supervision is present. Nonetheless, the proposed project takes advantage of the steps that Citygen [20] undertakes for the generation of roads.

David Mason's Approach

This project's filter uses and is compared against David Mason's [34] approach from the 2020 GDMC [26] competition. Therefore, it must be covered in more detail to fully understand why it was picked as a benchmark for this paper.

The main take from the filter is that it focuses mainly on building diversity and building mines. The houses built by the approach have their aesthetics adjusted to the environment [34]. This includes the exterior and interior of the houses. In addition, the filter builds a mine that is unusual from other GDMC [26] submissions.

Although David Mason's filter has been deemed a standout approach and was highly ranked by the judges [34], it lacks lifelike properties. The buildings can be blown out of proportion, being too unreal or big [34]. Moreover, the created pit mine could be too big for a human to actually dig [34].

Therefore, the project described in this paper tries to improve on these drawbacks. In future chapters it can be seen that the road system is highly neglected by the approach, leaving room for improving an already highly ranked filter. Additionally, David Mason's approach [34] mainly focuses on the buildings, where, on the other hand, the filter described in this paper focuses on the road system. Thus, in theory, by combining both filters, the result should be a vastly improved settlement.

2.2 Neural Networks

Neural Networks are a vital part of this project. Before diving into the different types of NN, one must know the general idea of how these systems work. First and foremost, Artificial Neural Networks are very flexible non-linear regression discriminant models [36]. They are used for tasks like data analysis or in robotics, mainly to find patterns in data. Similar to the human brain, these networks also have a number of neurons, as well as a number of layers of neurons. In a Feed-Forward Fully Connected Neural Network, also known as a Dense Neural Network, every neuron in a layer is connected with every neuron in the next and preceding layers in a forward fashion [44]. These connections are called weights, which are trained or adjusted, by an algorithm, like Gradient Descent [32]. This is an optimization algorithm that uses gradients to find a local minimum, going down from a slope. This can be observed in image 2.6.

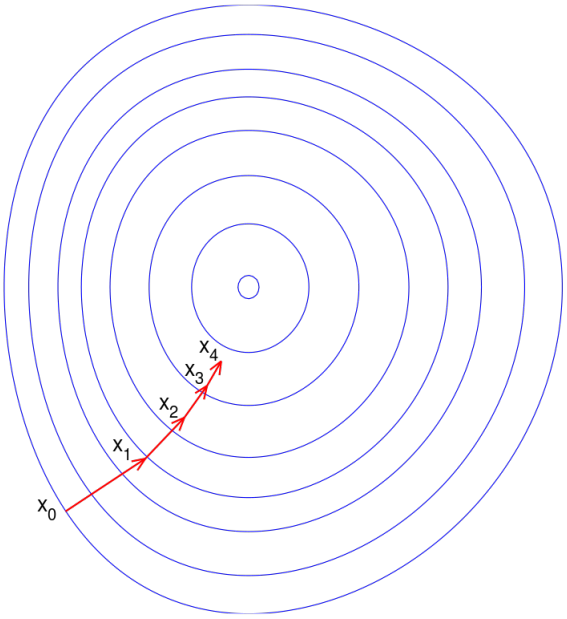


Figure 2.6: Gradient Descent on a series of different level sets, where the function takes a certain constant. Each x represents a different level or iteration of the algorithm, converging towards the minimum [47].

A typical neuron maps an input, given its weights, to an output with the help of an activation function. These non-linear functions can be with any shape, and it defines the output of that particular node. Hence, the architecture of a two-layered network with one neuron would consist of one input layer and one output layer with that single neuron. This can be observed in figure 2.7.

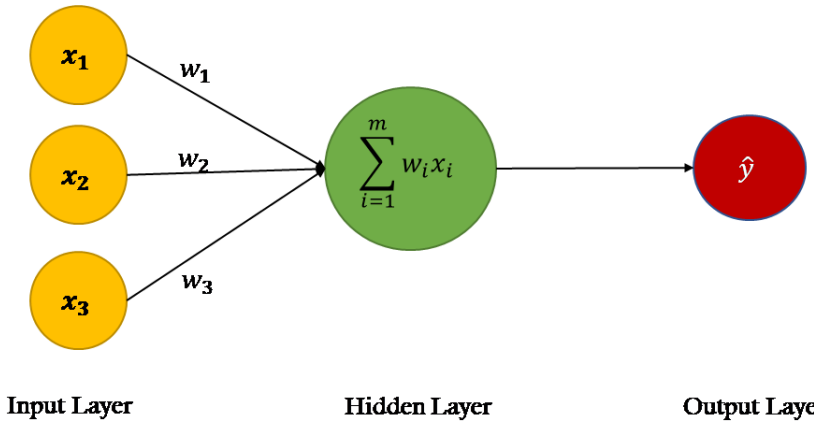


Figure 2.7: The architecture of a simple one neuron, two-layered NN [33]. The x's are the inputs, the w's are the weights. The green circle is the neuron as the sum of the product of all weights and inputs. And ŷ is the output of the neuron's activation function. There is a bias of 1, which is denoted as x0 and w0. Note, that the Hidden and Output layers in the image represent the same neuron, hence the network would be considered as Input-Output NN.

The function of a single neuron can be represented by equation 2.3, where ϕ is the selected activation function that transfers the result to the output layer.

$$\hat{y} = \phi(\sum_{i=1}^{m} w_i x_i) \tag{2.3}$$

The type of activation function may differ from layer to layer depending on what the NN's task is. A typical NN architecture consists of an Input Layer, a Hidden Layer, and an Output Layer. The Hidden Layer can consist of many layers that do not have access to the outer layers. The data comes from the Input Layer, passes through all hidden layers, and a result goes out through the Output Layer [44]. An example of such functions that are typically used in the Hidden and Output layers can be seen in image 2.8.

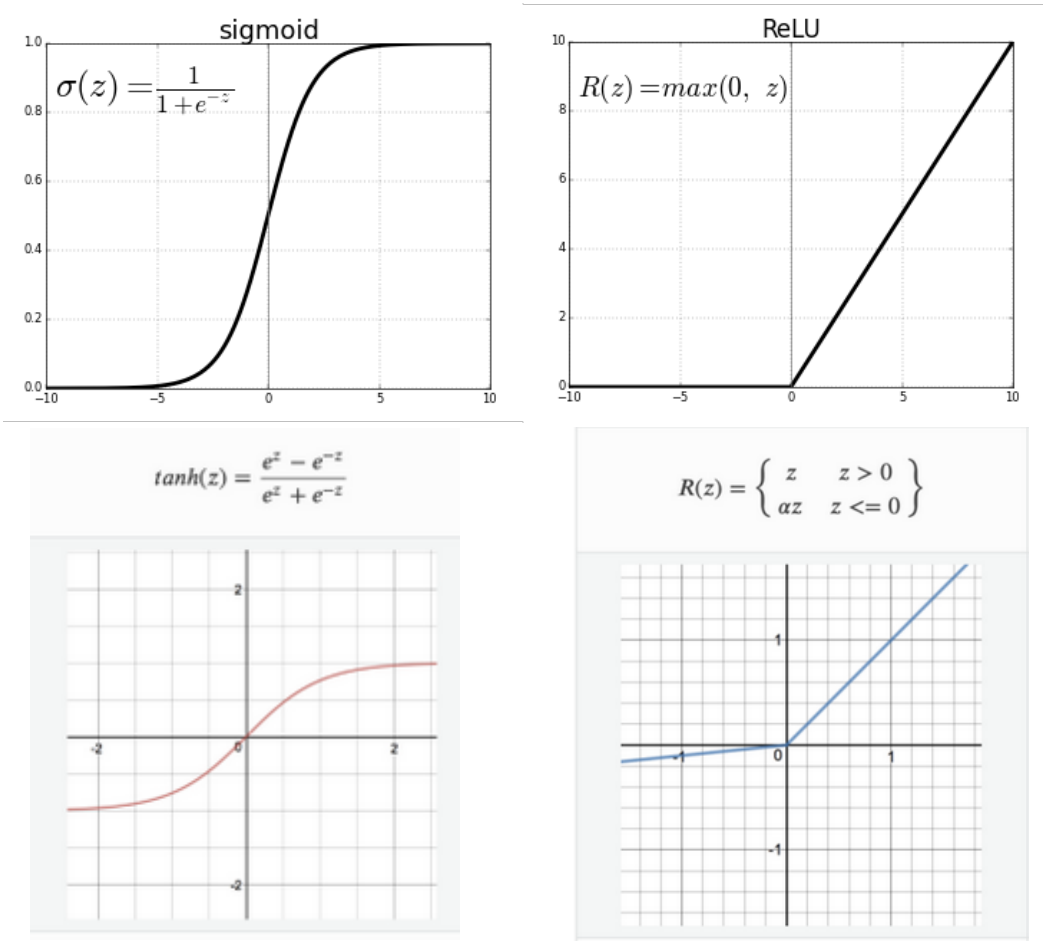


Figure 2.8: These are some of the most used activation functions in NN. The first one is the Sigmoid function, mainly used in the output layers for binary classification. The second one is the ReLU function, which is used in the hidden layer. It has a disadvantage that makes all negative values into a zero, making training the model harder. However, it is faster than the Sigmoid or Tanh functions. The Tanh function varies between -1 and +1, making it a mathematically shifted Sigmoid. Lastly, the LeakyReLU is a function that tries to expand the range of the limited ReLU function [37].

Furthermore, the training of any Neural Network is done via adjusting its weights [36]. This can be done with the process of Backpropagation, which computes derivatives in a backward fashion [5]. After a forward pass of the data, the weights of the network are updated going from the output to the input neurons. In that way, the network would learn from experience [5]. The update rule can be seen in equation 2.4. θ represents the weight at step t. In addition, α is a user-specified learning rate and X is the set of input-output pairs. E, on the other hand, is an error that is to be minimised. Any error can be used, typically the Mean Squared Error, which can be observed in equation 2.5.

$$\theta^{t+1} = \theta^t - \alpha \frac{\partial E(X, \theta^t)}{\partial \theta} \tag{2.4}$$

$$E(X, \theta) = \frac{1}{2N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2 \tag{2.5}$$

To elaborate, the N in equation 2.5 is the size of the set X, y_i is the target value for the input-output pair i, and $\hat{y}_i$ is the output of the network for the same input in pair i. For more detailed explanation of how Backpropagation works, the source [5] can be referenced.

In conclusion, Neural Networks are extremely powerful tools to find patterns in data. However, for the task of this project, different architectures need to be explored. Therefore, the next section covers Convolutional Neural Networks [28].

2.3 Convolutional Neural Networks

CNNs are the most used Neural Networks when it comes to image processing. They are the building blocks of GANs [17]. In addition, CNNs are usually followed by the previously-mentioned Dense Neural Networks [44] to make predictions.

The architecture of a Convolutional Neural Network [28] differs from the architecture of a normal NN. There are no typical neurons, instead, there are filters. Each layer consists of many learnable two-dimensional kernels. The depth comes from the number of filters in a single layer. These kernels will fire if they see a specific feature at a certain position, called activation [28]. A visual representation of a CNN layer can be seen in figure 2.9.

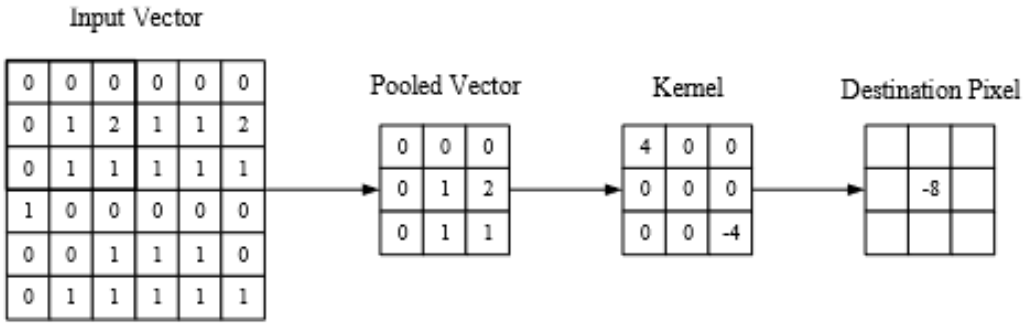


Figure 2.9: A Convolutional Layer that pools a certain part of the input vector (an image) and applies a filter with a specific kernel, resulting in an activation [28]. Usually, that process is done for the whole of the input vector, and for each filter, resulting in an activation map.

The pooled vector's complexity in 2.9 depends on whether the Convolutional Layer has implemented Padding or Striding. A Convolutional Layer usually can have no padding, keeping the input the same, or zero padding, which is adding 0's to the borders of the input vector. Applying zero-padding would result in the same size output as input [28]. On the other hand, the strides control how overlapped the kernels are with the input vector. If we define a stride of one, that would mean that we consider every cell in the input. However, if stride of two is defined, the layer would consider every other cell position, reducing dimensionality [28]. A simple formula for calculating the size of a Convolutional Layer output can be seen below, in equation 2.6. V is the size of the input (height*width*depth), R is the size of the filters, Z is the amount of padding, and S is the amount of striding applied [28]. This equation is very useful for keeping track of output sizes, as Convolutional Layers are extremely computationally heavy.

$$\frac{(V-R)+2Z}{S+1} \tag{2.6}$$

A CNN can have another type of layers as well [28]. One such layer is the Pooling Layer. Its aim is to gradually reduce the dimensionality of the vector, thus, reducing the model's complexity. These layers are also filter matrices that are applied to the vector [28]. The most famous approach is to use Max Pooling Layers, but there are other types like Average Pooling layers and General Pooling layers. Max Pooling would take the maximum value of the pooled vector, therefore, these layers are typically quite small (e.g., 2-by-2) [28]. These layers are usually used either in the beginning or at the end, before passing the representation to the Dense NN [44] [28].

Because of its complexity, CNNs are prone to overfitting. Thus, layers like Dropout and Batchnormalization can be used [14]. Dropout disables a certain percentage of neurons every

pass and can be used in both CNNs and Dense NNs [14]. On the contrary, Batchnormalization normalizes the output data of a previous layer, making learning more independent [14].

Other ways, that are also utilised by this paper, are weight clipping and scaling [6]. These methods deal with the problem of exploding gradients, as the networks use gradients when being trained. Essentially, the weights would become too big or small, rendering the model useless [6]. What clipping does is restricting the weights' values in a certain range, so if the weight becomes too large or small it will be clipped to a predefined value. In comparison, scaling would normalize all weights to a certain norm [6].

To conclude, Convolutional Neural Networks [28] are the preferred models for when it comes to processing images and identifying objects in them. However, they are also very computationally expensive. Therefore, they are very difficult to train.

As previously said, this project utilises Convolutional Neural Networks [28] and Dense Neural Networks [44] to generate a new road system. To do that, a Generative Adversarial Network [17] is required. Such networks are covered in the next section.

2.4 Generative Adversarial Networks

A Generative Adversarial Networks [17] are the building blocks of the proposed project. Thus, it is essential to understand their architecture and explore different variations that can help solve the problem at hand.

The main idea behind a GAN's architecture is the MiniMax game theory [13]. The network is comprised of two different Neural Networks that try to compete against each other. These two networks are made of a combination of Convolutional and Dense Layers.

The first model is called the Generator [17]. It is a combination of Convolutional Layers that try to generate an image given some initial noise, which has the same size as an image from the real dataset. The Generator's goal is to generate as close to the real dataset image as possible. Thus, it tries to fool the other network that is part of the overall GAN architecture, the Discriminator [17].

The Discriminator's part is to classify images as *real* or *fake* [17]. If it classifies a generated image as *real*, the Generator has produced good-enough image. This particular model is designed with a combination of Convolutional and Dense layers [17]. It is essentially a binary classification CNN [28], utilising the previously-covered Sigmoid function [37]. The overall structure of a GAN [17] can be observed in figure 2.10.

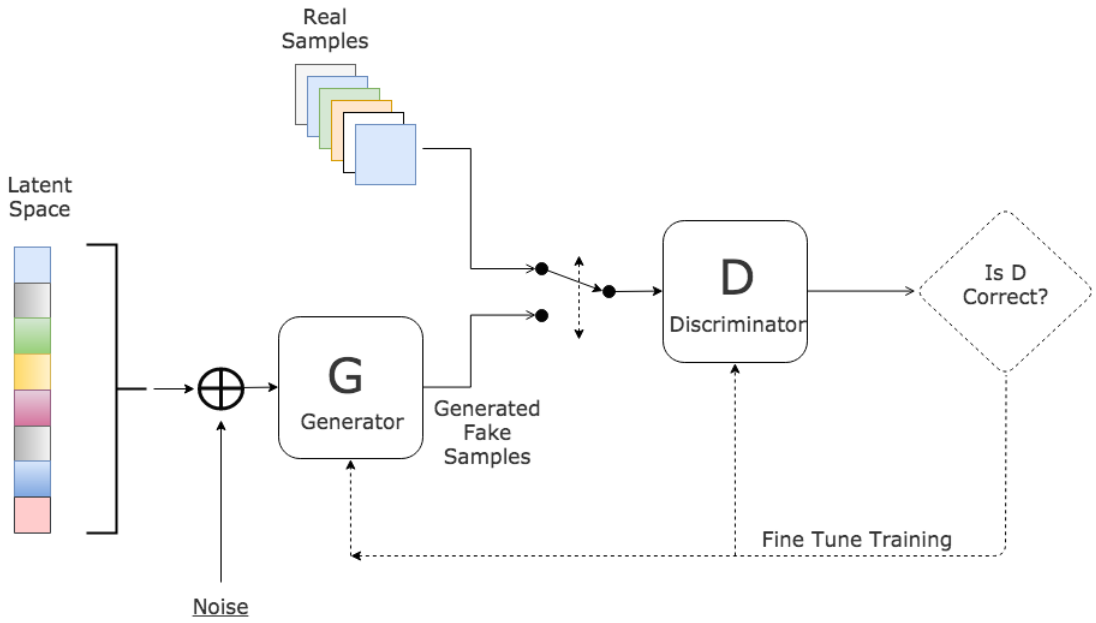


Figure 2.10: The process of the Generator trying to fool the Discriminator by generating a plausible image [1]. The Discriminator receives, in turn, either a real or a fake image and tries to distinguish them. On the other hand, the Generator tries to adjust its weights, so the newly generated image is real enough to fool the Discriminator [1]. The latent space is the generated sample list size (e.g, if the latent space is 100, the Generator will generate 100 images) [1].

Similar to other types of Neural Networks, GAN's training process involves gradients [17]. However, the Discriminator ascends its stochastic gradient $\nabla\theta_d$ for k number of iterations, where k is a hyper-parameter. And the Generator descends its stochastic gradient $\nabla\theta_g$. Thus, the training process can be described as minimising, for the Generator, and maximising, for the Discriminator, the Expected Log-Likelihood [17]. The whole expectancy formula can be seen in equation 2.7, where D is the Discriminator, G is the Generator, x is a real sample, and z is a generated sample.

$$\min_G \max_D V(D,G) = E_{x\sim p_{data}(x)}[\log D(x)] + E_{z\sim p_z(z)}[\log(1 - D(G(z)))] \tag{2.7}$$

From equation 2.7, the update equations for ascending the Discriminator's and descending the Generator's stochastic gradients can be derived. These update formulas can be seen in equations 2.8 and 2.9 respectively.

$$\nabla\theta_d \frac{1}{m}\sum_{i=1}^{m}[\log D(x^{(i)}) + \log(1 - D(G(z^{(i)})))] \tag{2.8}$$

$$\nabla\theta_g \frac{1}{m}\sum_{i=1}^{m}\log(1 - D(G(z^{(i)}))) \tag{2.9}$$

In addition, pseudo-code of the whole GAN process and more details about the calculus behind it can be found in the original paper [17]. Furthermore, examples of generated images can be observed in figure 2.11.

Figure 2.11: A visualization of the a) MNIST [11], b) TFD [21], and c, d) CIFAR-10 [31] datasets. The final rightmost column shows the randomly generated images of a neighboring sample [17].

In conclusion, Generative Adversarial Networks [17] can generate images from training samples quite well. However, they are prone to overfitting, resulting in poorly generated images that still manage to fool the Discriminator. This could happen if the Discriminator is too weak or if the training process is too difficult to learn. To deal with that problem, the project proposed by this paper, uses a WGAN architecture [41]. These networks are further covered in the following subsection.

2.4.1 Wasserstein GAN

As said before, the original GAN architecture [17] suffers from a slow and unstable training process. These problems can occur because both the Generator and Discriminator update their weights independently, without cooperation [41]. In addition, the model can suffer from vanishing gradients, covered in a previous section. Another common GAN problem is Mode Collapse, which is when the Generator produces always the same output, failing to learn how to represent complex data [41]. To deal with these issues, the WGAN architecture adds two main changes compare to normal Generative Adversarial Networks [41].

First and foremost, the loss is changed with a function that tries to compute the Wasserstein Distance in equation 2.10, resulting in equation 2.11, where f_w is a K-Lipschitz [41] continuous Wasserstein Loss function. Such K-Lipschitz functions are everywhere continuously differentiable, as described in [41]. Furthermore, g_θ is the generator, and x and z are the real and fake samples respectively. In addition, the *sup* in equation 2.10 is the supremum that measures the least upper bound (maximum value), and the $\|f\|_L \leq K$ formula means that the function satisfies the K-Lipschitz continous condition, with K being a constant [41].

$$W(p_r, p_g) = \frac{1}{K} \sup_{\|f\|_L \leq K} E_{x \sim p_r}[f(x)] - E_{x \sim p_g}[f(x)] \tag{2.10}$$

$$W(p_r, p_g) = \max_{w \epsilon W} E_{x \sim p_r}[f_w(x)] - E_{z \sim p_g(z)}[f_w(g_\theta(z))] \tag{2.11}$$

In a WGAN, the Discriminator is called a Critic and it no longer tries to differentiate between real and fake samples. Instead, it learns a continuous function to help compute the Wasserstein Distance [41]. During training, the loss for the Critic should be continuously decreasing. On the other hand, the Generator's loss would be continuously increasing [41].

A problem with these networks is maintaining the loss function continuous during training [41]. Therefore, weight clipping is introduced. The concept of weight clipping has been covered in previous sections. In this particular example, the lower and upper bounds of the weights are suggested to be *-0.01* and *0.01* respectively. However, even after weight clipping, the network might not be able to enforce the K-Lipschitz constraint [41].

Additionally, WGANs can be conditional as well [8]. This means that labelled data can be used for training the model, and the network will take into consideration the given label when generating a new image. The change from original WGANs [41] is that the network concatenates the data with the label after embedding every class. An Embedding Layer is used for mapping categorical variables to a vector, that can be later concatenated with the feature vector and create a new vector [22]. However, because of the lack of labelled road data, the idea of conditional WGANs [8] has been explored as a proof of concept only.

To finalise this section, WGANs [41] are not perfect. Nonetheless, they vastly improve upon the original GAN [17]. Thus, their faster and more reliable training process makes them a perfect fit for this project. The data used, as covered in future chapters, is not too difficult to learn, however, it is small in volume. Hence, the WGAN [41] designed in this paper, manages to produce desirable results.

Furthermore, the following section goes into more detail about different Computer Vision

methods for image processing. These specific functions have been used in this project to process and clear the created road image and to prepare the data for training. Thus, it is very important to understand why these specific methods have been chosen for the previously-mentioned tasks.

