AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

Wydział Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej

Projekt dyplomowy

Procedural Terrain Generation using Artificial Neural Networks

Wykorzystanie Sztucznych Sieci Neuronowych do Proceduralnej Generacji Terenu

Autor:	*Jakub Kluk*
Kierunek studiów:	Automatyka i Robotyka
Opiekun pracy:	*dr hab. Adrian Horzyk, Prof. AGH*

Kraków, 2020

Abstract

The work deals with the subject of using neural networks for the task of Procedural Generation. Specifically, the generation of the terrain maps is concerned. This work introduces some of the important topics in the subject and debate over the network architecture capable of imitating the Earth topography captured by the satellite in the form of the heightmaps. The architecture designed for the task is a GAN neural network learning the land heights conditional distribution on the planet's surface. The network preceded by the elements responsible for the data preparation is able to create grayscale maps containing landmasses resembling those naturally occurring on the Earth. These generated maps are then processed and visualized for a comparison with the classical PTG algorithm – Perlin's Noise, in which the GAN reveals its advantages in the quality of the data generated and the computational performance, but also its drawbacks in controlling the process.

Table of Contents

1. Introduction

Due to improvements in storage capacity, data transmission bandwidth, and computational power in general, Neural Networks and Artificial Intelligence achieved a major rise in popularity recently, which caused new algorithms and architectures to be created. Rapidly growing capabilities resulted in introducing artificial intelligence systems to more and more business domains and everyday life. These days not only marketing and data analysis exploits these advanced algorithms, but also the automotive industry (self-driving cars, driver assistance systems), social media (content suggestions, image recognition), gamedev (controlling enemies, pathfinding), and much more.

One of the most interesting techniques in the field of machine learning is data generation, which was used mostly for the task of natural language creation. However, with recently introduced neural network architectures, models producing sound or images became more common. For example, important papers in this field were Nvidia GAN generating fake faces of celebrities that never existed [15], or a program imitating Obama's voice and lips move [14]. These works proved image/voice generating models to be amazingly efficient.

Surprisingly, despite the unusual quality of produced data, generative models have not been used in the task of procedural generation yet. Only a few papers mention using artificial intelligence for terrain generation, two of which are a CGAN based tool for enhancing sketches with mountain-like textures [4] or a random map generator [2]. Although the latter has much in common with this work, it only covers the topic very slightly.

The lack of use cases in this field is meant to be addressed in this work.

1.1. Goals and objectives

This paper aims not only to study the possibilities of using machine learning in the field of procedural generation but also to compare generation effects with the traditional algorithms used in the field and provide the reader with graphically-enhanced visualizations of created landscapes.

This work will introduce some of the most common random generation algorithms and present their advantages and drawbacks in comparison with results achieved with an appropriately trained neural network.

Into consideration will be taken factors such as the naturalness of generated landmasses, computing complexity of the algorithm used, additional features like rivers and mountains, and the possibility to control the parameters of the output image.

1.2. Project scope

To meet all the above goals, there should be implemented an application that allows to:

- Load and process Earth heightmaps
- Train a neural network on the delivered data
- Generate maps resembling original landscapes using trained network
- Generate maps using any other procedural generation algorithm
- Visualize and compare the outputs

1.3. Technology used

All the parts of the project were created using Python programming language. It was a natural choice as Python is one of the most popular programming languages globally and probably the most popular one in terms of data science. Numerous libraries such as NumPy, PIL, or matplotlib allow for the fast and effective creation of data processing pipelines. It also offers a wide variety of neural network designing frameworks, out of which PyTorch has been chosen as the most “pythonic” one.

PyTorch is a machine learning framework designed specifically for Python, thus it guarantees ease of use in combination with other technologies – for example, NumPy arrays can be used interchangeably with PyTorch tensors. It also delivers rich documentation and a living community, which improves the development process. Last but not least, the framework supports CUDA accelerated GPU’s out-of-the-box, which means that the developer does not have to waste time configuring the environment in order to speed up computations.