2.5 Image Processing in Computer Vision

Images are comprised of pixels and channels. The amount of pixels contained in an image depends on its *height* and *depth (height*depth)*. On the other hand, channels represent an image's depth [38]. Thus, for example, an image with a shape of (50, 50, 3) means that it has 50 height and depth, and three channels. The three main types of images are:

- Grayscale - the image has one channel and its pixel values are between 0 and 255, where 0 is black and 255 is white [38].
- Binary - the image has one channel and its pixel values are either 0 or 1. The 0 represents black and the 1 is white [38].
- RGB - the image has three channels, where each channel represents the red, green, and blue colours respectively. In addition, the image's pixel values are between 0 and 255 [38].

After having a general understanding of what an image is, one can dive deeper into the concept of Image Processing in Computer Vision. Generally, Computer Vision is a scientific field that deals with how machines can gain understanding and valuable information from digital images and videos [46]. In addition, the process of image processing performs different operations on images to extract useful information from them or to reconstruct them [38]. This project, more specifically, performs image scaling, various morphological operations, and finds disconnected components, to prepare the training data and to enhance the generated images. Therefore, the following subsections explore these concepts in more detail.

2.5.1 Image Scaling

When changing the size of an image, it is inevitable to lose information. Thus, in Image Processing, scaling is used to resize an image without losing too much valuable information. To do that, this project uses Bilinear and Lanczos Interpolation to estimate the lost pixel values [15] at different stages of the filter.

Bilinear Interpolation is essentially performing linear interpolation in two directions. Thus, it is a weighted average of four neighbouring pixels [15]. Because of its low runtime and

descent results, Bilinear Interpolation [15] was chosen as the scaling algorithm for this project's training data. However, there are other types of interpolation that could produce better results but were found not suitable for processing large amounts of data. Such functions are Bicubic and Lanczos Interpolation [15]. Although, the latter is used as the interpolation method for when the project's model has to process a small amount of images.

Bicubic Interpolation is an advancement over the Bilinear Interpolation in terms of the number of neighbouring pixels considered [15]. As the name suggests, the method considers 16 grid points. However, this was found to produce worse results when performed on binary road images, as it would change the semantics of the roads by adding non-existent lines.

On the other hand, Lanczos Interpolation [15] was found to produce good results. Nonetheless, its high complexity makes it unfeasible to use for a big volume of very big images. The main idea behind this interpolation method is to map each sample of a signal to a translated copy of its kernel [15]. The kernel is the normalized *sinc(x)* function, which can be seen in equation 2.12, as per [15]. In addition, the whole interpolation formula for a sample x can be observed in equation 2.13. The α in both equations is a positive integer that determines the size of the kernel, $\lfloor x \rfloor$ is the floor function of x, and S_i are the samples for values i [15].

$$\mathrm{L(x)} = \begin{cases} 1 & \text{if x} = 0 \\ \frac{\alpha \sin(\pi x) \sin(\pi x/\alpha)}{\pi^2 x^2} & \text{if -}\alpha \leq \mathrm{x} < \alpha \text{ and x} \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.12}$$

$$\mathrm{S(x)} = \sum_{i=\lfloor x \rfloor - \alpha + 1}^{\lfloor x \rfloor + \alpha} S_i L(x - i) \tag{2.13}$$

2.5.2 Dilation & Erosion

This subsection covers the concepts of Dilation and Erosion, which are morphological processing operations for grayscale or binary images [40]. The main take of these operations is that they all use structural elements. Such elements can be flat like diamond or disk-shaped, and non-flat like a ball shape [40].

To begin with, Dilation is the process of growing regions, thus, it fills holes and gaps in the contour of the image [40]. Dilation [40] utilises the *XOR* logical operand for an image X and a structuring element B, as shown in 2.14.

$$X \oplus B = X + b = \{x + b : x \in X \& b \in B\} \tag{2.14}$$

On the contrary, Erosion eliminates irrelevant details like noise in images [40]. Similarly to Dilation, this process also uses structural elements and is denoted by equation 2.15.

$$X \Theta B = X - b = \{z : (B + z) \subseteq X\} \tag{2.15}$$

These two morphological operations can be combined into more powerful processes. If first Erosion is performed, followed by Dilation, the process is called Opening. Opening smooths the objects and eliminates protrusions that are generally thin [40]. On the other hand, if Dilation is done before Erosion, the process is called Closing. It is used to eliminate small holes and to fill gaps in the image [40]. Examples of Opening and Closing [40] can be observed in figure 2.12.

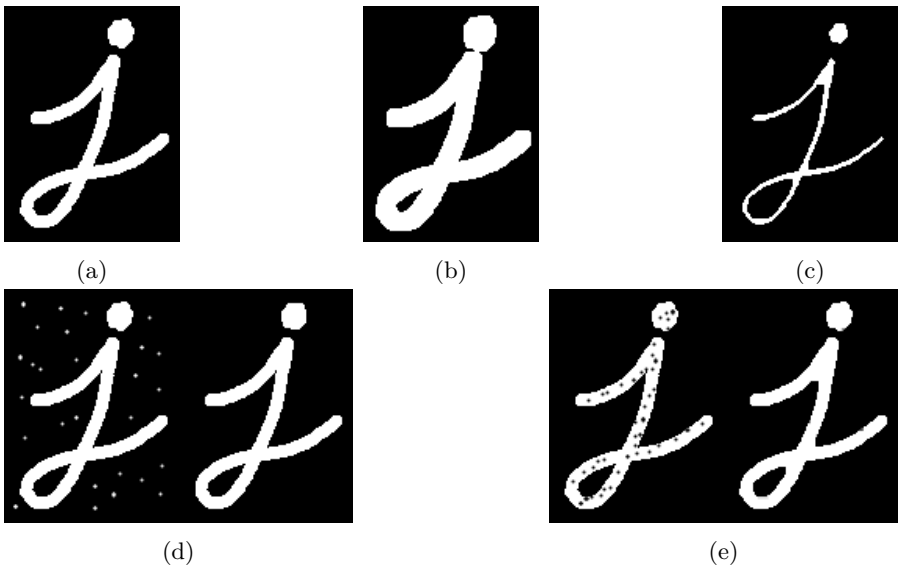


Figure 2.12: This figure shows a) an original image processed by performing b) Dilation, c) Erosion, d) Opening and e) Closing respectively [12].

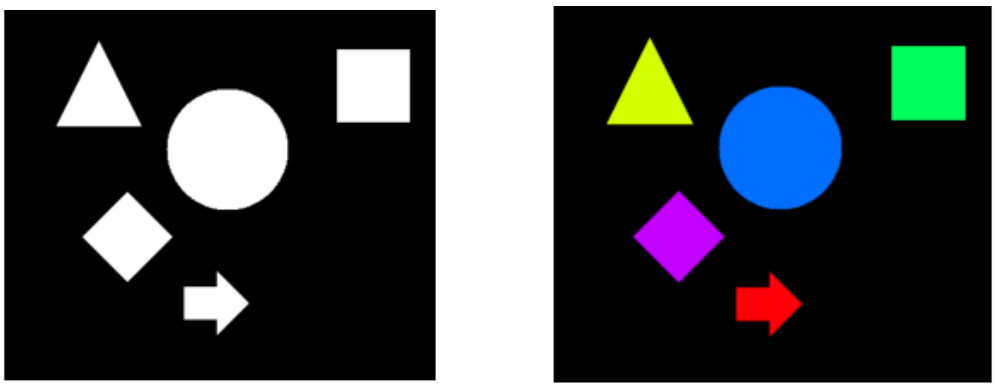
2.5.3 Finding Disconnected Components

Finding the disconnected components in an image is essentially finding all connected components, leaving only the disconnected ones. Finding these components is done with the help of a labelling algorithm part of Graph Theory [42]. These components are subgraphs where any two vertices are connected to each other and are not connected to any other vertices in the overall graph [19]. There are two types of pixel connectivity that can be observed in the list below, as per [19]:

- Four Pixel Connectivity - it connects all same-valued pixels along the horizontal and vertical axis.
- Eight Pixel Connectivity - it connects all pixels with the same value along all diagonals and edges.

The algorithm makes two passes over the image [19]. During the first pass, the algorithm checks every pixel whether it is of interest. If the pixel is white (255) it means it is part of the foreground, thus, the algorithm records it and checks its neighbours iteratively until the pixels above and to the left are background pixels [19]. If the pixel is black (0), it is skipped and a new label is assigned to the previously checked foreground positions [19]. If the above pixel is a different label to the left pixel of a certain position, they are both merged as the same label. This step is repeated for all pixels in the image [19].

The second pass over the image is to assign final labels by following each pixel's link to their parent until it reaches the root of the graph tree. After that, it assigns that label to the current pixel and this process is done for all unlabelled pixels [19]. Furthermore, an example of labelling connected components can be seen in figure 2.13.



(a) (b)

Figure 2.13: Figure a) represents the original image and figure b) shows the image after labelling the connected components [19].

This section of Image Processing has explored different methods for scaling and enhancing images. The project described in the paper uses some of the described methods after weighing their advantages and disadvantages. The choices made in this project, in terms of which method to be used, are based on testing and are highly dependant on the data. Therefore, the reasons behind why certain methods have been chosen are covered in future chapters. Furthermore, in the next chapter, the reader will be introduced to how and why these methods have been applied to process and enhance the generated images in the created Minecraft filter [48].

Chapter 3

WGAN Road Generation Filter

3.1 Aims & Objectives

As mentioned before, this project undertakes the task of creating a filter in Minecraft [48] by improving the road systems and combining that with already existing filters. The main point to take is that there are little to no other approaches that try to improve upon the roads of their methods. In chapter 2, other road generation approaches were covered. However, because of either their simplicity or their lack of autonomy [20], they cannot be used as a proper solution when it comes to creating an AI filter.

On the other hand, other methods decide to focus on making the buildings more lifelike and completely ignoring the road infrastructure that they are placed on [26]. Such approaches contradict entirely the field of Urban Planning, which suggests that roads are one of the most essential parts of urban planning and they take a significant percentage of the whole city's area, as per [24].

Therefore, this chapter describes a new approach to road generation. It uses a WGAN [41] to generate a road binary image that is to be applied in Minecraft [48] using MCEdit [25]. The previously-mentioned improvement of the road system is achieved by the used NN. This is so because by utilising Artificial Intelligence models, the generated road is unique and lifelike. These properties are due to the training set that the model was trained on. It is described more extensively further in this chapter, however, it is worth mentioning that the dataset consists of real-world roads. Thus, the generated images contain the specific characteristics of a real road.

Moreover, this project is a fully functional Minecraft filter [48] on its own. It is divided into sequentially executed modules. Parts of this project utilise methods that were previously

critiqued in this paper. However, their usage differs vastly from how they were used in other approaches. Therefore, it manages to harness their full potential with minimum drawbacks.

To achieve all of the above-mentioned functionalities and properties, this project takes advantage of various technologies. These technologies are covered in the following section.

3.2 Technologies

This section describes the required technologies for this project, as well as any datasets used for training and testing. The main programming language for coding the filter, as well as converting the AI model and training dataset, was **Python 2.7** on a local machine **IDE**. Because of python's deprecation, older versions of **Keras**, **Tensorflow**, **OpenCV**, etc., had to be used.

On the other hand, **Python 3.7** was used to train the WGAN model [41]. In addition, the model was entirely trained on Google Colab [4], utilising its GPU hardware accelerator. The cloud IDE provides different types of GPUs depending on their current availability. However, they should all be producing similar results, as they are all designed specifically for Artificial Intelligence model training. Furthermore, a detailed list of all the used libraries, alongside their respective versions, can be found in the appendices of this paper.

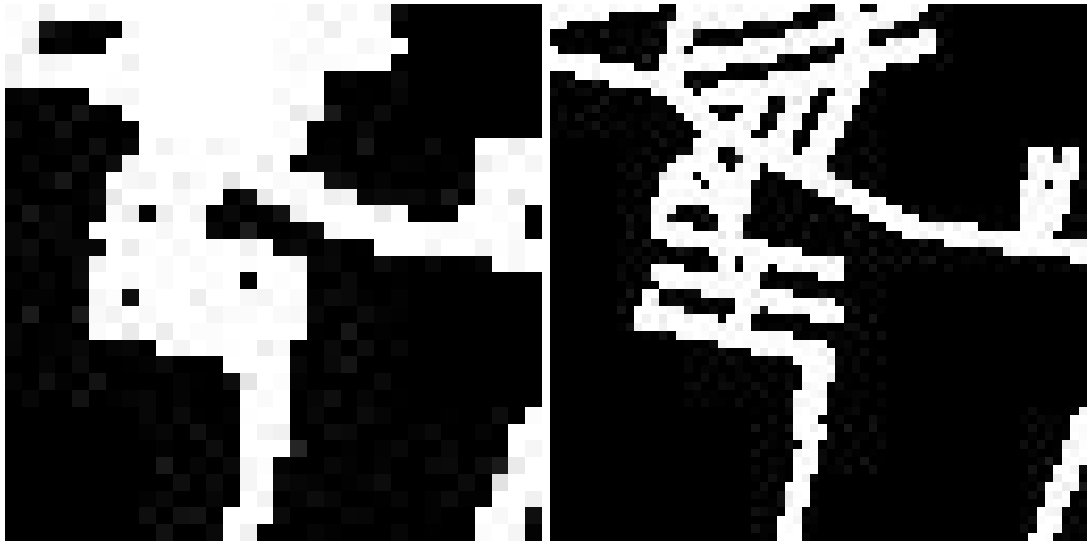
Additionally to what has been said, the used dataset for training the model is comprised of different binary road image maks of real-world roads. The data was obtained from the DeepGlobe Road Extraction Challenge [10]. The data consists of 12452 training masks and satellite images of roads [10]. However, for the purpose of this project, only the mask images have been used. Thus, the training dataset was cut down to 6226 images.

3.3 WGAN - Methodology & Design

Before diving into the main part of how the filter was created, the training process of the WGAN [41] model must be covered. It is not part of the filter, rather, it is completely separate from it. The process includes preprocessing the training data, defining the model's architecture, training the model, and finally, converting the model to a compatible version with the language version that MCEdit [25] uses.

3.3.1 WGAN's Architecture

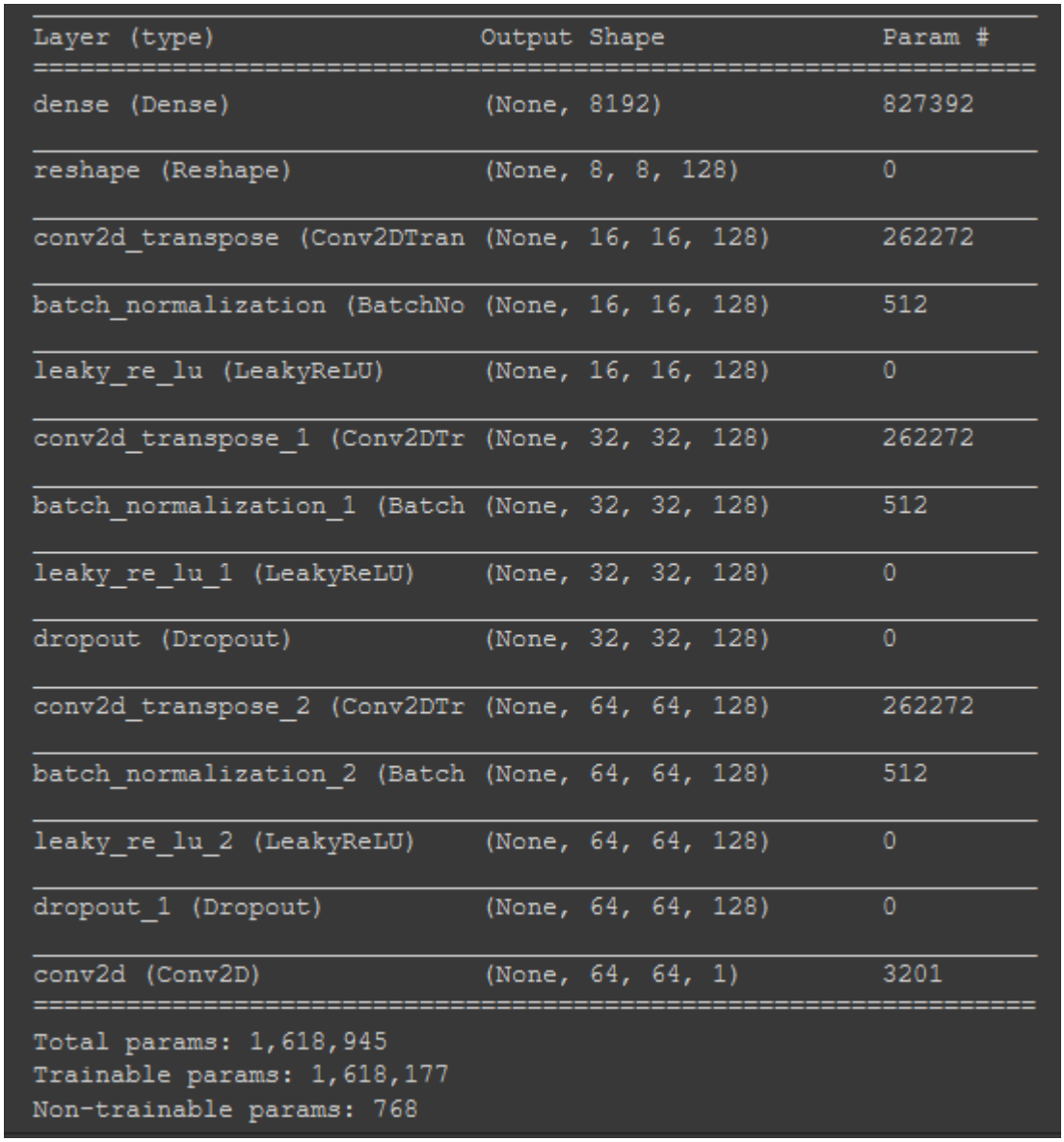
Chapter 2 gives an explanation to why WGAN [41] was chosen instead of other types of GANs [17]. Nonetheless, every model must be adapted for the data at hand. Because these networks are computationally expensive, the training images must be very small in size and still preserve enough information. After testing by observation, the image size of *64* by *64* was chosen. A comparison of both sizes can be seen in figure 3.1.



(a) (b)

Figure 3.1: The figure shows two random images taken from the training dataset [10]. Image a) is of size 32x32 and image b) is of size 64x64. It can be clearly seen that the smaller image has significantly less information contained after rescaling.

Knowing the training images' size and that they are grayscale, a model archtiecture can be designed. As previously said in chapter 2, the model consists of a Generator and a Critic [41]. The Generator consists of a Dense Layer [44], followed by three Convolutional Layers [28]. These convolutions are required because the Generator receives a noise vector that has an initial shape (8 by 8 by 128). The 128 is the sum of both the height and depth of the image, while the 8 by 8 is the starting size of the generated image. The Generator keeps upsampling until the actual size, hence the three Convolutional Layers [28]. Each layer upsamples the size of the input twice, making it 64 after the last layer. Additionally, LeakyReLU [37] is used as the activation function of these layers because Generative Adversarial Networks [17] usually suffer from sparse gradients. In the end, another Convolutional Layer generates the final grayscale image with the help of the Tanh function [37]. In addition, there are Batchnormalization and Dropout layers put in place to stop the model from overfitting the data [14]. A visual representation of the Generator can be seen in figure 3.2.



Layer (type)	Output Shape	Param #
dense (Dense)	(None, 8192)	827392
reshape (Reshape)	(None, 8, 8, 128)	0
conv2d_transpose (Conv2DTran	(None, 16, 16, 128)	262272
batch_normalization (BatchNo	(None, 16, 16, 128)	512
leaky_re_lu (LeakyReLU)	(None, 16, 16, 128)	0
conv2d_transpose_1 (Conv2DTr	(None, 32, 32, 128)	262272
batch_normalization_1 (Batch	(None, 32, 32, 128)	512
leaky_re_lu_1 (LeakyReLU)	(None, 32, 32, 128)	0
dropout (Dropout)	(None, 32, 32, 128)	0
conv2d_transpose_2 (Conv2DTr	(None, 64, 64, 128)	262272
batch_normalization_2 (Batch	(None, 64, 64, 128)	512
leaky_re_lu_2 (LeakyReLU)	(None, 64, 64, 128)	0
dropout_1 (Dropout)	(None, 64, 64, 128)	0
conv2d (Conv2D)	(None, 64, 64, 1)	3201

Total params: 1,618,945
Trainable params: 1,618,177
Non-trainable params: 768

Figure 3.2: This is a visual representation of the created Generator model. This includes the output shapes of all layers, as well as the number of parameters in the model.

On the other hand, the Critic downsamples the data to the original size of the Generator's input, which is 8. This network's architecture is very similar to the Generator's one. However, there is no Dense Layer [44] in the beginning and the three Convolutional Layers [28] downsample the input from *64* by *64* to *8* by *8*. In addition, the last two layers are a Flatten followed by a Dense Layer [44], utilising the Sigmoid function [37]. The Flatten Layer [28] is used to reduce the dimensionality of the data, so the information can be passed to the final Dense Layer [44]. Furthermore, the Sigmoid function has been chosen because the Critic needs to do binary classification, telling whether the image is fake or real. It was previously mentioned

that WGANs utilise weight clipping to maintain function continuity [41]. Thus, the suggested weight clipping range of (-0.01, 0.01) was used [41]. Full visualisation of the network can be observed in figure 3.3.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 32, 32, 128)	2176
batch_normalization_3 (Batch	(None, 32, 32, 128)	512
leaky_re_lu_3 (LeakyReLU)	(None, 32, 32, 128)	0
dropout_2 (Dropout)	(None, 32, 32, 128)	0
conv2d_2 (Conv2D)	(None, 16, 16, 128)	262272
batch_normalization_4 (Batch	(None, 16, 16, 128)	512
leaky_re_lu_4 (LeakyReLU)	(None, 16, 16, 128)	0
dropout_3 (Dropout)	(None, 16, 16, 128)	0
conv2d_3 (Conv2D)	(None, 8, 8, 128)	262272
batch_normalization_5 (Batch	(None, 8, 8, 128)	512
leaky_re_lu_5 (LeakyReLU)	(None, 8, 8, 128)	0
dropout_4 (Dropout)	(None, 8, 8, 128)	0
flatten (Flatten)	(None, 8192)	0
dense_1 (Dense)	(None, 1)	8193

Total params: 536,449
Trainable params: 535,681
Non-trainable params: 768

Figure 3.3: The model's Critic network visualised with all layers, output shapes, and parameters.

To finalise, the overall WGAN model [41] is comprised of the Generator and then the Critic connected to it. However, only the Generator is saved after the training process because the Critic is only used for training. Moreover, the mentioned design specifications for both networks have been chosen after various different training attempts. It was found that the more complicated a model is, the more prone to overfitting it is, also backed by [52]. Therefore,

because of the simplicity of the data and the good results shown, the aforementioned design has been chosen. Evaluation examples and results of the models can be found in the next chapter.

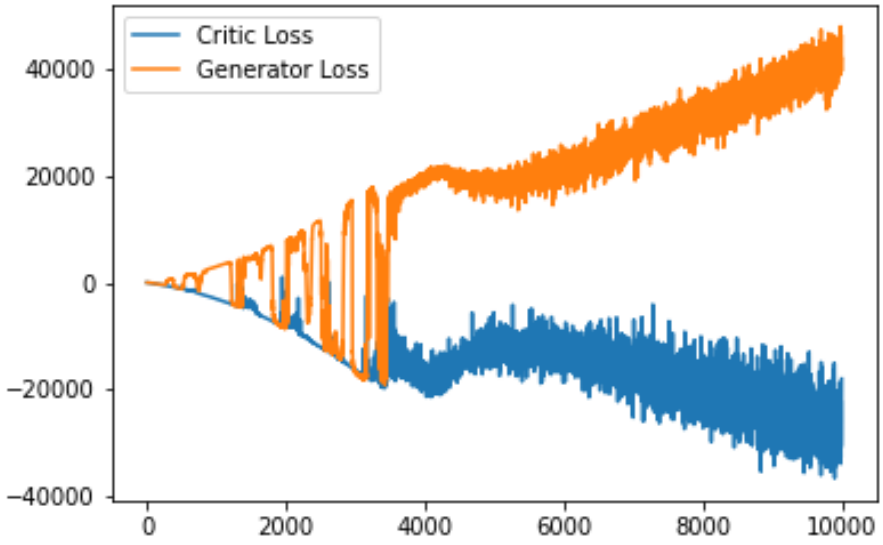
3.3.2 Preprocessing & Training

As mentioned in the previous section, the training images are of size *64* by *64*. However, their initial size is *1024* by *1024* [10]. Therefore, to scale the images down, preprocessing needs to be applied to the dataset.

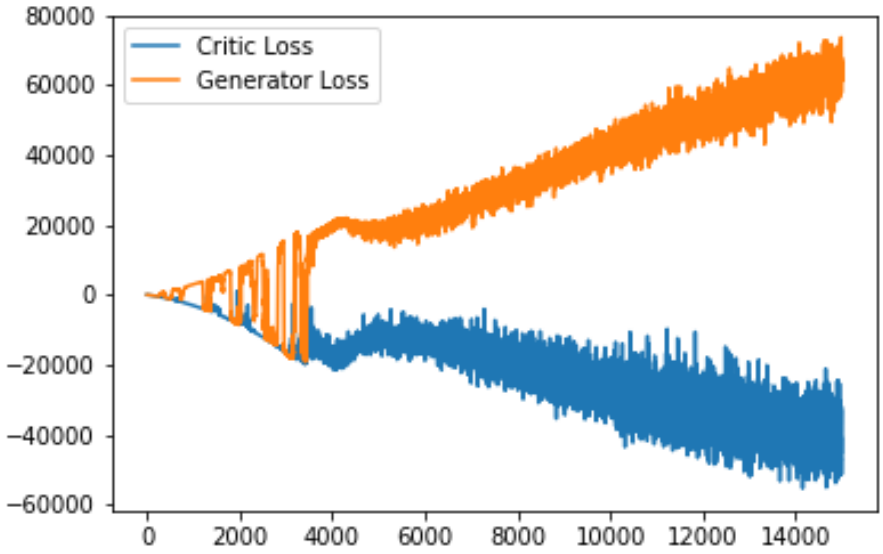
First and foremost, each image is loaded from a directory and resized to the desired shape. While resizing, Bilinear Interpolation [15] is applied to the image to lose as little information as possible. Moreover, no additional sharpening filter is applied. This is because edges need to be well-defined, thus, the edges are not highlighted additionally. On top of that, because the images are of grayscale, another conversion is performed. Every image has its pixel values changed to either 255 (perfect white) or 0 (perfect black). This conversion helps the model learn by not considering any middle pixel values, making the images essentially binary. The conversion formula for each image pixel X_i can be seen in equation 3.1. Moreover, image augmentation was also considered. However, it majorly slowed the training process without making big improvements in the results. Therefore, it was not used as an option for this model.

$$C(X_i) = \begin{cases} 255 & \text{if } X_i > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

After the preprocessing process of the training dataset [10], the model can be trained. The model took 15 000 epochs to train on Google Colab [4]. The training was stopped at that point because the loss of both networks started to oscillate within a certain range. The model was constantly monitored to see whether it is overfitting or not. In addition, every 5000 epochs the loss was checked to make sure it was constantly decreasing for the Critic and increasing for the Generator, as per [41]. The training history of the overall model can be observed in figure 3.4. Besides the number of epochs, the training process has other parameters that are adjustable. The Critic is trained more times than the Generator in WGANs [41], therefore, the Critic in this project was trained for 5 additional steps. This exact amount of steps and the batch size of 64 was taken from the original WGAN paper [41]. Furthermore, the generated results during training can be seen in figure 3.5. Additionally, the whole training process took around 4 hours before having Google Colab disconnect its runtime environment [4].



(a)



(b)

Figure 3.4: The first image a) shows the plot up until epoch 10 000. It can be concluded from the graph that the error is still decreasing for the Critic and increasing for the Generator [41]. Hence, the training process is yet to be terminated. On the other hand, image b) shows the training loss history for the full 15 000 epochs. It can be seen that the loss started to level after the $14\,000^{th}$ epoch, thus, the training process was stopped.

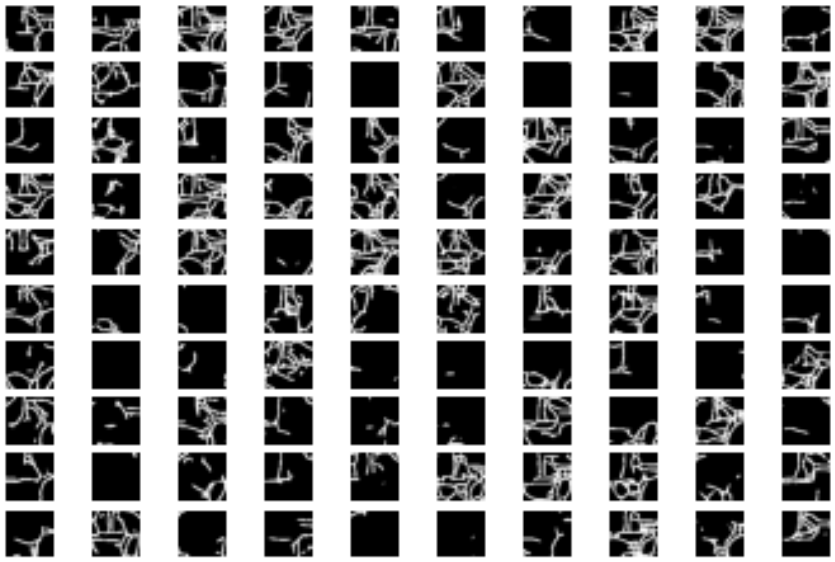


Figure 3.5: This figure presents the generated results during training on the $15\,000^{th}$ epoch. Clear roads can be distinguished. However, it must be also noted that there are almost empty generations. This is because the model could not learn from images that had very short or small roads. Nonetheless, only highly populated images are of interest. In addition, it has been accounted for this particular problem during the road generation in the created Minecraft [48] filter.

Because Google Colab [4] was used for training, Python 3.7 library versions were utilised. This would cause an issue when loading the Generator in MCEdit's Python 2.7. Thus, the newly trained network is converted to an older version, simply by transferring the weights to a new model and saving it again.

3.4 Filter - Methodology & Design

The filter, defined by this project, is designed to have 3 main sequential modules that can be seen in figure 3.6. Moreover, each module with its respective submodules has been described extensively in the upcoming subsections. These modules do not include the previously-mentioned WGAN [41] model, as it is not directly part of the filter.

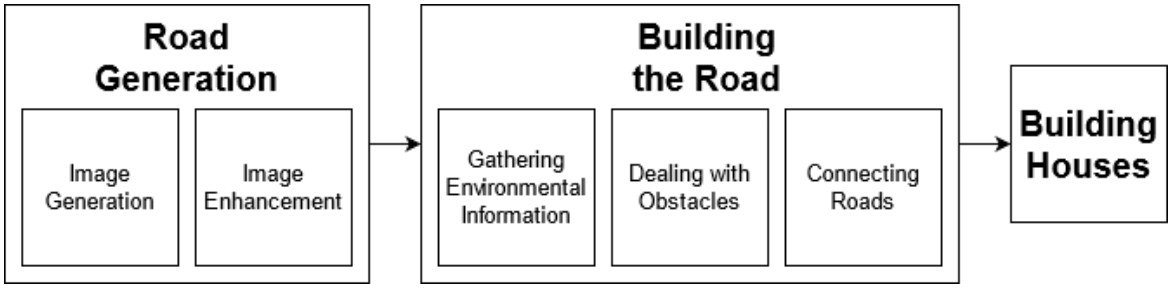


Figure 3.6: This figure shows the filter's modules with their submodules in a sequential manner from left to right.

3.4.1 Road Generation

This subsection covers in detail the first module of the overall filter. It is the backbone of the produced software application and it plays the role of generating the road system as a vector of positions. Similarly to every other GDMC [26] filter, this one also has a *perform* main function that is automatically executed upon calling the filter from MCEdit [25].

To begin with, the filter creates a Road Class that automatically loads the aforementioned WGAN [41] model and generates a series of road images. This is the Image Generation submodule, as per figure 3.6.

Image Generation

This submodule loads the created Generator and, using some noise, generates a predefined number of images. These images are then normalized to between values of 0 and 1. The normalization is required because of the used Tanh function [37] in the Generator.

After the images have been created, the submodule deals with the previously-mentioned problem of low-volume road images. With the help of a predefined parameter, the submodule checks whether there are any generated road images with at least 20% of coverage and adds them to a list. This percentage is a compromise between having either very big or very small road systems. The coverage, on the other hand, is calculated by getting the percentage of white pixels in every image. The predefined parameter decides how many images we want in that list of good coverage roads. To avoid having a low volume of roads, the images in the list are overlaid on top of each other. This results in a bigger road system. To keep the real-world aspect of the roads, a very small amount of images have to be combined. A recommended value is either 1 or 2, where selecting 1 would result in keeping the first generated road image with coverage above 20%. An example of this process can be observed in figure 3.7.

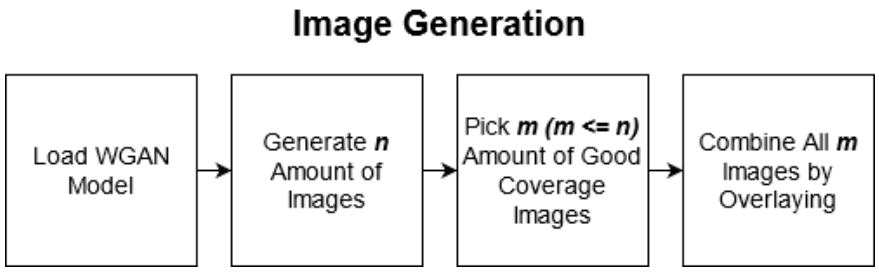


Figure 3.7: The figure depicts the process of Image Generation and how m out of n generated images are combined to produce one overall road system.

This process automatically leads to the next submodule that enhances the newly combined road system image and makes it ready to be applied on Minecraft [48].

Image Enhancement

This submodule only performs Computer Vision techniques for image enhancing. It consists of 3 steps that are executed sequentially and are vital for the overall quality of the produced road image. These steps can be seen below.

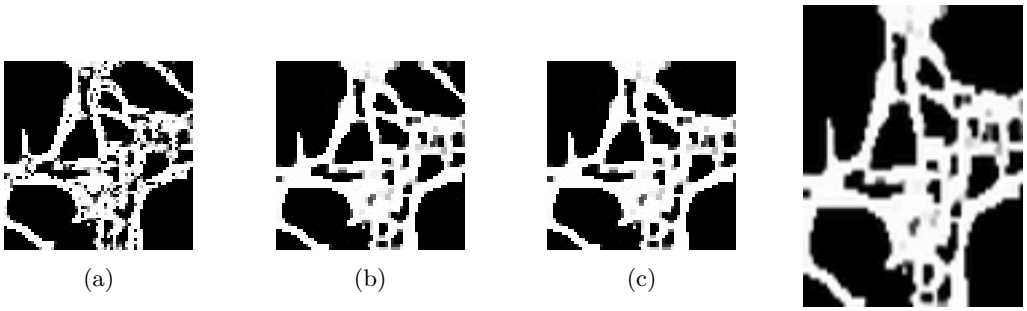
- Closing
- Removal of Disconnected Components
- Grid Conversion

The first step converts the passed image back into grayscale. This is done by simply multiplying every pixel value with 255, as it is initially a binary image. After that, the Closing process is performed on the image [40]. The structural element in place is a square with a *2* by *2* kernel. This process removes the noise and smooths the image.

The second step is used to deal with any disconnected roads on the newly generated image. Usually, the image consists of one continuous road. However, generation anomalies can be present. On top of that, artifacts may appear after combining several road images together. Therefore, all connected components are found using the connectivity of 4 pixels, as per [19]. The biggest connected component is recorded, as it is the main road system. This process is followed by removing every other component, resulting in an image with only one fully connected road.

The last and final step ensures Minecraft [48] environment compatibility. By using the selected grid for the filter in MCEdit [25], the road image is resized to match the shape and converted back to binary to make the overlaying process in Minecraft [48] easier. For resizing,

the Lanczos Interpolation [15] is used. Because there is only one image being processed, it is feasible to use a better but more computationally expensive interpolation method. Example images of every step can be seen in figure 3.8.



(a) (b) (c) (d)

Figure 3.8: The figure above shows an example of performing the Image Enhancing submodule on a road system image. The first image a) is the original image, the second image b) is the road system after the Closing operation [40]. Moreover, image c) is the road with its disconnected components removed, as it can be seen when comparing the top right corner of b) and c). The last image d) is the resized road that matches the selected grid size.

Additionally, it might be difficult for an observer to truly evaluate the quality of the produced roads in figure 3.8. This is because of the small size and distortion of the images. However, there are minuscule black pixels inside and around the roads that keep the semantics of a real-world road. The produced road is further evaluated in future chapters.

3.4.2 Building the Road

The module described in this subsection takes on the responsibility of applying the road system image in Minecraft [48] and adapting it to the environment. It has three main submodules that make the created road more adaptable and lifelike. Other GDMC [26] filters do not really deal with obstacles, as seen in chapter 2. To account for that, this filter learns the environment. All three submodules are responsible for that. However, each submodule deals with a different type of environmental obstacle. The first one gathers valuable information from the environment that the subsequent submodules will use.

Gathering Environmental Information

In order for the filter to really consider the environment, when applying the generated road system, it needs to know its surroundings. Therefore, valuable information is gathered from the selected area.

To begin with, the filter gets the current biome. It is an iterative process that goes through

every present biome in a slice of the selected area and picks the one with the highest probability. The probability is the percentage of a biome present out of all seen biomes in the area. For example, if there are two biomes present, and the first one is seen 3 times out of 5, then the probability of selecting that biome would be 60%. After choosing a biome for each of the slices, the one that appears the most in the newly created list is picked.

Based on that information, the filter picks the types of blocks, slabs, and stairs. These are the materials that they are made of. The environment plays a vital role in choosing the block types, as they need to match the environment. It is not plausible for a jungle tree woodblock to appear in the desert, e.g. In addition, because the filter can be matched with other filters, all selections can be easily overwritten by another parent filter. If that other filter does not account for the environment, the aforementioned functions will be automatically performed.

The next step is making an environmental floor map. It is a dictionary of positions and 3D coordinates. Depending on the selection made by the user in MCEdit [25], the filter maps every natural block that could be seen, except air and lava. The created map has every coordinate of the floor of the selected area. Additionally, another array makes a 2D map of the floor, so it can be later used to easily apply the 2D road image on it. It also records any obstacles and is used for finding the fastest path between two disconnected roads. In the end, the submodule sends the gathered information to the main road building method that is later used to build roads, tunnels, and bridges.

Dealing with Obstacles

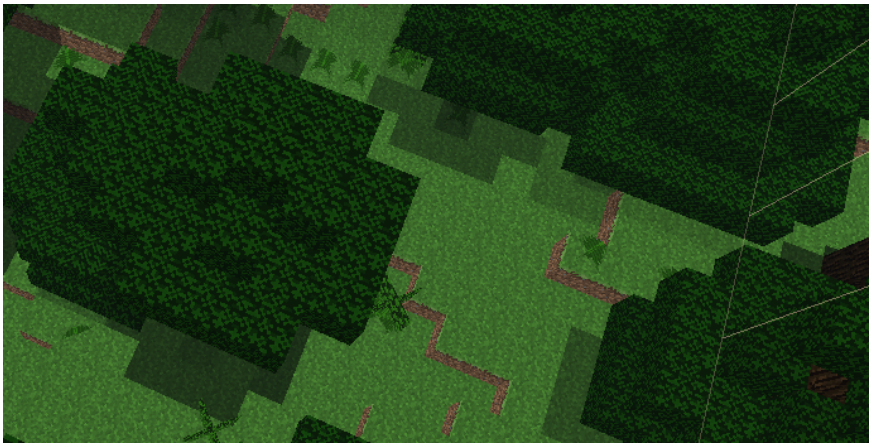
The obstacles that this submodule accounts for are lava, mountains, and trees on the path. The main road building method goes through every floor position and checks for these obstacles.

First and foremost, it checks if there is a lava block on any of the recorded floor positions. That ensures that there is no lava on the surface of the settlement in general. Instead of simply removing the lava, the filter gathers the most present surrounding block, which is not an obstacle of itself and replaces the lava with that block type. This ensures, to some extent, that the Minecraft [48] surface remains smooth and untouched.

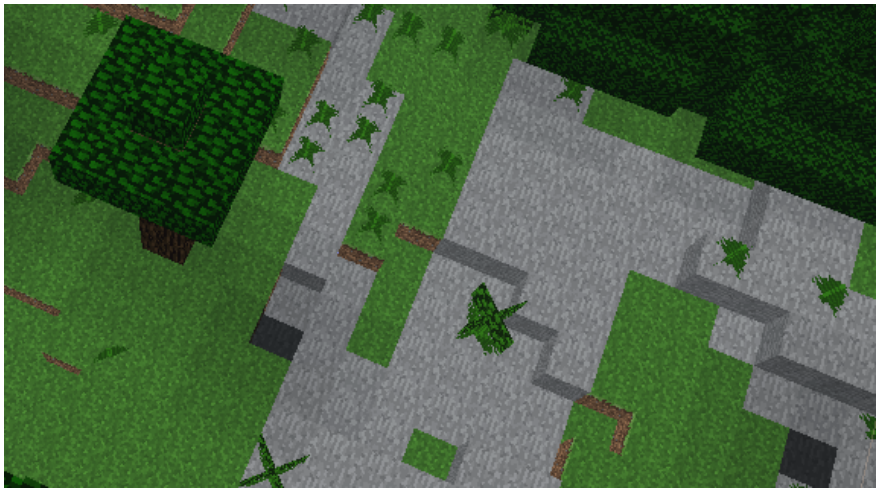
After that, the filter starts using the generated road system. It checks if any of the road positions in that image correspond to any of the floor positions that were previously mapped. If the coordinates correspond with each other, it means that there is a road block to be put. The position is checked if on or above it there is a tree present. Because MCEdit [25] has no easy way of finding the trees, an iterative approach has been taken. If there is a tree on or above

the road block, then every block on the y level axis is recorded until a tree block is no longer present. While doing that, for every block on that axis, the two other horizontal and vertical axis are checked iteratively until a tree is no longer present. Upon completion of this process, the filter is left with a list for each of the x, y, and z positions. A bounding box around the tree is drawn by taking the minimum and maximum recorded position values. Furthermore, every tree block is removed from that 3D bounding box.

However, it is possible for two trees to be connected, while one of the trees is on the path and the other is not. In order to preserve the environment, any other tree that has been affected by that bounding box is regenerated. This is done by going through every tree in the specified 3D range, that is not on a road block, and growing its leaves back. A visualisation of this process can be seen in figure 3.9.



(a)



(b)

Figure 3.9: Image a) shows the environment before applying the road on it. On the left side of the image, two trees can be seen connected to each other. The filter understands that this is not one big tree. Instead, as can be seen in image b), the tree on the road is completely removed, while the connected tree is just trimmed down.

The final obstacle that this submodule deals with, are mountains. Sometimes, Minecraft [48] can generate high mountains that are not really passable. If the selected area goes through a mountain, the road will see that there are other blocks in front of it and will automatically build a tunnel. In addition, it also fills in any gaps on the ceiling or walls of that tunnel that have occurred during the digging process. An example of such a tunnel can be seen in figure 3.10.

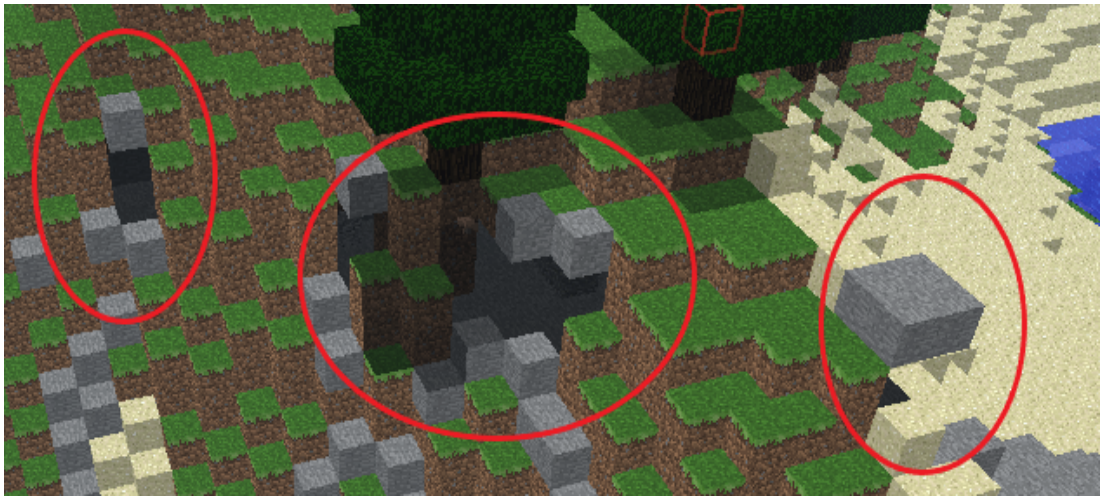


Figure 3.10: The figure shows how the road adapts to obstacles in front of it by digging a tunnel. Every red circle represents an entrance/exit to/from the tunnel. The size of each opening depends on the size of the road that connects to it.

Connecting Roads

This submodule fixes any discontinuities on the road infrastructure, occurred because of the terrain. For example, if the terrain is not passable due to big water bodies, or simply if the road has been disconnected when overlaying.

First, the filter records the position of every road coordinate on that second 2D floor array that was covered in the first submodule section. This map becomes essentially an image of 255's (where a road is present) and 0's. Thus, the aforementioned algorithm for detecting connected components can be utilised [19]. If there are any disconnected components, the filter records any visible water bodies on that 2D array, as well.

Having the whole road map with water put on it, the filter calls the A-star algorithm on that array [51]. At first, the algorithm was performed at most two times per pair of disconnected roads. The first time to try and find a connection between any two roads, and the second, if there is water that makes the connection unfeasible, a bridge is built. However, after testing, this considerably slowed the filter. Therefore, the A-star [51] is called only once, and then if

the path crosses any water bodies, a bridge is built.

The filter keeps track of when a bridge starts building, thus, it knows its beginning and end. In that way, the proper blocks can be put in place. Moreover, a bridge is built if and only if there is water between both roads and the path found by the A-star [51] algorithm passes that water body. Otherwise, the shortest land path will be found. This submodule keeps the road continuous and passable. This can be considered a big improvement compared to other GDMC [26] filters, covered in chapter 2. In addition, a visual representation of connecting such discontinuous roads can be seen in figure 3.11.

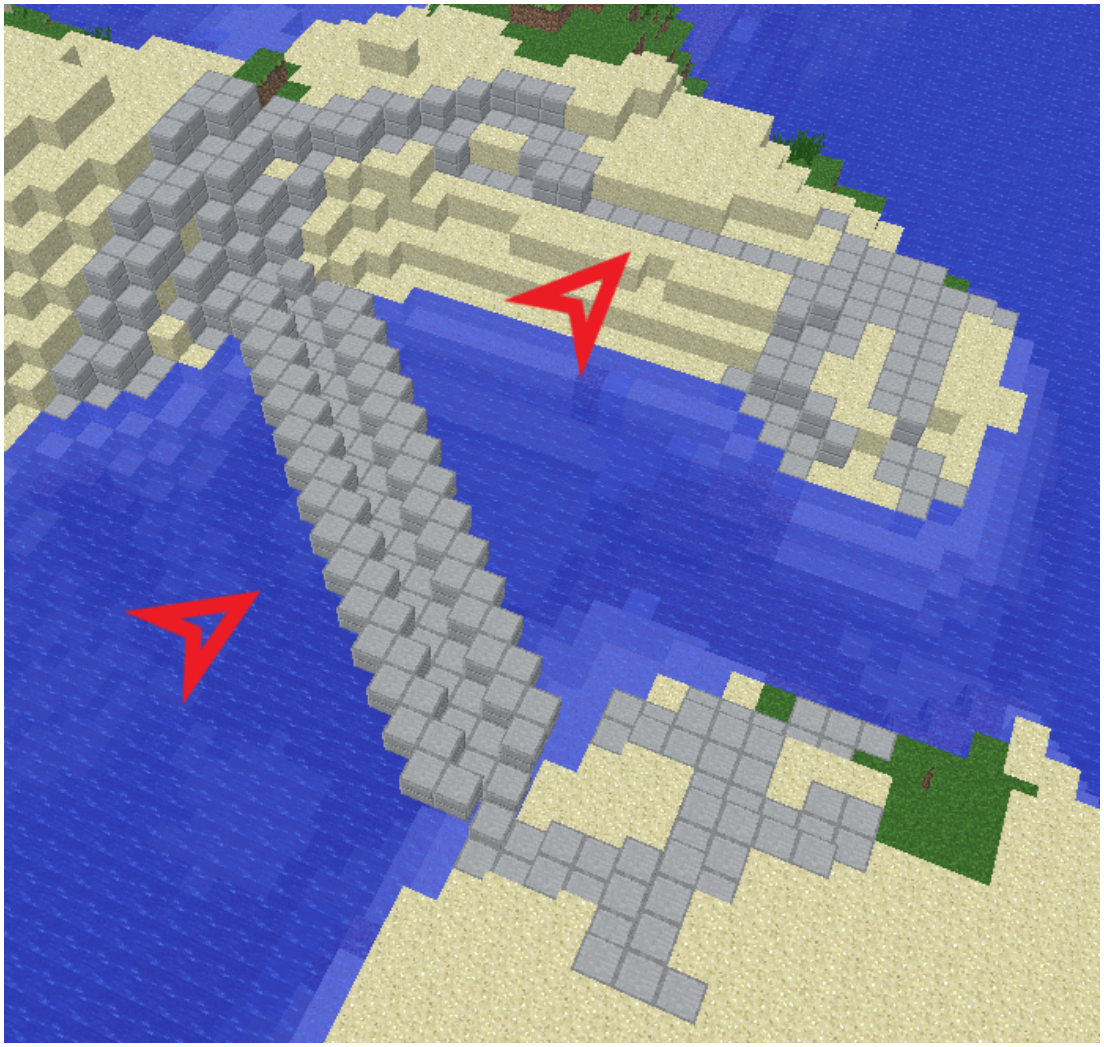


Figure 3.11: An example image showing how disconnected roads are connected on land and over water, with the help of additional roads and bridges. The red arrows point to the places where the connections have occurred.

3.4.3 Building Houses

This module, unlike the others, is not directly part of the filter. Rather, it uses a 3rd party filter to build the houses. The filter is designed in a way that can be either used individually to build the road system or as a part of another filter. Nonetheless, it still knows where the houses are, thus, it can support the roads even after they have been built.

Moreover, its flexibility allows for it to be built either after or before the houses. If it is built before the houses, as a Top-Down approach (chapter 2), the road is further connected. After placing the houses, the road might get disconnected, or the houses might be disconnected from the road itself. Therefore, this module also ensures that there are no disconnected houses. This is done by performing the A-star [51] algorithm after placing the locations of each house on the 2D floor map as a part of the road. Thus, the houses can be treated as a disconnected component [19] and can be easily connected with a road. This ensures the whole road infrastructure of the settlement and also makes the houses reachable.

To conclude, as previously mentioned, this project is to be used with other filters to improve their road infrastructure and build houses. The examples provided in the next chapter show this filter placed on top of David Mason's GDMC submission approach [34]. Furthermore, the road systems have been evaluated individually, as well as the overall quality of the generated settlement. On top of what has been already said, the chapter also goes into the disadvantages and advantages of the produced filter.

Chapter 4

Evaluation & Results

In this chapter, the whole project is critically evaluated against other approaches and on its own. It also provides the reader with analytical and visual data that can be used for further judgement of the produced method. Additionally, every part of the produced filter, including the WGAN [41] model, is critiqued accordingly using consistent units of measure. This ensures that when comparing against other approaches, the evaluation will not be biased. Nonetheless, these units of measure depend on the tested part, thus, they may differ.

4.1 WGAN Evaluation

The model was already evaluated in the previous chapter. However, it was not critiqued properly. Therefore, this section gives an explanation of what went wrong during the designing, training, and generating stages of the model.

To begin with, the model's design was inspired by another WGAN [41] architecture that was initially built for the MNIST dataset [11], and had to be adjusted [7]. The architecture was completely changed to accompany the road dataset [10]. The final design was agreed on after multiple training iterations. The initial design was also tried, however, the MNIST images [11] are of a very small size. Additionally, as it was seen in chapter 3, the road images need to be of a big enough size. Moreover, more layers were added and the number of filters in each layer was changed. That resulted in higher model complexity that ensured pattern learning. Additional Dropout and Batchnormalization [14] layers were added to make sure that the model could properly learn the already small in volume dataset.

Furthermore, different data preprocessing methods were tried for the dataset. It was found that the more the image was filtered, the worse the model would learn. In addition, sharpening the images resulted in the model not generating properly-looking roads. It would generate a cluster of white roads, as the edges of the images would be enhanced. This phenomenon can be observed in figure 4.1.

Figure 4.1: The figure shows the generated roads after 15 000 epochs on data with sharpened edges. It can be seen that the model cannot learn the individual road arteries, thus, it is unusable.

Also, the model was not able to be trained for more than around 17 000 epochs because of the Google Colab [4] free trial limitations. This resulted in a more limited model because of the time constraints. Nonetheless, the produced results can be deemed satisfactory.

Moreover, the model has its disadvantages as specifically said in chapter 3. Sometimes, the model would generate almost empty images. This is because of the dataset [10], as it consists of a lot of single-road images. Producing such results is not that common, and the filter further accommodates for that issue. Another problem, that was identified, was the balance between the time that it took training and the complexity of the model. The dataset does not consist of that many images [10], and in order to preserve the semantics of the roads, the images had to be considerably big for the WGAN [41]. All of this, including Google Colab's [4] usage

constraints, contributed to a more complex model that took a lot of time training. This, on its own, made the testing of the model's design extremely difficult and time-consuming.

In conclusion, the trained model produces good quality road images, with the exception of when there are single roads. More training would not necessarily improve its quality. However, testing more improved architectures and different types of GANs [17] should improve the overall results.

4.2 Filter Evaluation

This section evaluates the quality of the produced road system on its own, as well as against David Mason's generated road [34]. Furthermore, it critiques the filter's ability to deal with obstacles, maintaining the environment's structure, and keeping the continuity of the road infrastructure. In addition, the overall quality of the generated settlements between both filters is evaluated.

4.2.1 Road Evaluation

Because the GDMC [26] competition judges the produced results not by using analytical data but by using human judges, the evaluation of the produced roads may vary. Only factual arguments are presented in this section that should eliminate the bias factor when evaluating. This, however, can only be done by looking at visual examples.

First and foremost, the roads' overall quality will be discussed. This includes its shape and how it is overlaid in the given environment. An example of a generated road can be seen in figure 4.2.

(a)

(b)

Figure 4.2: This figure depicts two images of a generated road. The first image a) shows the road overlaid on an uneven surface. Image b) shows a different road on a flat surface.

The roads in figure 4.2 have been automatically generated by the filter. Right away, it can be seen that every artery of the road system has a different size. In addition, there are big filled spaces that represent city centers or squares. Similar to a real-world road, it has different

sized roads, where around the center, the wider roads can be seen. There are also house plots defined usually in the center of the road system, but also around it. It can also be seen in image a) that the road preserves the elevation of the environment. There are no roads built 2 blocks away from each other. On top of that, image b) shows an example of how the road does not take the whole selected area. Rather, it leaves space for further expansion. If houses are placed there, the filter will automatically connect them to the main road system.

On the other hand, the roads suffer from a problem where the house plots are too small. This can be clearly seen on both images in figure 4.2. These are slots that cannot be fully utilised. This issue can be overcome by either generating more sparse roads or by filling the holes with ornaments or farms. Nonetheless, this problem makes the filter harder to adapt to other filters that build houses on top of the already generated roads.

Moreover, one must compare against other approaches to truly evaluate a solution. Thus, the images in figure 4.3 show the generated road by David Mason's filter [34].

(a) (b)

Figure 4.3: Both images show a generated road by David Mason's filter [34]. Image a) shows the road on an uneven surface, where image b) depicts the road on a flat surface.

It can be immediately seen the specific shape of the generated roads in figure 4.3. It must be noted that the generated shape is always the same, with slight variations depending on the selected size. This can be considered a drawback against the filter described in this paper, as

the roads lack authenticity and look hardcoded. In addition, every road artery is the same size, which makes the road system not creative enough to be made by a human. Another issue is that the road does not interact with the environment. In image a) it can be seen that there are no trees removed from the road, rather, they are just kept on it. This is unlike the image a) in figure 4.2. Some trees may appear that they are still on the road, however, that is only because there are small openings in the road that allow for a tree to be there.

Nonetheless, the overall appearance of the road looks attractive and aesthetic. On top of that, there are well defined house slots that can be fully utilised. However, without a doubt, it lacks realism compared to the road generated by this project.

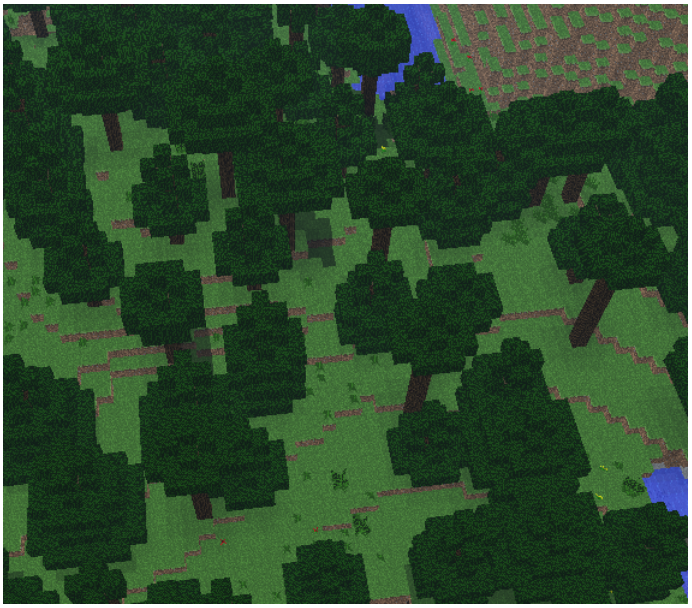
4.2.2 Dealing with Obstacles

In the previous section, it was briefly mentioned the ability of the filter to remove trees if they are on the road. This is not only limited to trees. It is also true for lava, water bodies, and mountains, as it was covered in chapter 3. Each type of obstacle will be covered individually with visual examples.

Trees

To begin with, removing trees from the road is a valuable asset for any filter. As it can be seen in figure 4.3, David Mason's filter [34] does not remove any of the trees if they are on the road. This is unrealistic, however, it also helps preserving the environment.

On the other hand, the filter proposed by this paper, removes any trees on the road itself and regenerates any connected trees that have been also affected by the algorithm. This can be seen in figure 4.4. However, the algorithm is not perfect. It can be seen in image b) in the same figure that there are some tree leaves not removed. Although Minecraft [48] automatically removes leaves in the air, it still affects the overall look of the environment. It is not aesthetically pleasing. The issue is with MCEdit [25] limitations. The platform does not allow users to easily locate trees and mark them. Nonetheless, the effectiveness of the algorithm is almost perfect, as this phenomenon can be seen only in rare cases, where there are multiple connected trees. In addition, by changing the natural environment, the generated road gains more lifelike human touch to it. As in the real world, people would either go around trees or remove them.



(a)

(b)

Figure 4.4: The first image a) shows the original unfiltered environment. Image b) depicts the environment after generating a road and removing any trees on it. The red circle shows an instance where the removal algorithm did not fully remove the tree leaves. An explanation for that could be because there were 3 connected trees, or the leaves were outside of the selected bounding box.

Lava & Water

In chapter 3, lava was covered as a potential threat to the environment. The filter simply removes it and replaces it with the most common natural block around that lava block. Therefore, the filters' ability to remove that obstacle cannot be evaluated as it is always 100%.

However, water bodies cause massive disruptions when building a settlement. The road can either be applied on it, as David Mason's filter [34] does, or have the road adapted to it. In figure 4.5 it can be seen that David Mason's approach [34] does not account for the water when building the roads. This is unrealistic and a big drawback of that approach.

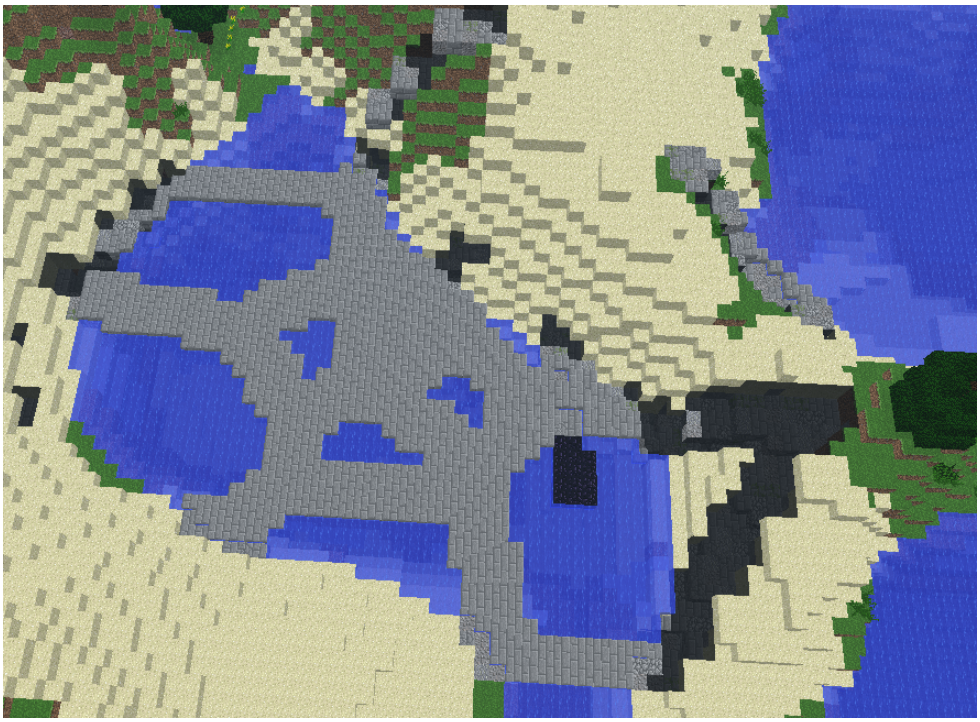


Figure 4.5: This figure shows the inability of David Mason's [34] generated road to avoid water obstacles.

On the contrary, the filter proposed by this paper knows that there is water while applying the road. Thus, it disconnects it and then connects it back with the help of bridges, observed in figure 4.6. This is a huge improvement to David Mason's filter [34], as it brings aspects of Urban Planning [24] to the environment. Additionally, it makes it possible for multiple different settlements to be created and connected via bridges.

Figure 4.6: The depicted image shows how the proposed filter builds bridges over water bodies.

Nonetheless, there are some limitations to the approach taken in this paper. If there are many disconnected roads, really close to each other, multiple bridges might be built. This makes the produces settlement look unrealistic, as it can be seen in figure 4.7. However, this occurs only in rare cases where the selected region is very small. This drawback is outweighed by all the other advantages that dealing with water obstacles bring to the generated settlement, as mentioned previously.

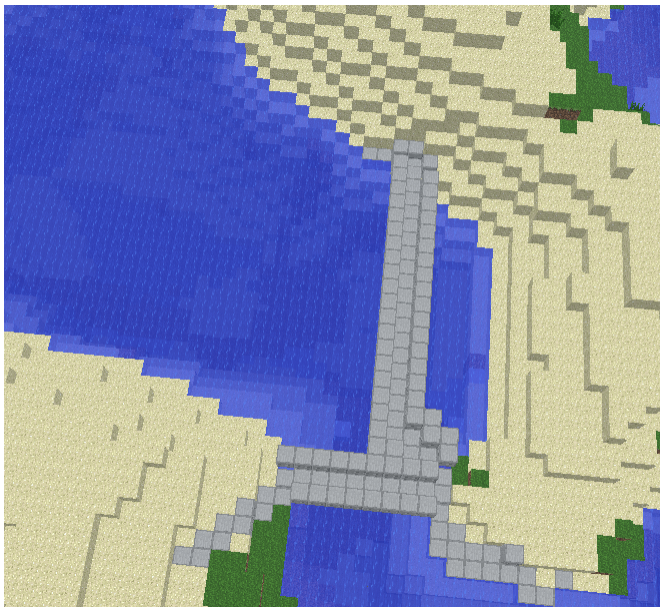
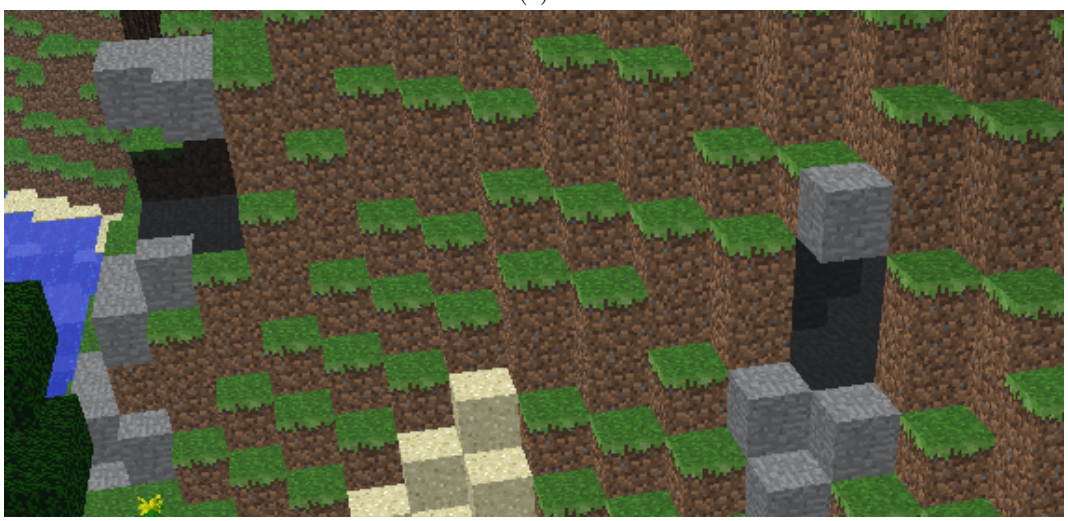


Figure 4.7: This figure shows a limitation of the filter's ability to overcome water obstacles. After testing, it was concluded that this occurs only if the selected MCEdit [25] region is of small size, as there is not enough space for the bridges to be built apart.

Mountains

As it was previously mentioned, the proposed filter deals with high unreachable elevations by digging tunnels. Although that is a great way to keep the continuity of the road infrastructure, it has its limitations. The built tunnels are not very detailed. The walls are kept as the initial blocks and there is no accommodation for lights and emergency exits. In addition, the size of the tunnel depends on the size of the road that goes in. This preserves the aesthetics of the road, however, it might result in very small or very big tunnels, that are disproportionate to the size of the mountain. Figure 4.8 shows images of built tunnels and highlights their issues.

(a)



(b)

Figure 4.8: Image a) shows a well-built tunnel with several exits. Image b) depicts a very small single tunnel with one exit. It can be seen that the walls of the tunnels are kept natural and there is no lighting.

On the other hand, David Mason's filter [34] does not accommodate for these obstacles at all, as it can be seen in figure 4.9. Therefore, although limited, the filter's ability to deal with such obstacles is a big improvement upon that other approach.

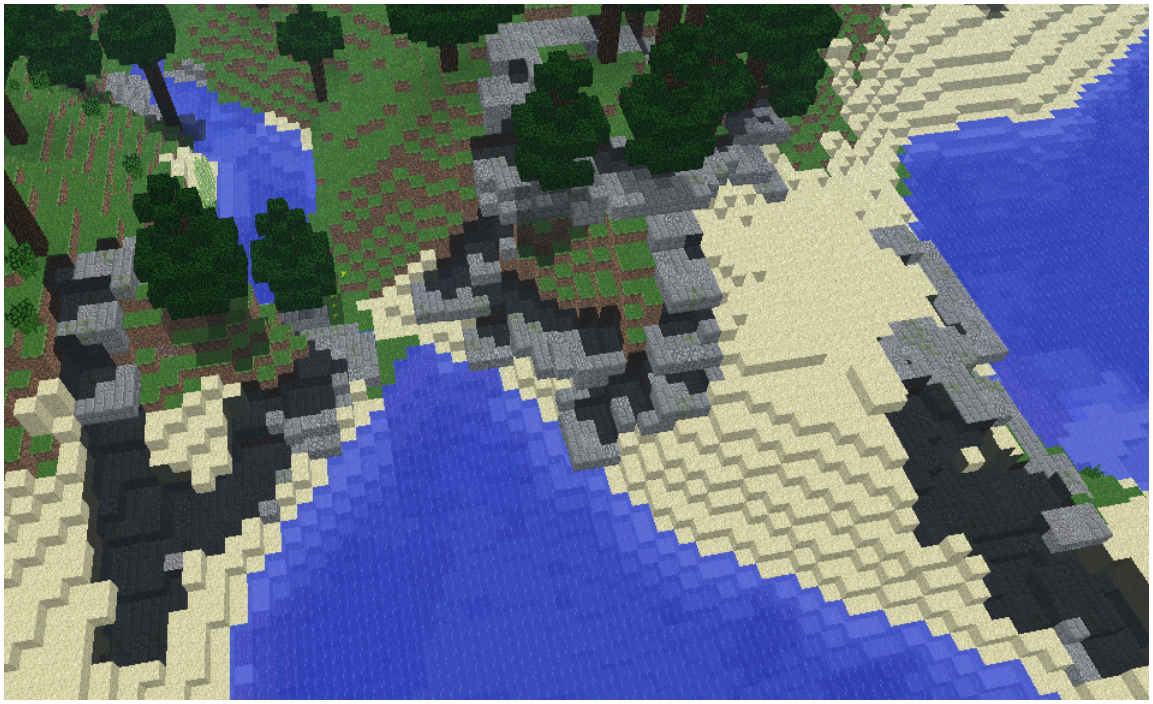


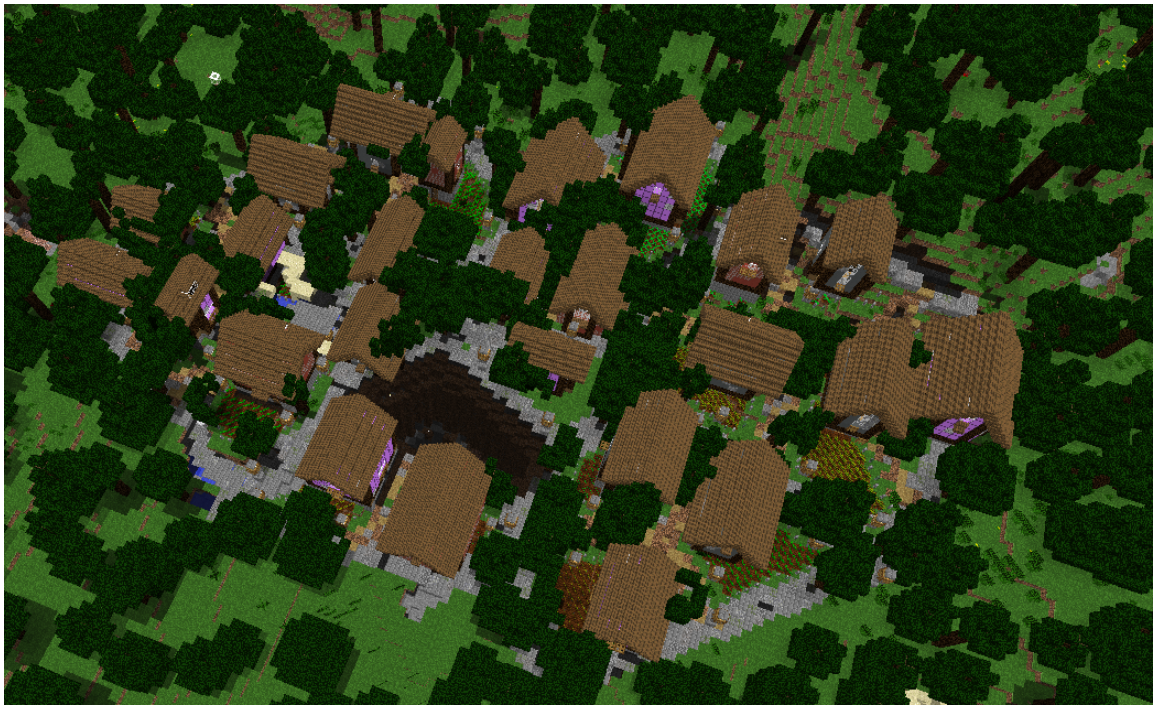
Figure 4.9: This image shows the inability of David Mason's filter [34] to deal with high elevation obstacles. The road is disconnected where a tunnel should be built.

4.2.3 Settlement Evaluation

The filter proposed by this paper does not build any buildings, as previously mentioned. Instead, it connects with other filters and generates the road that these buildings are later placed on. This has the aim of improving the overall quality of the produced settlement. To evaluate that, the filter is combined with David Mason's one and compared against the original David Mason filter [34].

To begin with, figure 4.10 shows the original David Mason [34] generated settlement on a selected region. The first, and most obvious observation that can be made by looking at both images, is the giant pit hole. It is placed next to the houses, without any walls around it. This, if comparing against real-world settlements, is unrealistic. Additionally, because the shape of the road is always the same, it tries to fit as many buildings in one place as possible. This can be easily seen in image b), where in the center there are four houses next to each other. Obstructing the entrances of the buildings leads to unreachable houses in a real-world scenario.

Nonetheless, the houses match the environment and they are all connected to the road as they are placed given the same shape. The generated settlement is a good solution to the GDMC competition [26] that adapts well to the environment. However, it lacks realism in terms of how the houses are placed on the road and the road infrastructure itself.



(a)

(b)

Figure 4.10: This figure shows two images of a generated settlement by David Mason's filter [34]. Image a) shows a generated settlement on an uneven surface, where image b) shows the settlement on a flat surface.

After critically evaluating David Mason's filter [34], one can evaluate the filter proposed by this paper. It must be noted that the placed buildings are the same in terms of visuals and the hole is still present. This is due to the fact that the filter is only used to build the roads and connect any disconnected houses. However, it tries to fix the issues posed in the previous paragraphs. This can be seen in figure 4.11. Moreover, this same figure has been referenced in the following several paragraphs.

To begin with, the hole is accounted for by building a bridge over it. This can be seen in image b) of the same figure 4.11. To add to what has been said, in image a) the road is built with respect to the hole. If the road goes next to it, a bridge is built instead, making it safer and more realistic compared to David Mason's filter [34].

Another major difference between both filters is the placement of the houses. Because the filter proposed by this paper generates a road with differently sized house slots, the buildings are placed more naturally on the road. This can be clearly seen in image a), where, on the left, there is a group of trees, and the houses are placed around them and are evenly spaced. The main take is that the houses are not cluttered as much as in the original filter and take into consideration the environment. This is so because of the generated road.

On top of that, there are certain places where the roads are thicker, which can be used as city squares. For example, this can be seen in image a), where the road is extended to a nearby hill and thickened at that exact location. This makes it infeasible for houses to be built and the space is left for further decoration (e.g., putting a monument).

Furthermore, image b) clearly depicts the functionality of connecting disconnected roads and houses. In the center of the image, there are single-sized roads that connect two houses together with a bigger centerpiece. In addition, the houses at the very top and left have an extended road connecting to the bridge and the whole road infrastructure. To even add to that, these road extensions try to connect directly with the doors of the houses. This makes sure that the whole settlement is fully connected and continuous, making it more lifelike.

To conclude, the filter proposed by this paper improves the overall quality of the produced settlement. Because the only difference is the road system, it can also be concluded that the better the road, the more lifelike the settlement. Additionally, the generated settlement can be further improved if a new filter is built on top of the proposed one, taking advantage of the exact positions of the road and its structure. Nonetheless, although the evaluation is subjective, there are certain aspects of the road system that can be definitively deemed as improved. Further generation and limitation examples for both filters can be found in the appendices of this paper.

(a)

(b)

Figure 4.11: The figure shows the settlement generated by the combined filter made of the proposed by this paper filter and David Mason's one [34]. Image a) shows the settlement on an uneven surface. Image b) shows the settlement on a flat surface.

4.3 Runtime Comparison

This section compares the runtimes of the filter proposed by this paper with the original David Mason filter [34]. Moreover, the runtime of both combined approaches is examined. The filters are run on the same local machine CPU. The machine has the following specifications:

- CPU - Intel Core i7-4720HQ
- RAM - 8GB
- GPU - NVIDIA GeForce GTX 960M

Furthermore, the table underneath this paragraph shows the average execution time for all filters. All runs have been made on the same selected MCEdit [25] region, which should make the results as consistent and reliable as possible. The region can be seen in figure 4.12.

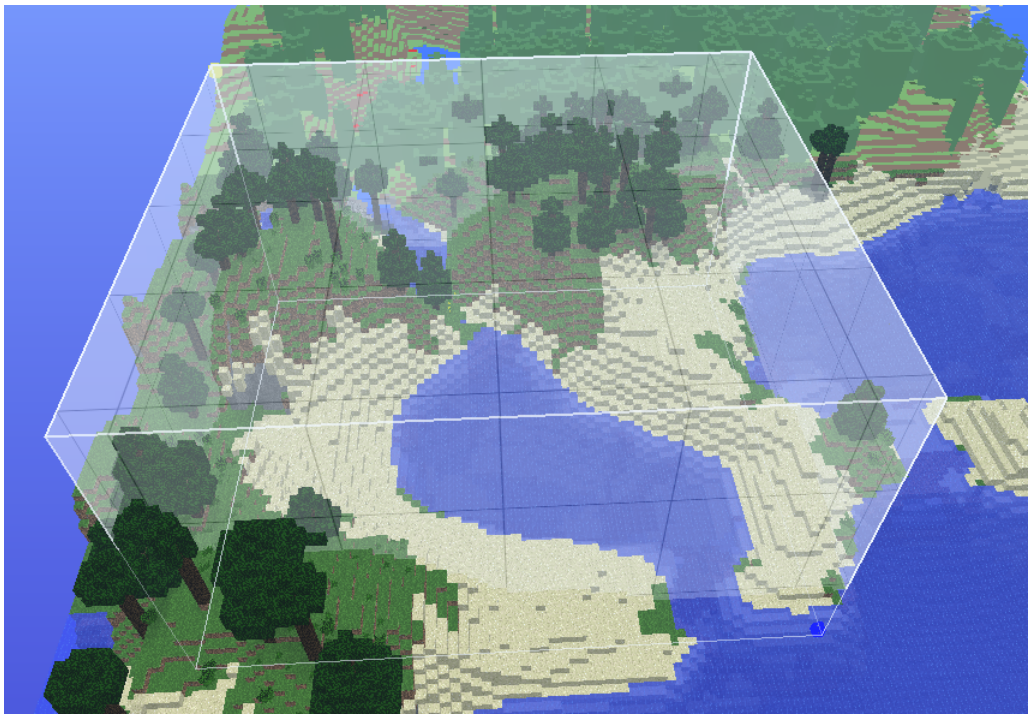


Figure 4.12: The figure shows the exactly selected region that all filters have been tested on. The region is relatively large, thus, the results should be representative of the overall filter runtime. In addition, it includes obstacles like mountains and water, which further slows down the execution time.

	Road Filter	**David Mason's Filter**	**Combined Filter**
Runtime	16.528s	35.633s	46.993s

The table above shows the execution times of the two filters individually, as well as of the two filters combined. It can be seen that the runtime for the filter described in this paper is approximately twice as low as David Mason's filter [34]. This is to be expected, as the filter only generates the road system without any buildings.

On the other hand, the combined filter's execution time is relatively the same as just combining the two filter's individual times because they are both run sequentially. Nonetheless, these results show that by sacrificing approximately 1/3 of David Mason's [34] filter execution time, a considerably better settlement can be generated.

Chapter 5

Conclusion & Future Work

5.1 Overall Conclusion

Without a doubt, generating a more lifelike road system has shown big potential when it comes to improving the overall quality of a produced settlement in Minecraft [48]. The more realistic road curves, thickness, and structure, proved to make the generated settlement be more human-like, which is one of the main goals of the GDMC competition [26] and this paper. In addition, with the help of AI and WGANs [41], any type of road system can be generated, which makes the project flexible enough for further research.

Nonetheless, the created filter is a combination of AI and normal algorithms that are combined to produce a road system that can be adapted to any environment. Although the WGAN [41] model is not entirely responsible for the entire final road infrastructure, the filter is still fully autonomous. The predefined parameters can be also tuned for any user preferences. However, they should not drastically change the generated settlement.

On top of what has been already said, the filter can be also used either individually or as a support to another filter. Being that flexible, the project can be easily extended and featured in future research projects. Despite its drawbacks, described in chapters 3 and 4, the project improved the overall quality of an already highly ranked GDMC filter [26]. This, in itself, proves that improving the road infrastructure benefits the final settlement significantly. Therefore, further research in road generation and Conditional WGANs [8] would be a good start for a future fully autonomous AI filter.

5.2 Future Work

As mentioned before, this project is open for further research. Many aspects can be improved and integrated better. Future research path that can be taken is to make the WGAN [41] model learn more complex road systems. A possible solution to that would be to use different GAN [17] architectures. Additionally, more complex data can be utilised. For example, datasets with predefined house slots can be used to make the building placement easier and more realistic. Moreover, a conditional GAN [8] network can be used to make the filter more adaptable. If different road labels are assigned, the model can be learnt to generate roads that are specifically designed for the selected environment. An example of that would be generating a forest road because the current Minecraft [48] biome is a forest. This would result in smaller and highly curved roads, instead of big industrial ones. Such an approach was tried, however, because of insignificant data, the model was never able to learn to generate images with complex labeling. Nonetheless, the results show a possibility for further research and improvement. They can be observed in figure 5.1, as a reference. However, they should not be taken as definitive. Nonetheless, getting more data and improving the model's design might fix the opposed issues.

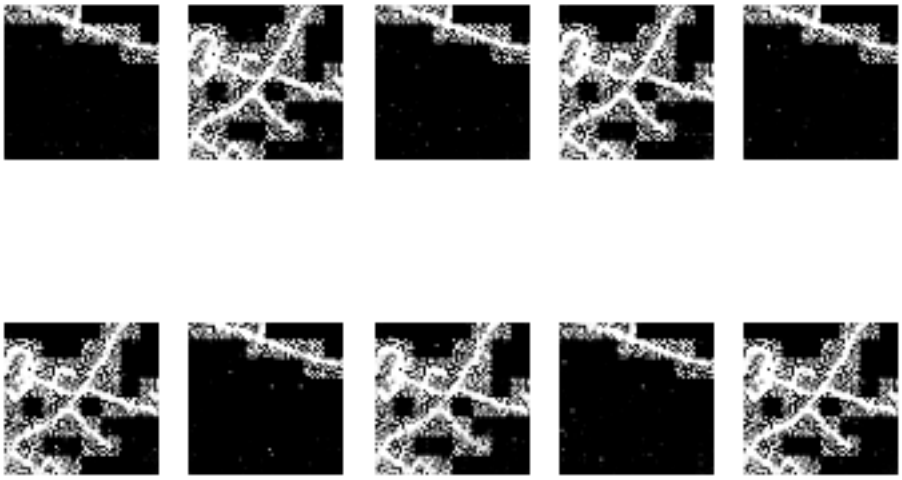


Figure 5.1: This figure shows the generated road images of a Conditional WGAN [8] after training for around 15 000 epochs, with 2 classes (small and big road). The model experiences Mode Collapse [41] and would not produce any new images. Rather, they would stay unchanged with the noise around the roads still present.

Another possible future improvement path could be the classical algorithms present in the filter. For example, A-star [51] has been used for connecting roads and houses to roads. However, this algorithm is slower compared to other non-optimal algorithms. The optimality is not entirely necessary, as sometimes, roads are not built along the shortest distance line. Moreover, the ability of the filter to deal with obstacles can be significantly improved. Currently, the filter cannot, with 100% accuracy, identify trees that are obscuring the road. This can be remedied with the help of a CNN [28] or DNN [44] that can categorize trees. In addition, tunnels and bridges can be built better by researching how and where these structures are usually placed in real-world settings. This can improve the aesthetics of the generated settlement, making it more livable. Furthermore, it was previously mentioned that the filter does not build the houses on its own. Thus, it must be noted, that by using a better filter for settlement generation, the filter described in this paper would exceed.

To conclude, this project shows a Minecraft [48] settlement generation filter that produces a lifelike road system. On top of that, the filter builds houses with the help of other filters. As there have not been many road generation approaches for the GDMC [26] competition, this project can be considered a decent basis for further research in this area.

Chapter 6

Legal, Social, Ethical and Professional Issues

This project is built to be used as a Minecraft [48] filter and as part of the GDMC [26] competition. Nevertheless, the project consists of a software code portion. Therefore, it was designed in a way to abide by the British Computer Society code of conduct [43]. Furthermore, the following sections cover any other legal, professional, and ethical issues that may be in effect due to this project.

6.1 Plagiarism & Originality

The created project has utilised multiple code libraries and code sources. Without them, the project would not have been possible. These libraries have been legally permitted to be used in an open-source environment and can be found in the appendices of this paper. In addition, local and cloud IDEs have been utilised, such as Google Colab [4]. Google Colab [4] provides its users with a free runtime environment for training AI models. Thus, this free membership option has been used by this project.

Furthermore, the A-star [51] algorithm used in the created filter, has been entirely taken from [3], under the General Public Licence, which states that everyone is permitted to copy and distribute the code [23]. Other sources have also been utilised and have influenced the outcome of this project. For the creation of the model, the [7] blog has been studied. Nonetheless, every source, that has influenced this paper and project in any way, has been accordingly referenced. Additionally, this project should not be lightly used for commercial use, nor any of its separate

parts, unless specifically agreed upon. However, it is open-source and free for personal use. Further research of this project is also highly encouraged, given the proper reference style is followed.

6.2 Risks & Software Competence

The codebase, as well as the created AI model, have been tested for errors and bugs. Library errors may occur because of version incompatibility. These errors should not appear unless either they have been deemed unsupported by their author or the wrong Python version is being used. However, the libraries on themselves should not cause any instability issues of the project. In addition, exceptions directly from MCEdit [25] might be thrown and can be due to errors either in the platform itself or in the created filter. Nevertheless, this project tries to account for such exceptions, giving a detailed explanation of why they might have occurred.

6.3 AI Concerning Ethics

The WGAN [41] model, described in this paper, is an Artificial Intelligence system, thus, ethical issues might occur. Therefore, the AI system is not to be used for any type of malicious purpose, that is directly and indirectly. Its solemn purpose is to be used as a Minecraft [48] filter and to inspire further research work in the field.

Moreover, the model might suffer from bias if it is to be adapted for other datasets. That could cause various ethical issues and concerns, so the model's bias needs to be properly accounted for. In addition, the quality of the generated images is not to be trusted and deemed 100% accurate. The results are open for personal interpretation and are not definitively marked as perfect road representations.

To conclude, the model is a valid Minecraft [48] and GDMC [26] filter solution and its purpose is to serve as such. Any other usage of the created AI is not recommended. Such an example is using the model to generate or to support the generation of roads for real-world use. This might cause disruptions of the whole city infrastructure or other problems. Therefore, this project is only to be used for the stated purpose in this paper.

References

[1] KDnuggets Al Gharakhanian. Generative adversarial networks – hot topic in machine learning. https://www.kdnuggets.com/2017/01/generative-adversarial-networks-hot-topic-machine-learning.html, last checked on 11-06-2021.

[2] Analytics India Magazine Anirudh VK. Evolution of ai in video games: Transition from cheating to 'smarts'. https://analyticsindiamag.com/evolution-of-ai-in-video-games-transition-from-cheating-to-smarts/, last checked on 11-06-2021.

[3] Baijayanta Roy at Medium-Article. A star algorithm code. https://github.com/BaijayantaRoy/Medium-Article, last checked on 10-06-2021.

[4] Ekaba Bisong. Google colaboratory. In *Building Machine Learning and Deep Learning Models on Google Cloud Platform*, pages 59–64. Springer, 2019.

[5] Brilliant.org. Backpropagation. Retrieved 16:39, August 13, 2021, from https://brilliant.org/wiki/backpropagation/.

[6] Jason Brownlee. How to avoid exploding gradients with gradient clipping. https://machinelearningmastery.com/how-to-avoid-exploding-gradients-in-neural-networks-with-gradient-clipping/, last checked on 12-08-2021.

[7] Jason Brownlee. How to develop a wasserstein generative adversarial network (wgan) from scratch. https://machinelearningmastery.com/how-to-code-a-wasserstein-generative-adversarial-network-wgan-from-scratch/, last checked on 10-06-2021.

[8] Yue Cao, Bin Liu, Mingsheng Long, and Jianmin Wang. Hashgan: Deep learning to hash with pair conditional wasserstein gan. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1287–1296, 2018.

[9] Paul Dean. A review of the game banished, by shining rock software. `https://www.eurogamer.net/articles/2014-02-25-banished-review`, last checked on 30-07-2021.

[10] Ilke Demir, Krzysztof Koperski, David Lindenbaum, Guan Pang, Jing Huang, Saikat Basu, Forest Hughes, Devis Tuia, and Ramesh Raskar. Deepglobe 2018: A challenge to parse the earth through satellite images. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2018.

[11] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[12] OpenCV Documentation. Morphological transformations. `https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_morphological_ops/py_morphological_ops.html`, last checked on 14-08-2021.

[13] Ding-Zhu Du and Panos M Pardalos. *Minimax and applications*, volume 4. Springer Science & Business Media, 2013.

[14] Rohit Dwivedi. Everything you should know about dropouts and batchnormalization in cnn. `https://analyticsindiamag.com/everything-you-should-know-about-dropouts-and-batchnormalization-in-cnn/`, last checked on 12-08-2021.

[15] Shreyas Fadnavis. Image interpolation techniques in digital image processing: an overview. *International Journal of Engineering Research and Applications*, 4(10):70–73, 2014.

[16] Edoardo Giacomello, Pier Luca Lanzi, and Daniele Loiacono. Doom level generation using generative adversarial networks. In *2018 IEEE Games, Entertainment, Media Conference (GEM)*, pages 316–323. IEEE, 2018.

[17] Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *arXiv preprint arXiv:1406.2661*, 2014.

[18] Zekun Hao, Arun Mallya, Serge Belongie, and Ming-Yu Liu. Gancraft: Unsupervised 3d neural rendering of minecraft worlds. *arXiv preprint arXiv:2104.07659*, 2021.

[19] Yash Joshi. Connected component labeling. `https://iq.opengenus.org/connected-component-labeling/`, last checked on 14-08-2021.

[20] George Kelly and Hugh McCabe. Citygen: An interactive system for procedural city generation. In *Fifth International Conference on Game Design and Technology*, pages 8–16, 2007.

[21] Pooya Khorrami, Thomas Le Paine, and Thomas S. Huang. Do deep neural networks learn facial action units when doing expression recognition? In *Proceedings of the IEEE International Conference on Computer Vision (ICCV) Workshops*, December 2015.

[22] Will Koehrsen. Neural network embeddings explained. `https://towardsdatascience.com/neural-network-embeddings-explained-4d028e6f0526`, last checked on 15-08-2021.

[23] GNU General Public License. Gnu general public license. *Retrieved December*, 25:2014, 1989.

[24] Ivan Marović, Ivica Androjić, Nikša Jajac, and Tomáš Hanák. Urban road infrastructure maintenance planning with application of neural networks. *Complexity*, 2018, 2018.

[25] MCEdit. Mcedit: World editor for minecraft. `https://www.mcedit.net/`, last checked on 11-06-2021.

[26] Rodrigo Canaan Christian Guckelsberger Julian Togelius Michael Cerny Green, Christoph Salge. Generative design in minecraft (gdmc) competition. `https://gendesignmc.engineering.nyu.edu/`, last checked on 11-06-2021.

[27] Rodrigo Canaan Christian Guckelsberger Julian Togelius Michael Cerny Green, Christoph Salge. Generative design in minecraft (gdmc) settlement generation. `https://gendesignmc.wikidot.com/wiki:settlement-generation-competition`, last checked on 11-06-2021.

[28] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[29] Amit Patel. Introduction to a star. http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html, last checked on 11-06-2021.

[30] William L Raffe, Fabio Zambetta, and Xiaodong Li. A survey of procedural terrain generation techniques using evolutionary algorithms. In *2012 IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2012.

[31] Benjamin Recht, Rebecca Roelofs, Ludwig Schmidt, and Vaishaal Shankar. Do cifar-10 classifiers generalize to cifar-10? *arXiv preprint arXiv:1806.00451*, 2018.

[32] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[33] Vaibhav Sahu. Power of a single neuron. https://towardsdatascience.com/power-of-a-single-neuron-perceptron-c418ba445095, last checked on 01-08-2021.

[34] Christoph Salge, Claus Aranha, Adrian Brightmoore, Sean Butler, Rodrigo Canaan, Michael Cook, Michael Cerny Green, Hagen Fischer, Christian Guckelsberger, Jupiter Hadley, et al. Impressions of the gdmc ai settlement generation challenge in minecraft. *arXiv preprint arXiv:2108.02955*, 2021.

[35] Christoph Salge, Michael Cerny Green, Rodrigo Canaan, Filip Skwarski, Rafael Fritsch, Adrian Brightmoore, Shaofang Ye, Changxing Cao, and Julian Togelius. The ai settlement generation challenge in minecraft: First year report. *arXiv preprint arXiv:2103.14950*, 2021.

[36] Warren S Sarle. Neural networks and statistical models. 1994.

[37] Sagar Sharma. Activation functions in neural networks. https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6, last checked on 12-08-2021.

[38] Simplilearn. What is image processing : Overview, applications, benefits, and who should learn it. https://www.simplilearn.com/image-processing-article, last checked on 14-08-2021.

[39] Harmanjit Singh and Richa Sharma. Role of adjacency matrix & adjacency list in graph theory. *International Journal of Computers & Technology*, 3(1):179–183, 2012.

[40] Sagar B Tambe, Deepak Kulhare, MD Nirmal, and Gopal Prajapati. Image processing (ip) through erosion and dilation methods. 2013.

[41] Lilian Weng. From gan to wgan. *arXiv preprint arXiv:1904.08994*, 2019.

[42] Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.

[43] Su White. Comp1205 lepp: Bcs code of conduct. 2015.

[44] Wikipedia. Artificial neural network. `https://en.wikipedia.org/wiki/Artificial_neural_network`, last checked on 07-08-2021.

[45] Wikipedia. Bio-inspired computing. `https://en.wikipedia.org/wiki/Bio-inspired_computing`, last checked on 11-06-2021.

[46] Wikipedia. Computer vision. `https://en.wikipedia.org/wiki/Computer_vision`, last checked on 14-08-2021.

[47] Wikipedia. Gradient descent. `https://en.wikipedia.org/wiki/Gradient_descent`, last checked on 01-08-2021.

[48] Wikipedia. Minecraft, sandbox video game developed by mojang. `https://en.wikipedia.org/wiki/Minecraft`, last checked on 11-06-2021.

[49] Wikipedia. Nim, mathematical game of strategy. `https://en.wikipedia.org/wiki/Nim`, last checked on 11-06-2021.

[50] Wikipedia. Procedural content generation. `https://en.wikipedia.org/wiki/Procedural_generation`, last checked on 11-06-2021.

[51] Wikipedia. A star search algorithm. `https://en.wikipedia.org/wiki/A*_search_algorithm`, last checked on 11-06-2021.

[52] Xue Ying. An overview of overfitting and its solutions. In *Journal of Physics: Conference Series*, volume 1168, page 022022. IOP Publishing, 2019.

[53] Xinjie Yu and Mitsuo Gen. *Introduction to evolutionary algorithms*. Springer Science & Business Media, 2010.

Appendix A

Additional Proof Samples

A.1 Road Generation

Figure A.1: This figure shows two additional generated roads by the proposed filter. Both images represent the roads on a flat surface.

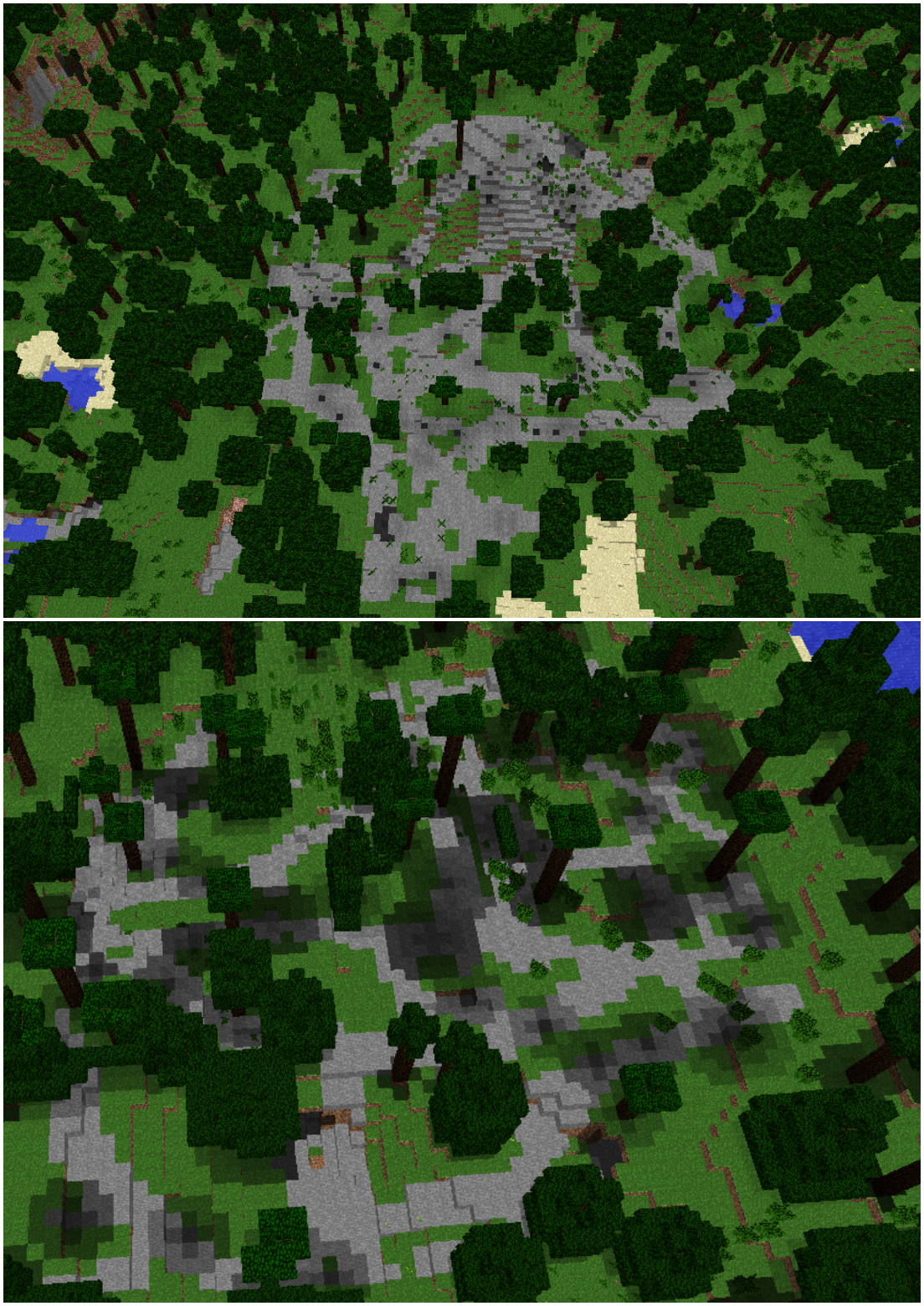


Figure A.2: The two depicted images show a generated road on uneven surfaces.

A.2 Settlement Generation

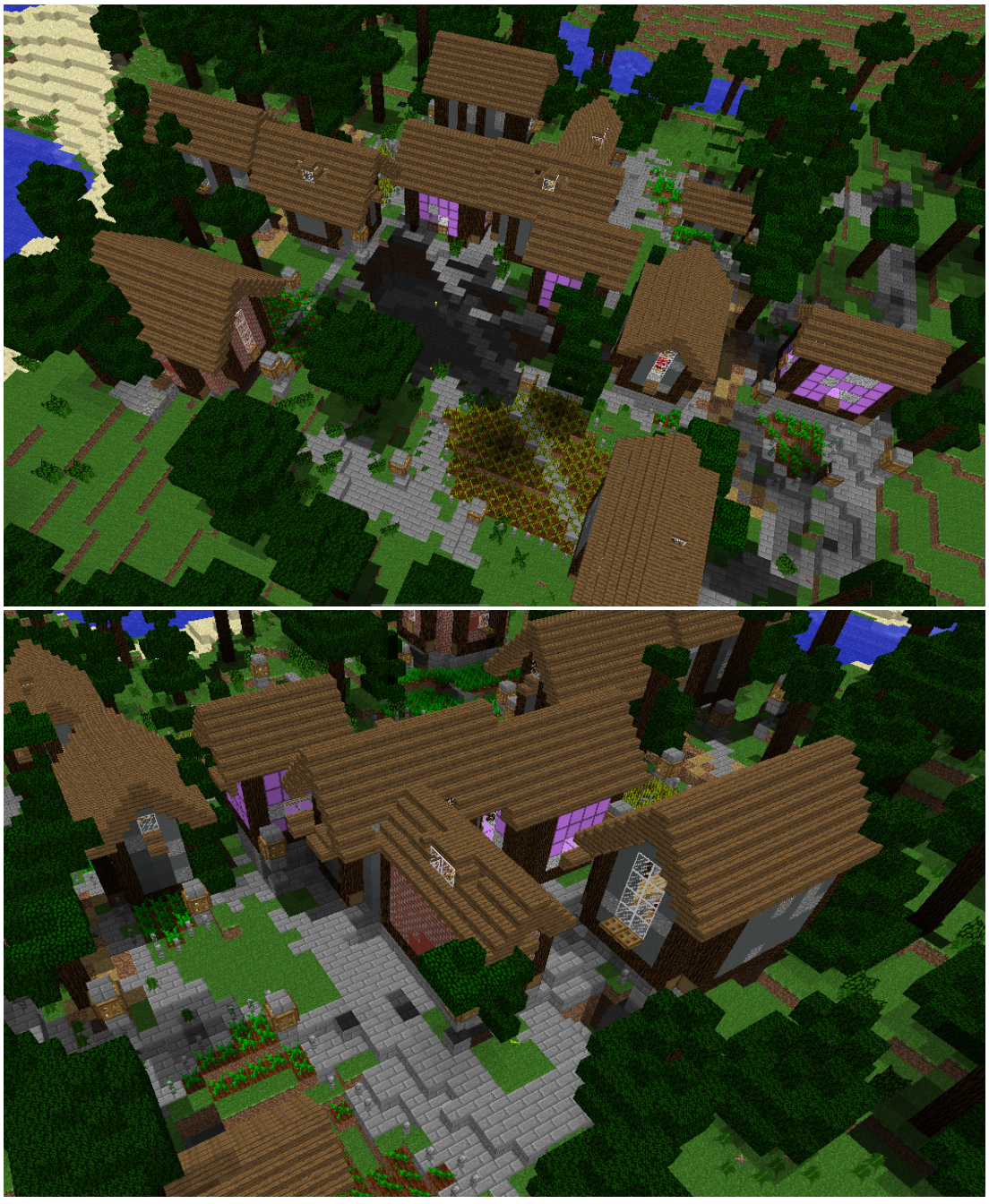


Figure A.3: The figure depicts two generated settlements on an uneven surface. The settlements have been generated using the combined Road Filter and David Mason's filter [34].

Figure A.4: This figure shows more generated settlements by the combined filter. The first image is on an uneven surface, where the second is on a flat surface.

A.3 Filter Limitations

(a)

(b)

Figure A.5: Image a) shows the combined filter generating a settlement on top of a water body. Image b) depicts the same settlement generated by David Mason's [34] filter. The figure shows the limitations of David Mason's filter [34], where it cannot differentiate between water and land.

Appendix B

User Guide

B.1 MCEdit Download Instructions

To run the project and recreate its results, MCEdit [25] needs to be installed. First, the platform must be downloaded by using - *git clone –recursive https://github.com/mcgreentn/GDMC*. It can also be found at `https://github.com/mcgreentn/GDMC`. This command will download MCEdit [25]. All the required packages are part of the provided *requirements.txt* file in the main folder of the project described by this paper. They can be installed by following the instructions in the next section. The official guide can be found at `https://github.com/mcgreentn/GDMC/wiki/The-GDMC-Framework` or at `https://gendesignmc.wikidot.com/wiki:submission-mcedit`. Everything needed is covered in the next section, thus, it can be ignored. This is, however, with the exception of if the used operating system is not Windows. If it is MacOS, then the additional provided system-specific steps, under **Mac only section**, need to be followed.

B.2 Project & MCEdit Installation Instructions

All project and MCEdit [25] required libraries can be found under the *"requirements.txt"* file. They can be easily installed by following the described steps. Furthermore, the project has been made on Windows Operating System, thus, the described steps are for that particular system. Nonetheless, links to installation guides for other systems have been made available.

First, **Python 2.7** needs to be present on the machine. If the **virtualenv** package is not present, it is advisable that it is installed from `https://virtualenv.pypa.io/en/latest/installation.html` or by running - *py -2.7 -m pip install virtualenv*, for Windows. Other-

wise, an environment can be created and activated. After navigating to the projects' main folder, an environment can be created by running - *py -2.7 -m virtualenv venv*, for Windows, or check `https://virtualenv.pypa.io/en/latest/user_guide.html#introduction` for more information. After the environment has been created, one can activate it by running the Windows command - *venv\Scripts\activate*, or by looking at the main documentation at `https://virtualenv.pypa.io/en/latest/user_guide.html#activators`. The activated environment should be shown as in figure B.1, depending on the system.

Figure B.1: A Windows Command Line representation after activating the *venv* virtual environment.

The second step, after activating the virtual environment, is to install all required packages. The command - *pip install -r requirements.txt* can be used to easily install all listed libraries. Because the **Tensorflow Library** is deprecated for the current Python version, it must be installed separately from a wheel file. In the main project folder, there are three TensorFlow 1.10 **.whl** files for every type of system. Moreover, they can be downloaded from the following links:

- Tensorflow for Windows - `https://github.com/fo40225/tensorflow-windows-wheel/tree/master/1.10.0/py27/CPU/avx2`.
- Tensorflow for MacOS and Linux - `https://pypi.org/project/tensorflow/1.10.0/#files` and navigate to **Download files**.

To install the package, select the correct file for your system and substitute it in the following command - *pip install "tensorflow .whl filename"*. This should install the library into your virtual environment. An example of these commands can be seen in figure B.2.

(a)

(b)

Figure B.2: This figure shows the two commands needed to install all required project libraries. It must be noted that the second command b) installs the Window's TensorFlow version. On the other hand, command a) installs all listed libraries from the specified file.

Moreover, a complete list of all the used libraries can be found below:

- numpy==1.14.5
- keras==2.2.0
- scipy
- scikit-learn
- PyOpenGL
- pygame==1.9.4
- pyyaml
- pillow
- ftputil==3.4
- pypiwin32
- matplotlib
- opencv-contrib-python
- Augmentor
- Tensorflow==1.10.0

B.3 User Instructions

This section goes over how to run all the files present and how to replicate (relatively) the results covered in a previous chapter. Each of the following subsections describes how to run a specific file.

B.3.1 Running finalwgan.py

This file contains the design of the WGAN [41] network and can be used to train the model. This should be done in Google Colab [4] with its GPU accelerator utilised because that is what has been used in this paper. In addition, the previously created virtual environment will not suffice because the model has been trained using **Python 3**. Thus, Google Colab [4] provides all the necessary libraries to train the model. Moreover, the only path that the file needs is

the path to the training data. Currently, the given path is as *"train/"*, which states that the dataset is named as *"train"* and is located at the current root directory. To change the path to the dataset, the line of code in figure B.3 must be changed.

```
204   # Train WGAN
205   model = WGAN()
206   model.train("train/")  # Change the parameter depending on the path
```

Figure B.3: The figure shows the exact line of code that must be changed to accommodate different dataset paths.

The dataset can be acquired from `https://emckclac-my.sharepoint.com/:f:/g/personal/k1763856_kcl_ac_uk/ErB57uMLQTtPrcj7n33KzKEBUraXryiZRG3Kv1BghBKc4g?e=6XUCS4`. Here, it can be only downloaded and immediately used. Alternatively, it can be downloaded from `https://www.kaggle.com/balraj98/deepglobe-road-extraction-dataset?select=train`. However, only the mask images from the **train** folder are taken. Therefore, this method requires further work to be done before usage. The original dataset paper [10] can be found in the references.

At the end of the training process, a new model plot history will be saved, alongside the trained model as *"trained_model.h5"*. In addition, the training loss and accuracy will be shown for every epoch.

B.3.2 Running convert_model.py

In the Convert folder, a *"model.h5"* file can be found. It is an already trained model that is there for testing purposes. This file, as well as any other in the project folder, has to be run with the virtual environment activated. The *"convert_model.py"* file can be executed without changing any parameters. It will convert the aforementioned model to a **Python 2.7** version one. To change the path to the selected model for conversion, the line in figure B.4 must be changed with the new path. Moreover, the same figure B.4 shows which exact line to uncomment if the user would want to generate an image and test the converted model. It must be also noted that an image plot might not be generated, if testing, because of issues with the virtual environment. To fix that, from the original **Python 2.7** folder, the **tcl** folder needs to be copied and placed in the virtual environment directory.

```
78    convert_model("model.h5")
79    # test_converted_model()
```

Figure B.4: The first line in the image shows the position which must be changed if a different model path is to be used. On the other hand, the second line must not be commented if the user would like to test the model.

The converted model has a predefined filename (*"converted_model.h5"*). This newly converted model can be used to generate road images. The file can be run either from IDE or by executing the following command from the project's root directory - *py Convert\convert_model.py*.

B.3.3 Running convert_images.py

This file has been used for preprocessing the dataset before using it for training. The file has two adjustable parameters. The first one is *out_path = "out/"*, which specifies the output folder of the newly converted images. If not changed, the *"out"* folder must be created, as otherwise, the code will throw an error. The second parameter is *in_path = "train/"*, which specifies the path to the dataset. Again, if not changed, the dataset must be present in the current directory under the name *"train"*. The file can be run either from an IDE or from Command Line, using the Windows command from the project's root directory - *py Convert\convert_images.py*.

B.3.4 Running convert_images - Conditional.py

This file has been used when testing a Conditional WGAN [8]. It converts a dataset into a categorical one. There are two predefined classes (small and big roads). The conversion is done based on road coverage, as per chapter 3. In addition, augmentation of the data is present as a possible method to use. It also has the same two parameters as in the previous subsection. The lines that can be changed are shown in figure B.5. Furthermore, the file can be run either from an IDE or by executing the command - *py "Convert\convert_images - Conditional.py"*.

```
# 1st) Augment_data
# augment_data(in_path[:-1])  # Remove last \
# 2nd) Convert data
# Do only after moving output folder to main folder after augmentation
convert_data(in_path, out_path, labels)
```

Figure B.5: The figure shows the code lines that can be changed before running the file. The first non-commented code line can be uncommented to run augmentation on the data. The data will be automatically put into a different folder and must be transferred to the main dataset folder. The last code line can be edited to accommodate for different **"in"** and **"out"** paths.

B.3.5 Running MCEdit

After downloading and installing MCEdit [25], as per previous sections, one must know how to navigate and run filters on the platform. As it is a 3^{rd} party platform, the following guidelines can be followed - `https://github.com/mcgreentn/GDMC/wiki/Using-MCEdit`. Moreover, the GDMC wiki [27] can be referenced for more information - `https://gendesignmc.wikidot.com/wiki:submission-mcedit`.

B.3.6 Running Filters

This section shows the reader how to run the main portion of the project. First and foremost, the **"Filters"** folder must be put inside the newly downloaded MCEdit [25] directory. This should be done under **"stock-filters"**, where all filters are originally located. It must be noted that the original David Mason filter [34] is not present, as for plagiarism purposes. It can be downloaded and put inside the **"stock-filters"** folder from `https://drive.google.com/file/d/1lVUHBibnWSWRdyLsIgqpO_rcrB6et5hn/view`.

To begin with, the **"Fitlers"** folder contains four separate files. These files can be seen in the tree below:

```
Filters
├── A_star.py
├── RoadFilter_WGAN.py
├── David Mason RF.py
└── RoadWGAN.h5
```

The first file is the A-star [51] algorithm file. It is not an original codebase. It was taken from `https://github.com/BaijayantaRoy/Medium-Article` [3] under a free licence. Moreover, it does not require any separate execution. The code is automatically run from the main **RoadFilter_WGAN.py** file.

The **RoadWGAN.h5** file is the trained model and it does not require any separate execution either. It is also automatically run by **RoadFilter_WGAN.py**.

The first filter file is the **RoadFilter_WGAN.py** and it contains the main code base of this project. It generates a road system and edits the environment respectively. To be executed as an individual filter, it can be simply called from MCEdit's [25] filter menu on a selected region, by choosing the filter's name. Although it is an autonomous filter, there are two parameters that can be adjusted. They both affect the generated image and different values can be tested. The first parameter is the *n_maps=2*, which decides how many images to combine, as per

chapter 3. The second one is *n_imgs=25*, and it specifies the number of generated road images.

The second filter file, **David Mason RF.py**, is the combination of the proposed filter and David Mason's original filter [34]. It can be executed in the same way as the aforementioned road filter, from MCEdit's [25] filter menu. It generates a road system and builds houses on top of it. Figure B.6 shows how exactly the proposed road filter is incorporated in the original David Mason filter.

```
2314    # Added code to connect the RoadFilter_WGAN Filter.
2315    block = choice([(98,0),(98,0),(98,0),(98,0),(1,5),(4,0),(13,0),(98,1),(98,2)])
2316    road = rf.perform(level, box, options, block, True)
```

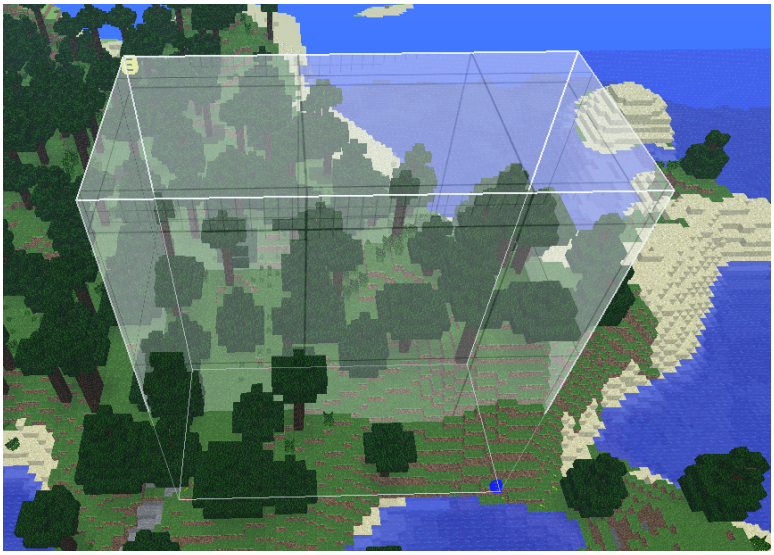
(a)

```
2679        # Added code to connect the RoadFilter_WGAN Filter.
2680        rf.connect_houses_to_roads(level, house.door_pos, road, [(1,6),(4,0),(43,0),(43,5)])
```

(b)

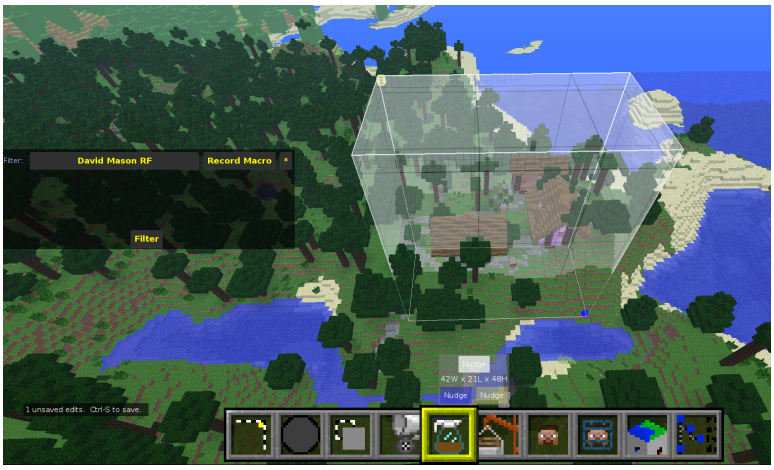
Figure B.6: The figure shows the exact and only lines of code that were added to combine both filters. In addition, on line 16, the filter has been imported via - *import RoadFilter_WGAN as rf*. Image a) shows the code that was added to generate the road. Image b) shows the code that was added for connecting disconnected houses. Moreover, lines 2263 - 2313 have been commented. This is the code where David Mason's filter [34] uses to generate its roads.

In addition, a step-by-step tutorial is shown in figure B.7. It must be also noted that the selected regions in MCEdit [25] need to cover perfectly the bottom floor, without any air blocks present. This is a limitation of David Mason's filter [34], where it does not recognise air blocks and the code breaks. Furthermore, because of the same reasons, the height of the selected region must be made considerably tall.



(a)

(b)



(c)

Figure B.7: This is a figure that serves as a guide to executing a filter in MCEdit [25]. Image a) shows that the first step is selecting a region. The region specifications are covered in the paragraph before the image. Image b) has 3 highlighted steps (in red). The first step is selecting the filter menu, then selecting the filter to run, and finally, executing the filter. The name of the filter may differ from the one shown in the image, depending on the directory. The final image c) shows the results of running the selected filter.

B.3.7 Reconstructing Results

By running the previously mentioned filters, in addition to the original David Mason filter [34], the results in chapter 4 can be reconstructed. Moreover, these filters should be executed on the provided **"TestWorld"** Minecraft [48] world. This exact world has been used for getting all results in chapter 4. It can be loaded from the main menu in MCEdit [25], as it can be seen in figure B.8. Nonetheless, the shown results cannot be fully recreated because of the specificity of the selected regions. There is no way to transfer the exact positions of the selected regions, thus, this must be done by observation.

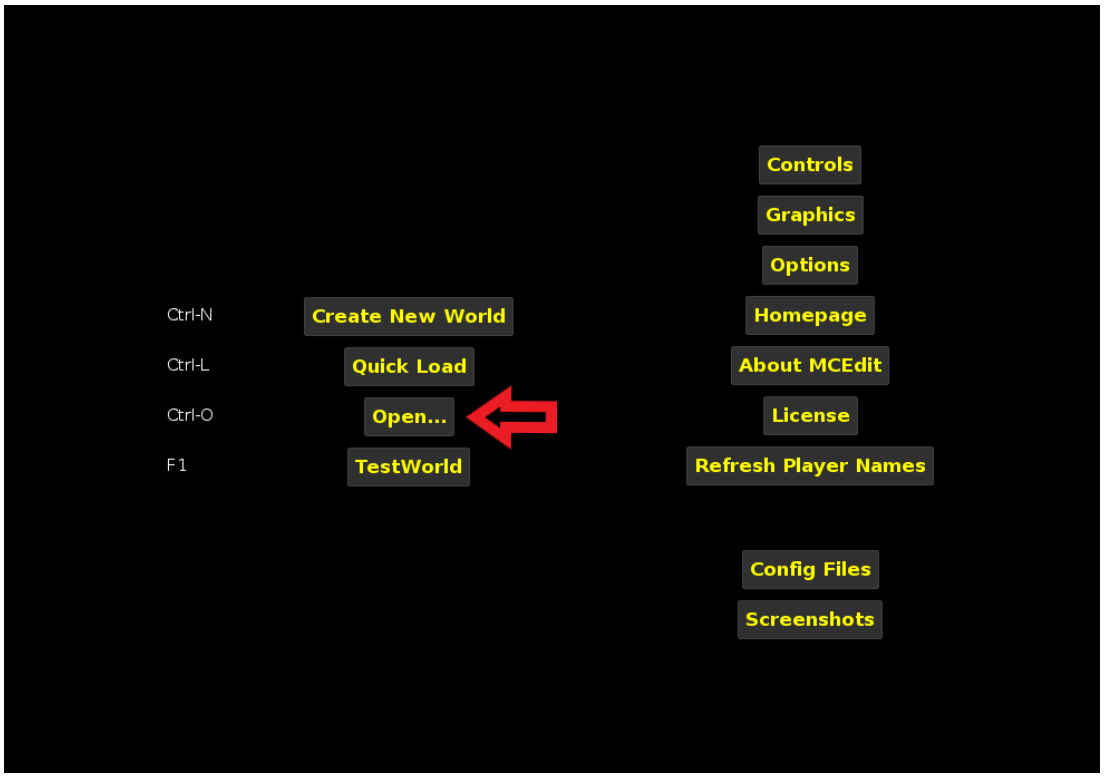


Figure B.8: This image shows the MCEdit [25] Open menu, which loads worlds from a folder.

Appendix C

Source Code

C.1 Originality Avowal

"I verify that I am the sole author of the programs contained in this folder, except where explicitly stated to the contrary."

Preslav Kisyov

August 22, 2021

C.2 Project Structure

- Main Project Folder — The root folder
  - Convert — The folder with all conversion files
    - "convert_images - Conditional.py" — Converts a dataset to conditional
    - "convert_images.py" — Data preprocessing
    - "convert_model.py" — Converts a model to Python 2
    - "finalwgan.py" — Specifies the design of the AI model
    - "model.h5" — A pretrained model
  - Filters — The folder with all filters
    - "A_star.py" — Used for connecting entities
    - "David Mason RF.py" — The combined filter
    - "RoadFilter_WGAN.py" — The proposed road filter
    - "RoadWGAN.h5" — A pretrained model
  - TestWorld — The world used for testing
  - README.pdf — Provides the structure of the archive
  - UserGuide.pdf — The installation and usage instructions
  - OriginalityAvowal.pdf — Originality statement
  - requirements.txt — A list of Python libraries
  - tensorflow-1.10.0-cp27-cp27m-macosx_10_11_x86_64.whl . Tensorflow for Mac
  - tensorflow-1.10.0-cp27-cp27mu-manylinux1_x86_64.whl Tensorflow for Linux
  - tensorflow-1.10.0-cp27-cp27m-win_amd64.whl — Tensorflow for Windows

C.3 convert_images - Conditional.py

```
from PIL import Image
import os
import numpy as np
import sys
import Augmentor

"""
This file converts a dataset of images into different size and labels
used for preprocessing a Conditional WGAN Dataset
"""

# Defined variables with paths
out_path = "out/"
in_path = "train/"
labels = ["_small", "_big"]
img_format = ".png"  # might need to be changed depending on the dataset

# This method converts every image from a folder
# into a 64x64 Binary Image and assigns a label
# given the percentage of road in it.
# @param in_path The path to the images
# @param out_path The path to the output folder
# @param labaels The labels for the images
# @param pct The percentage that decides the labels
def convert_data(in_path, out_path, labels, pct=35):
    counter = 1
    print("Converting...")
    for im_path in os.listdir(in_path):
        if img_format not in im_path: continue
        # os.rename(in_path+im_path, str(counter)+".jpg")  # if renaming is
        ↪ needed
        img = Image.open(in_path+im_path)
```

```
32          np.set_printoptions(threshold=sys.maxsize)
33          img = np.array(img.resize((64, 64), Image.BILINEAR))
34          count = np.count_nonzero(img)
35          pct_count = (count/float(img.shape[0]*img.shape[1])) * 100.0  # Get
            ↪ percentage of road
36          img = Image.fromarray(np.uint8(np.where(np.array(img) > 0, 1, 0)*255))
37          if pct_count > pct: img.save(out_path + str(counter) + labels[1] +
            ↪ ".jpg")
38          else: img.save(out_path + str(counter) + labels[0] + ".jpg")
39          counter += 1
40      print("Finished!")
41
42  # This method augments the data
43  # by rotating and zooming
44  # @param path The path to the images
45  def augment_data(path):
46      print("Augmenting...")
47      augmentor = Augmentor.Pipeline(path)
48      augmentor.rotate90(1)
49      augmentor.process()
50      augmentor.rotate270(1)
51      augmentor.process()
52      print("Finished!")
53
54  # 1st) Augment_data
55  # augment_data(in_path[:-1])  # Remove last \
56  # 2nd) Convert data
57  # Do only after moving output folder to main folder after augmentation
58  convert_data(in_path, out_path, labels)
```

C.4 convert_images.py

```
from PIL import Image, ImageEnhance, ImageFilter, ImageOps
import os
import cv2
import numpy as np
import sys

"""
This file converts a Dataset of images
into a Dataset of 64x64 jpg images ready
to be used for WGAN training
"""

# Defined paths and counter
out_path = "out/"
in_path = "train/"
counter = 1

# The function that reformats all images
print("Converting...")
for im_path in os.listdir(in_path):
    if ".png" not in im_path: continue
    img = Image.open(in_path+im_path)
    im = img.resize((64, 64), Image.BILINEAR)

    im = Image.fromarray(np.uint8(np.where(np.array(im) > 0, 1, 0)*255))
    im.save(out_path+str(counter)+".jpg")
    counter += 1
print("Finished!")
```

C.5 convert_model.py

```
from PIL import ImageOps
from PIL import Image
import numpy as np
from keras.datasets import mnist
from keras import backend
from keras.optimizers import RMSprop
from keras.models import Sequential
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 LeakyReLU
from keras.layers import BatchNormalization, Dropout, UpSampling2D,
    Activation, ZeroPadding2D
from keras.constraints import Constraint
from keras.initializers import RandomNormal
from matplotlib import pyplot
import keras

# This file converts a newer version of keras model
# to an older version. Note, the model architecture needs
# to be known, only the weights are being loaded.

# Defines the model.
# @return The generator model
def create_model():
    generator = Sequential([
        Dense(128*8*8, input_dim=100),
        Reshape((8, 8, 128)),
        # upsample 2x
        Conv2DTranspose(128, kernel_size=4, strides=2, padding="same"),
```

```
32          BatchNormalization(),
33          LeakyReLU(alpha=0.2),
34          # upsample 2x
35          Conv2DTranspose(128, kernel_size=4, strides=2, padding="same"),
36          BatchNormalization(),
37          LeakyReLU(alpha=0.2),
38          Dropout(0.2),
39          # upsample 2x
40          Conv2DTranspose(128, kernel_size=4, strides=2, padding="same"),
41          BatchNormalization(),
42          LeakyReLU(alpha=0.2),
43          Dropout(0.2),
44          Conv2D(1, kernel_size=5, activation='tanh', padding='same')])
45      return generator
46
47  # Creates and converts a model to an older Keras version
48  # and saves it as h5.
49  # @param path The path of the model to be converted
50  # @param output_path The predefined output path of the converted model
51  def convert_model(path, output_path = "converted_model.h5"):
52      print("Starting conversion process...")
53      generator = create_model()
54      print("Created a model replica!")
55      generator.load_weights(path)
56      generator.save(output_path)
57      print("Conversion is done! New model is saved as %s!" % output_path)
58
59  # Loads the newly converted model
60  # and generates images to test it.
61  # param path The path of the model to be tested
62  def test_converted_model(path = "converted_model.h5"):
63      print("Loading model from %s..." % path)
64      generator = keras.models.load_model(path)
```

```
    print("Generating images...")
    noise = np.random.random(100 * 100).reshape(100, 100)
    img = generator.predict(noise)
    img = ((img + 1)/2.0)  # normalize images

    # Plots a 10 by 10 plot = 100 images
    for image in range(100):
        pyplot.subplot(10, 10, 1 + image)
        pyplot.axis('off')
        pyplot.imshow(img[image, :, :, 0], cmap='gray')
    print("Done!")
    pyplot.show()

convert_model("model.h5")
# test_converted_model()
```

C.6 finalwgan.py

```
# Used Python 3 as for Google Colab
import os
from PIL import ImageOps
from PIL import Image
import numpy as np
from keras import backend
from tensorflow.keras.optimizers import RMSprop
from keras.models import Sequential
from keras.layers import Reshape, Dense, Flatten, Conv2D, Conv2DTranspose
from keras.layers import BatchNormalization, Dropout, LeakyReLU
from keras.constraints import Constraint
from matplotlib import pyplot as plt

"""
This is the Weight Clipping class that is used
```

```
in the WGAN layers for Weight Clipping.

@author Preslav Kisyov, influenced by Jason Brownlee
->  https://machinelearningmastery.com/how-to-code-a-wasserstein-generative-adversarial-network-wg
"""
class WeightClip(Constraint):

    # The main class method
    #
    # @param c_value The clipping value
    def __init__(self, c_value): self.c_value = c_value

    # The method that is called when clipping
    #
    # @param clipped_weights The weights of the layer
    # @return The newly clipped weights
    def __call__(self, clipped_weights): return backend.clip(clipped_weights,
     -self.c_value, self.c_value)

    # The config method that defines
    # the name and clipping value name
    def get_config(self):
        return {'name': self.__class__.__name__,
                'c_value': self.c_value}

"""
This is the main model WGAN class
that loads data, defines models and trains them.

Note: The method that saves the model has been removed, so
it does not interfere with any already saved models. This
is only for recreational purposes.
```

```
47
48  @author Preslav Kisyov, influenced by Jason Brownlee
49  ->
    ↪  https://machinelearningmastery.com/how-to-code-a-wasserstein-generative-adversarial-network-wg
50  """
51  class WGAN:
52
53      # This is the main class method
54      # that defines all models and image shape
55      def __init__(self):
56          self.latent_dim = 100
57          self.img_shape = (64, 64, 1) # Define image shape
58          self.generator = self.create_generator()
59          self.critic = self.create_critic()
60          self.model = self.create_model()
61
62      # This methid defines the Generator
63      #
64      # @return generator The generator model
65      def create_generator(self):
66          generator = Sequential([
67              Dense(128*8*8, input_dim=self.latent_dim),
68              Reshape((8, 8, 128)),
69              # Upsample from 8x8 to 64x64
70              Conv2DTranspose(128, kernel_size=4, strides=2, padding="same"),
71              BatchNormalization(),
72              LeakyReLU(alpha=0.2),
73              Conv2DTranspose(128, kernel_size=4, strides=2, padding="same"),
74              BatchNormalization(),
75              LeakyReLU(alpha=0.2),
76              Dropout(0.2),
77              Conv2DTranspose(128, kernel_size=4, strides=2, padding="same"),
78              BatchNormalization(),
```

```
79              LeakyReLU(alpha=0.2),
80              Dropout(0.2),
81              # Get Output Image
82              Conv2D(self.img_shape[-1], kernel_size=5, activation='tanh',
                ↪  padding='same')])
83          generator.summary()
84          return generator
85
86      # This method defines the Critic/Descriminator
87      #
88      # @return critic The critic/descriminator model
89      def create_critic(self):
90          const = WeightClip(0.01)
91          critic = Sequential([
92              # Downsample from 64x64 to 8x8
93              Conv2D(128, kernel_size=4, strides=2, padding='same',
                ↪  kernel_constraint=const, input_shape=self.img_shape),
94              BatchNormalization(),
95              LeakyReLU(alpha=0.2),
96              Dropout(0.25),
97              Conv2D(128, kernel_size=4, strides=2, padding='same',
                ↪  kernel_constraint=const, input_shape=self.img_shape),
98              BatchNormalization(),
99              LeakyReLU(alpha=0.2),
100             Dropout(0.25),
101             Conv2D(128, kernel_size=4, strides=2, padding='same',
                ↪  kernel_constraint=const, input_shape=self.img_shape),
102             BatchNormalization(),
103             LeakyReLU(alpha=0.2),
104             Dropout(0.25),
105             # Get Output
106             Flatten(),
107             Dense(1)])
```

```
108         # Compile and return the Critic
109         critic.compile(loss = self.wasserstein_loss, optimizer =
            ↪  RMSprop(learning_rate=0.00005))
110         critic.summary()
111         return critic
112
113     # This method freezes all model layers,
114     # except for Batchnormalization
115     #
116     # @param model The model that has its layers frozen
117     def freeze_layers(self, model):
118         for l in model.layers:
119           if not isinstance(l, BatchNormalization): l.trainable = False
120
121     # This method defines the overall model
122     #
123     # @return model The overall keras model
124     def create_model(self):
125         self.freeze_layers(self.critic)
126         model = Sequential([
127             self.generator,
128             self.critic])
129         model.compile(loss = self.wasserstein_loss,
            ↪  optimizer=RMSprop(learning_rate=0.00005))
130         return model
131
132     # This is the main Train method that trains the Critic/Descriminator
133     # and the generator. At the end, it plots the losses.
134     #
135     # @param path The path to the dataset to be loaded
136     # @param critic_steps The amount of iterations for training the Critic
        ↪  every epoch
137     # @param training_epochs The amount of training epochs/iterations
```

```
138     # @param batch The batch size
139     def train(self, path, critic_steps = 5, training_epochs = 15000, batch =
        ↪  64):
140         final_critic_loss, final_generator_loss = list(), list()
141         data = self.load_data(path)
142         batch_samples = int(batch/2)
143 
144         # Train the Network
145         for epoch in range(training_epochs):
146             critic_loss = 0
147             # Train the Critic/Descriminator
148             for _ in range(critic_steps):
149                 # Pick random Real Sample and Noise
150                 real_image = data[np.random.randint(0, data.shape[0],
                    ↪  batch_samples)]
151                 fake_labels = np.ones((batch_samples, 1))
152                 real_labels = -np.ones((batch_samples, 1))
153                 # Pick random Fake Smaple and Noise
154                 noise = np.random.randn(batch_samples *
                    ↪  self.latent_dim).reshape(batch_samples, self.latent_dim)
155                 fake_image = self.generator.predict(noise)  # generator
                    ↪  predict
156                 # Update Critic on Real Image
157                 real_loss = self.critic.train_on_batch(real_image,
                    ↪  real_labels)  # train on real
158                 fake_lose = self.critic.train_on_batch(fake_image,
                    ↪  fake_labels)  # train on fake
159                 critic_loss = np.add(real_loss, fake_lose) / 2
160             # Train the Generator
161             noise = np.random.randn(batch * self.latent_dim).reshape(batch,
                ↪  self.latent_dim)
162             generator_loss = self.model.train_on_batch(noise, -np.ones((batch,
                ↪  1)))  # generator train
```

```
163             final_generator_loss.append(generator_loss)
164             final_critic_loss.append(critic_loss)
165             print('Epochs: %d/%d, Critic_Loss=%.3f Generator_Loss=%.3f' %
                ↪  (epoch+1, training_epochs, critic_loss, generator_loss))
166
167         self.plot_loss(final_critic_loss, final_generator_loss,
            ↪  training_epochs)
168         self.generator.save("trained_model.h5")
169
170     # This method plots the history of losses
171     #
172     # @param final_critic_loss, final_generator_loss All Generator and Critic
        ↪  losses
173     # @param epoch All epochs
174     def plot_loss(self, final_critic_loss, final_generator_loss, epoch):
175         plt.plot(final_critic_loss, label='Critic Loss')
176         plt.plot(final_generator_loss, label='Generator Loss')
177         plt.legend()
178         plt.savefig('WGAN Loss Plot_%d.png' % (epoch))
179         plt.close()
180
181     # This method defines the Wasserstein Loss for the model
182     #
183     # @param real, fake The real and fake labels
184     def wasserstein_loss(self, real, fake): return backend.mean(real * fake)
185
186     # This method loads a dataset of images and prepares
187     # them for training
188     #
189     # @param path The path to the image folder
190     # @return X_train The Training vector
191     def load_data(self, path):
192         X_train = list()
```

```
193         # Loading from directory
194         for im_path in os.listdir(path):
195             img =
                ↪ ImageOps.grayscale(Image.open(path+im_path)).resize((self.img_shape[0],
                ↪ self.img_shape[1]), Image.BILINEAR)
196             img = Image.fromarray(np.uint8(np.where(np.array(img) > 0, 1,
                ↪ 0)*255))
197             X_train.append(np.array(img))
198         # Normalizing the data
199         X_train = (np.array(X_train).astype(np.float32) - 127.5) / 127.5
200         X_train = np.expand_dims(X_train, axis=-1)
201         print("Data Shape: "+str(X_train.shape))
202         return X_train
203
204 # Train WGAN
205 model = WGAN()
206 model.train("train/")  # Change the parameter depending on the path
```

C.7 RoadFilter_WGAN.py

```
1  import keras
2  import numpy as np
3  from PIL import Image
4  import cv2.cv2 as cv2
5  import A_star as star
6  import utilityFunctions as uf
7  from itertools import combinations
8
9  """
10     This class creates and processes the Road Network Image.
11     A WGAN Generates a road image and then it is adapted
12     to the selected terrain using Computer Vision techniques.
13     The road is then placed in Minecraft and further processed
```

```
    to resemble a real-like road system.

    @author: Preslav Kisyov
"""
class Roads:

    # A static variable that defines the path for the model
    path = "stock-filters/Roads/RoadWGAN.h5"

    # This is the main method of the class
    # It initializes vital variables
    # @param n_maps The number of roads to combine
    # @param n_imgs The number of images generated
    def __init__(self, n_maps=2, n_imgs=25):
        self.latent_dim = 100
        self.n_imgs = n_imgs
        self.n_maps = n_maps
        self.usable_floor = None
        self.floor = None
        self.blocks = None

        # Execute functions
        imgs = self.load_model()
        self.road_network = self.get_road_network(imgs)

    # This method loads a keras model from path
    # and produces noise to be passed through the model
    # @return The predicted WGAN images
    def load_model(self):
        model = keras.models.load_model(self.path)
        noise = np.random.random(self.n_imgs *
        ↪  self.latent_dim).reshape(self.n_imgs, self.latent_dim)
        # Generate n_imgs number of images
```

```
46          imgs = model.predict(noise)
47          imgs = (imgs + 1) / 2.0  # Normalize to between 0-1
48
49          return imgs
50
51      # From n_imgs images, perform Preprocessing
52      # and extract one road network image
53      # @param imgs The list of images
54      # @return road_network One generate road image
55      def get_road_network(self, imgs):
56          road_maps = list()
57          image_index = 0
58
59          while len(road_maps) != self.n_maps:
60              # If there are no more valid road maps (above the pct)
61              if image_index > self.n_imgs:
62                  index_list = np.random.random_integers(self.n_imgs,
                    ↪  size=self.n_maps - len(road_maps))
63                  for i in index_list: road_maps.append(imgs[i, :, :, 0])
64                  break
65
66              # Get the percentage of roads present
67              img = imgs[image_index, :, :, 0]
68              count = np.count_nonzero(img)
69              pct_count = (count/float(img.shape[0]*img.shape[1])) * 100.0
70              image_index += 1
71              # Add a road if it contains at least 20% of road infrastructure
72              if pct_count > 20.0: road_maps.append(img)
73
74          # Combine all road maps into one
75          road_network = road_maps[0]
76          for road_map_index in range(1, self.n_maps):
77              road_network += road_maps[road_map_index]
```

```
78
79          return road_network
80
81      # This method processes the road image
82      # by applying Computer Vision methods to clean it
83      # @param region_size The size of the selected Minecraft region
84      # @return A processed road image
85      def process_road_network(self, region_size):
86          # Get the required size and convert to grayscale image
87          size = np.mgrid[region_size.minx:region_size.maxx,
            ↪  region_size.minz:region_size.maxz][0].shape
88          min_size, max_size = min(size), max(size)
89          road_network_gray = np.array(self.road_network * 255, dtype=np.uint8)
90
91          # Perform Closing (Dilation -> Erosion)
92          kernel = np.ones((2, 2), np.uint8)
93          road_network_dilated = cv2.dilate(road_network_gray, kernel)
94          road_network_closed = cv2.erode(road_network_dilated, kernel)
95
96          # Remove Disconnected Components
97          components, objects, stats, _ =
            ↪  cv2.connectedComponentsWithStats(road_network_closed,
            ↪  connectivity=4)
98          sizes = stats[1:, -1]  # Remove the background as a component
99          max_component = np.where(sizes == max(sizes))[0] + 1
100         road_network_closed[objects != max_component] = 0
101
102         # Resize to the required size using Interpolation and convert to
            ↪  Binary
103         processed_road_network =
            ↪  np.array(Image.fromarray(road_network_closed).resize((min_size,
            ↪  max_size), Image.LANCZOS))
```

```
104         return cv2.threshold(processed_road_network,
            ↪ int(np.mean(processed_road_network)), 255, cv2.THRESH_BINARY)[1]
105
106     # This method build the road if not in water
107     # @param level The level provided by MCEdit
108     # @param floor_points, usable_floor The whole floor map and the array
        ↪ containing the roads
109     # @param road The road generated image as an array
110     # @param box The bounding box provided by MCEdit
111     # @param blocks A list of block types
112     def build_road(self, level, floor_points, road, box, usable_floor,
        ↪ blocks):
113         for pos in sorted(floor_points):
114             x, y, z = floor_points[pos]
115
116             # Remove Lava
117             self.remove_lava(level, x, y, z, box)
118
119             if pos in tuple(zip(*np.where(road == 255))) and level.blockAt(x,
                ↪ y, z) not in [0, 8, 9]:
120                 # Remove trees if on road
121                 self.remove_tree(level, x, y, z)
122                 # Build the road
123                 uf.setBlock(level, blocks[0], x, y, z)
124                 # Build tunnel
125                 self.build_tunnel(level, x, y, z, blocks[0])
126                 try:
127                     usable_floor[pos[0]][pos[1]] = 255
128                 except: print("Error: Index is out of range for
                    ↪ usable_floor!\nNo Roads will be connected!\nProbably the
                    ↪ floor is invalid!")
129
130         # Assign floor array and map
```

```
131         self.usable_floor = usable_floor
132         self.floor = floor_points
133         self.blocks = blocks
134 
135         # Get Components
136         components, usable_floor = self.get_components(level, usable_floor,
            ↪  floor_points)
137         floor = np.zeros(usable_floor.shape) if usable_floor is not None else
            ↪  None
138         # Check if there are separated components
139         if components is not None:
140             self.connect_components(level, components, floor, floor_points,
                ↪  np.array(usable_floor), blocks)
141 
142     # This method builds a tunnel if there are block above the path
143     # @param level The level provided by MCEdit
144     # @param x, y, z The coordinates of the path
145     # @param block The type of block
146     def build_tunnel(self, level, x, y, z, block):
147         for i in range(1, 3): # Make the tunnel 2 blocks high
148             if level.blockAt(x, y+i, z) in [1, 2, 3, 12, 13, 78, 80]:
149                 uf.setBlock(level, (0, 0), x, y+i, z)
150                 block_points = [(x+1, z), (x+1, z+1), (x+1, z-1), (x-1, z),
151                                 (x-1, z+1), (x-1, z-1), (x, z+1), (x, z-1)]
152                 is_tunnel_roof = False
153                 for p in block_points:
154                     if level.blockAt(p[0], y+3, p[1]) != 0: is_tunnel_roof =
                        ↪  True; break
155                 # Build tunnel ceiling if necessary
156                 if is_tunnel_roof and level.blockAt(x, y+3, z) in [0, 17, 81,
                    ↪  162, 18, 161]: uf.setBlock(level, block, x, y+3, z)
157 
158     # This method tries to find a path between every disconnected road
```

```
159     # @param level The level provided by MCEdit
160     # @param components The map of disconnected roads
161     # @param free_floor The floor without any obstructions
162     # @param floor_points, usable_floor The whole floor map and the array
        ↪ containing the roads
163     # @param blocks A list of block types
164     def connect_components(self, level, components, free_floor, floor_points,
        ↪ usable_floor, blocks):
165         start_end_points = list(combinations(components.keys(), 2))
166         print("There are " + str(len(components)) + " disconnected roads!")
167         visited = set()
168         # For every disconnected road point
169         for points in start_end_points:
170             if points[0] in visited: continue
171             visited.add(points[0])
172             path = star.search(free_floor, 1, components[points[0]],
                ↪ components[points[1]])
173             ar_path = np.array(path)
174 
175             if path is not None:  # Build connecting road
176                 rows, cols = np.where(ar_path != -1)
177                 for i in range(len(rows)-1): usable_floor[rows[i]][cols[i]] ==
                    ↪ 255  # Add the new path
178                 self.connect_roads(rows, cols, level, floor_points,
                    ↪ usable_floor, blocks)
179 
180         # Update floor array
181         self.usable_floor = usable_floor
182 
183     # This method uses the A* path to connect the disconnected roads
184     # @param rows, cols The rows and columns of the path array
185     # @param level The level provided by MCEdit
```

```
186     # @param floor_points, usable_floor The whole floor map and the array
        ↪ containing the roads
187     # @param blocks A list of block types
188     # @param package=None The package of lvl, start point and blocks for when
        ↪ connecting houses
189     def connect_roads(self, rows, cols, level, floor_points, usable_floor,
        ↪ blocks, package=None):
190         is_start, is_end = False, False
191         block, slab, _ = blocks
192         # Go through every path point
193         for i in range(len(rows)):
194             r, c = rows[i], cols[i]
195             x, y, z = floor_points[(r, c)]
196             # Check if connecting houses to roads
197             if package is not None:
198                 lvl, start, floor_blocks = package
199                 if [r, c] == start or i in [1, 2, 3]:  # If the first several
                    ↪ blocks
200                     if lvl == 0: y = y - 2
201                     elif lvl == 1: y = y - 1
202                     else: y = y
203                 if level.blockAt(x, y, z) in floor_blocks or level.blockAt(x,
                    ↪ y-1, z) in floor_blocks: continue
204 
205             if level.blockAt(x, y, z) in [0, 8, 9]:  # Build Bridge
206                 uf.setBlock(level, block, x, y+1, z)
207                 if not is_start: self.build_bridge_walls(level, block, x, y,
                    ↪ z)
208                 else: is_start = False
209                 is_end = True
210             else:  # Build Road
211                 if is_end:
212                     uf.setBlock(level, slab, x, y+1, z)
```

```
213                     is_end = False
214                 uf.setBlock(level, block, x, y, z)
215                 try:  # Put a step in front of the bridge
216                     if usable_floor[r+1][c+1] != len(usable_floor)-1 and
                        ↪ usable_floor[r+1][c+1] == 1:
217                         uf.setBlock(level, slab, x, y+1, z)
218                         is_start = True
219                 except:
220                     print("Out of bounds for slab - maybe the floor is
                        ↪ invalid!")
221
222     # This method builds the walls of any bridge
223     # @param level The level provided by MCEdit
224     # @param block The block type for the walls
225     # @param x, y, z The coordinates
226     def build_bridge_walls(self, level, block, x, y, z):
227         wall_points = [(x+1, y+1, z+1), (x-1, y+1, z-1), (x+1, y+1, z-1),
            ↪ (x-1, y+1, z+1), (x, y+1, z+1), (x, y+1, z-1), (x+1, y+1, z),
            ↪ (x-1, y+1, z)]
228         for p in wall_points:  # Build the block for each wall point
229             if level.blockAt(p[0], p[1], p[2]) not in [block[0], 8, 9] and
                ↪ level.blockAt(p[0], p[1]-1, p[2]) in [0, 8, 9]:
230                 uf.setBlock(level, block, p[0], y+2, p[2])
231         if level.blockAt(x, y+2, z) in [block[0]]: uf.setBlock(level, (0, 0),
            ↪ x, y+2, z)  # Remove wall if on path
232
233     # This method gets if there are any disconnected roads
234     # @param level The level provided by MCEdit
235     # @param floor_points, usable_floor The whole floor map and the array
        ↪ containing the roads
236     def get_components(self, level, usable_floor, floor_points):
237         try:  # Check if the selected bottom has no air gaps
238             ar_floor = np.array(usable_floor, dtype=np.uint8)
```

```
239             except:
240                 print("Invalid floor has been selected!\nNo components will be
                    ↪ found!")
241                 return None, None
242             # Get any disconnected components from the road map array/image
243             components, objects, _, _ = cv2.connectedComponentsWithStats(ar_floor,
                ↪ connectivity=8)
244
245             if components < 3: return None, None  # The background + 2 separated
                ↪ roads = 3 components
246             comp = np.arange(1, components)  # All components except the
                ↪ background
247             components_map = {}
248             for x in range(0, ar_floor.shape[0]):
249                 for y in range(0, ar_floor.shape[1]):
250                     posx, posy, posz = floor_points[(x, y)]
251                     if level.blockAt(posx, posy, posz) in [8, 9]: ar_floor[x][y] =
                        ↪ 1  # put water in path array
252
253                     if objects[x][y] in comp:  # Record the positions of each
                        ↪ component
254                         components_map[objects[x][y]] = [x, y]
255                         comp = np.delete(comp, np.argwhere(comp == objects[x][y]))
256             return components_map, ar_floor
257
258         # This method removes any detected lava
259         # @param level The level provided by MCEdit
260         # @param box The selected Minecraft region
261         # @param x, y, z The coordinates of the block
262         def remove_lava(self, level, x, y, z, box):
263             block = level.blockAt(x, y, z)
264             c_x, c_y = 0, 0
265             # Get the closest non-obstacle block
```

```
266             while block in [8, 9, 10, 11]:
267                 if x+c_x != box.maxx:
268                     block = level.blockAt(x+c_x, y, z)
269                     c_x += 1
270                 else:
271                     block = level.blockAt(x, y+c_y, z)
272                     c_y += 1
273 
274                 if y+c_y == box.maxy: block = 0
275 
276             # If there is lava
277             if level.blockAt(x, y, z) in [10, 11]: uf.setBlock(level, (block, 0),
                ↪  x, y, z)
278             elif level.blockAt(x, y+1, z) in [10, 11]: uf.setBlock(level, (block,
                ↪  0), x, y+1, z)
279 
280         # This method sets the bounding box of each tree on path
281         # @param level The level provided by MCEdit
282         # @param x, y, z The coordinates of the road
283         def remove_tree(self, level, x, y, z):
284             points_x, points_y, points_z = [], [y], []
285             is_start = True
286             posy = y+1
287             y += 1
288             while True:  # For each y (height) level, get the bounding box if a
                ↪  tree
289                 if level.blockAt(x, y, z) in [17, 81, 162] and is_start:  # Remove
                    ↪  tree iff the tree base is on the road
290                     points_y.append(y)
291                     points_x = self.get_x_bound(level, x, y, z, points_x)
292                     points_z = self.get_z_bound(level, x, y, z, points_z)
293                     if level.blockAt(x, y+1, z) in [18, 161]: is_start = False
```

```
294             elif level.blockAt(x, y, z) in [18, 161] and not is_start:  #
                ↪ Remove the leaves of the tree
295                 points_y.append(y)
296                 points_x = self.get_x_bound(level, x, y, z, points_x)
297                 points_z = self.get_z_bound(level, x, y, z, points_z)
298             else: break
299             y += 1
300         self.remove_tree_from_bound(level, points_x, points_y, points_z, [x,
            ↪ posy, z])
301
302     # This method goes through every point in the found
303     # x, z, y boundary box and removes any trees found
304     # @param level The level given from MCEdit
305     # @param points_x, points_y, points_z The list of x, y, z points
306     # @param road The coordinates of the current road
307     def remove_tree_from_bound(self, level, points_x, points_y, points_z,
        ↪ road):
308         if points_x and points_y and points_z:
309             # Get the min and max of each direction of the box
310             minx, maxx = min(points_x), max(points_x)
311             miny, maxy = min(points_y), max(points_y)
312             minz, maxz = min(points_z), max(points_z)
313
314             first_tree, leaf = False, None
315             last_tree, trees = [], {}
316             # Remove Trees on Path using the found x, z, y boundary box
317             for x in range(minx-2, maxx+2):
318                 for z in range(minz-2, maxz+2):
319                     for y in range(miny-1, 350):
320                         if level.blockAt(x, y, z) in [17, 18, 81, 161, 162]:
321                             if level.blockAt(x, y, z) in [17, 81, 162] and not
                                ↪ first_tree and [x, y, z] != road: trees[(x,
                                ↪ z)] = y
```

```
322                             if level.blockAt(x, y, z) in [17, 81, 162] and not
                                ↪ first_tree and [x, y, z] == road:
323                                 first_tree = True
324                                 uf.setBlock(level, (0, 0), x, y, z)
325                                 last_tree.append((x, z))
326                             elif level.blockAt(x, y, z) in [17, 81, 162] and
                                ↪ first_tree:
327                                 if (x, z) in last_tree: uf.setBlock(level, (0,
                                    ↪ 0), x, y, z)
328                                 else: break
329                             elif level.blockAt(x, y, z) in [18, 161]:
330                                 leaf = level.blockAt(x, y, z)
331                                 uf.setBlock(level, (0, 0), x, y, z)
332                                 first_tree = False
333
334             # Grow back leaves on cut trees that are not on road
335             for tree in trees.iterkeys():
336                 y = trees[tree]
337                 points = [(tree[0]+1, tree[1]+1), (tree[0]-1, tree[1]-1),
                            ↪ (tree[0]+1, tree[1]-1),
338                           (tree[0]-1, tree[1]+1), (tree[0], tree[1]+1),
                            ↪ (tree[0]+1, tree[1]),
339                           (tree[0]-1, tree[1]), (tree[0], tree[1]-1)]
340                 leaf = leaf if leaf is not None else 18  # Set default leaf
341                 for p in points: uf.setBlock(level, (leaf, 0), p[0], y, p[1])
342                 uf.setBlock(level, (leaf, 0), tree[0], y+1, tree[1])
343
344     # Get the leaves of the tree for the X axis
345     # @param level The level provided by MCEdit
346     # @param x, y, z The coordinates to search from
347     # @param points_x The list of found points on the X axis
348     def get_x_bound(self, level, x, y, z, points_x):
349         posx = x+1
```

```
350         reverse = False
351         while True:
352             if reverse is False:  # Search one direction
353                 if level.blockAt(posx, y, z) in [18, 161]:
354                     points_x.append(posx)
355                     posx += 1
356                 else:
357                     posx = x-1
358                     reverse = True
359             else:  # Search the other direction
360                 if level.blockAt(posx, y, z) in [18, 161]:
361                     points_x.append(posx)
362                     posx -= 1
363                 else: break
364         return points_x
365 
366     # Get the leaves of the tree for the Z axis
367     # @param level The level provided by MCEdit
368     # @param x, y, z The coordinates to search from
369     # @param points_z The list of found points on the Z axis
370     def get_z_bound(self, level, x, y, z, points_z):
371         posz = z+1
372         reverse = False
373         while True:
374             if reverse is False:  # Search one direction
375                 if level.blockAt(x, y, posz) in [18, 161]:  # Removes all
                    ↪ trees TODO: Move it so you remove only if on path way
376                     points_z.append(posz)
377                     posz += 1
378                 else:
379                     posz = z-1
380                     reverse = True
381             else:  # Search the other direction
```

```
382                 if level.blockAt(x, y, posz) in [18, 161]:  # Removes all
                    ↪ trees TODO: Move it so you remove only if on path way
383                     points_z.append(posz)
384                     posz -= 1
385                 else: break
386         return points_z
387
388 """
389 This is the main function that
390 gets called after executing the filter in MCEdit
391 """
392 def perform(level, box, options, block=None, need_return=False):
393     road = Roads()
394     road_network = road.process_road_network(box)  # Get the road as a Numpy
        ↪ array Image
395     biom, road_blocks = get_biom(level, box)  # Get the biom and types of
        ↪ blocks
396     floor, usable_floor = get_floor(level, box)  # Get the usable floor box
        ↪ region
397     if block is not None: road_blocks[0] = block # Select Block types
398     road.build_road(level, floor, road_network, box, usable_floor,
        ↪ road_blocks)  # Build and connect roads
399     if need_return: return road
400
401 # This method connects a House (from the door position)
402 # To the closest road, by extending the road system.
403 # It should be called from another filter, and roads should
404 # already be present.
405 # @param level The level provided by MCEdit
406 # @param door_loc The location point of the door
407 # @param road The road class
408 # @param blocks A list of block types
409 def connect_houses_to_roads(level, door_loc, road, blocks):
```

```
410     if road.floor is None or road.usable_floor is None or road.blocks is None:
411         print("Road Floor is None!")
412         return
413     floor = dict((value, key) for key, value in road.floor.iteritems())  #
        ↪ Revert the point map
414     usable_floor = np.array(road.usable_floor)
415     if len(usable_floor.shape) != 2: return
416     x, z, y = door_loc
417 
418     # Get the point in space of the door (Depending on the door position)
419     if floor.get((x, y-2, z)) is not None:  # Beginning of door
420         start = floor.get((x, y-2, z))
421         lvl = abs((x, y, z)[1]) - abs((x, y-2, z)[1])  # 2
422     elif floor.get((x, y-1, z)) is not None:  # Middle of door
423         start = floor.get((x, y-1, z))
424         lvl = abs((x, y, z)[1]) - abs((x, y-1, z)[1])  # 1
425     elif floor.get((x, y, z)) is not None:  # Top of door
426         start = floor.get((x, y, z))
427         lvl = 0
428     else: print("Path cannot be found!"); return
429     start = [start[0], start[1]]
430 
431     # Get end point (First run => get closest)
432     end, run = None, 1
433     ending = [usable_floor.shape[0], usable_floor.shape[1]]
434     beginning = [start[0], start[1]] if start[1] < ending[1] else [0, 0]
435     while run < 3:
436         for r in range(beginning[0], ending[0]):
437             for c in range(beginning[1], ending[1]):
438                 if usable_floor[r][c] == 255: end = [r, c]
439         if end is None: beginning = [0, 0]; run += 1
440         else: break
441     if end is None: return
```

```
442
443         # Get the A* path
444         path = star.search(np.zeros(usable_floor.shape), 1, start, end)
445         if path is None: return
446
447         rows, cols = np.where(np.array(path) != -1)  # Get the actual path
            ↪  positions
448         package = [lvl, start, [block[0] for block in blocks]]
449         road.connect_roads(rows, cols, level, road.floor, usable_floor,
            ↪  road.blocks, package)
450
451     # Get the floor points array and the floor map of coordinates
452     # @param level The level provided by MCEdit
453     # @param box The region selected in Minecraft
454     def get_floor(level, box):
455         mapped_points = {}
456         usable_floor = []
457         pos_x, pos_y = 0, 0
458         for x in range(box.minx, box.maxx):  # depth
459             col = []
460             for z in range(box.minz, box.maxz):  # width
461                 for y in range(box.maxy, box.miny-1, -1):  # height (col) but
                    ↪  count the selected level
462                     if level.blockAt(x, y, z) in [1, 2, 3, 8, 9, 10, 11, 12, 13,
                        ↪  78, 80] and (pos_x, pos_y) not in mapped_points.keys():
463                         mapped_points[(pos_x, pos_y)] = (x, y, z)
464                         col.append(0)
465                         break
466                     elif level.blockAt(x, y-1, z) in [1, 2, 3, 8, 9, 10, 11, 12,
                        ↪  13, 78, 80] and (pos_x, pos_y) not in
                        ↪  mapped_points.keys():
467                         mapped_points[(pos_x, pos_y)] = (x, y-1, z)
468                         col.append(0)
```

```
469                     break
470             pos_y += 1
471         usable_floor.append(col)
472         pos_y = 0
473         pos_x += 1
474
475     return mapped_points, usable_floor
476
477 # This method  gets the biom and
478 # selects road block types given that information.
479 # It is based on probabilities
480 # @param level The level provided by MCEdit
481 # @param box The region selected in Minecraft
482 # @return road_biom, road_blocks The biom and road block types selected on
    ↪ majority voting
483 def get_biom(level, box):
484     road_choices = []
485     # Choice per slice
486     for (chunk, slices, point) in level.getChunkSlices(box):
487         bins = np.bincount(chunk.root_tag["Level"]["Biomes"].value)
488         count = bins/float(len(chunk.root_tag["Level"]["Biomes"].value))
489         bioms = np.flatnonzero(count)
490         probs = {i:count[i] for i in bioms}
491         road_choices.append(np.random.choice(probs.keys(), p=probs.values()))
492
493     road_biom = max(road_choices, key=road_choices.count)
494     road_blocks = get_road_blocks(road_biom)
495     return road_biom, road_blocks
496
497 # All Minecraft blocks can be found here -
    ↪ https://minecraft-ids.grahamedgecombe.com/
498 # All Minecraft biomes can be found here -
    ↪ https://minecraft.fandom.com/wiki/Biome/ID
```

```
499 # From official Minecraft Wiki (Biomes and Blocks ID)
500 # This method is to be used if the filter is used alone
501 def get_road_blocks(road_biom):
502     # Block, Slab, Stairs
503     if road_biom in [2, 7, 16, 17, 27, 28, 36, 37, 38, 39, 130, 165, 166,
        ↪  167]: return [(43, 0), (44, 0), (109, 0)]
504     else: return [(1, 0), (44, 5), (67, 0)]
```