2. Review of standard algorithms

This work is meant not only to explore the possibilities of adopting Neural Networks to the task of Procedural Terrain Generation but also to compare the results obtained against those of algorithms commonly used in the field. Therefore, there is a need to introduce some definitions of concepts, goals, and assumptions associated with the topic that will heighten the understanding of the domain of PCG (*Procedural Content Generation*), of which a part is PTG (*Procedural Terrain Generation*). This chapter will provide the reader with an arbitrary definition of both concepts that will suit the needs of this work. It is also meant to introduce a variety of different use cases of PCG and explain the place of PTG in the area. Further, the two most popular algorithms used in today's world will be introduced and in-depth explained. Especially the pros and cons of these two will be highlighted in order to enable both computational efficiency and visual quality comparison of the results.

2.1. Procedural Generation

The term Procedural Content Generation "*refers to the algorithmic generation of game content with limited or no human contribution*" [16]. The term "content" is understood as "*levels, maps, game rules, textures, stories, items, quests, music, weapons, vehicles, characters, etc. The game engine itself is not considered to be content in our definition. Further, non-player character behaviour—NPC AI—is not considered to be content either.*" [16]. Under that definition, one can say that content generated procedurally refers to every non-technical aspect of the game. Given the set of the rules (game engine), every "static" content (things that can be translated into a binary file – not functions reacting to players behaviour) can be produced as the result of some PCG algorithm.

Although the definition refers to PCG as a game-only solution, it is not completely the case. Procedural Generation has also been widely used in cinematography (for example, TRON movie) or computer graphics. These non-game related fields have been using the same algorithms for a content generation as the gamedev but without some of the constraints. For example, it must be possible in the game to trespass the procedurally generated labyrinth, while in the movie, it probably does not matter. As the difference between these two approaches is inconsequential in this paper, the Procedural Content Generation will refer to every content created by an algorithm without human contribution. Given the fact that the term “game” is tough to define [16], this approach seems to be the correct one.

Although this work focuses on the terrain generation task, one has to be aware of other possible use cases of the procedural generation. Algorithms can be used to automatically create quests, dialogues, or even stories and genealogical lineages of whole tribes (using Natural Language Processing) that can enrich games story and help the player get involved in the “living” world. Procedural generation can also be applied to creating graphics and textures, for example dynamically created scars on the fallen enemies. Those are the main areas where this technique is being used, but in some minor projects, PCG can be applied to the process of creation of music, items (statistics, descriptions, and textures), or NPC’s (Non-Player Characters). There are also artificial intelligence solutions for creating entire games (for example, ANGELINA project). [11]

The terrain generation, which is the subject of this work, is one of the most common procedural generation use cases. Nowadays, hardly any video game that aims to draw customer’s attention with replayability and countless hours of gameplay comes without some kind of automatic level generation. Creating a different map each time the user starts a new game grants him a unique experience and gives meaning to some aspects of the game, like the exploration, which only makes sense when one is not aware of what lies where. Usage of random-generated maps also benefits over hand-crafted ones in the case of production costs because it is very time-consuming to design several maps manually. Gameplay enhancement and lowering production costs are the reasons why so many developers got algorithms to work over a world design . As one can see, in many cases, it proved to be beneficial. Among the most successful projects using procedural generation there are: “Minecraft”, “Civilization” series, or “No Man’s Skye”. In all of those productions, the random world generation is not just a feature. It is the core of the game.

2.2. Perlin’s Noise

Perlin’s Noise is a gradient noise invented by Ken Perlin while working on a TRON movie in 1982. It is a noise function that generates a continuous-like series of numbers from the input coming from a space of arbitrary dimensionality. Generated numbers are pseudo-random because the function, once created with a particular seed (random number generator’s parameter), always produces the same output for a given set of inputs. Series of numbers generated by the Perlin’s Noise function differs from those drawn from standard random noises by their property of arranging into structures resembling naturally occurring objects, like textures of some materials, or, what is the case for this paper, height maps of landmasses. [13]

Ken Perlin’s algorithm can be used to acquire a grayscale image where the brightness of a pixel with coordinates (x, y) can be interpreted as the height of an (x, y) point in a two-dimensional space. The algorithm can be adjusted to work with data of any dimensionality. However, the most common use cases are for two (land surface, coastline) and three dimensions (cave systems, three-dimensional worlds).

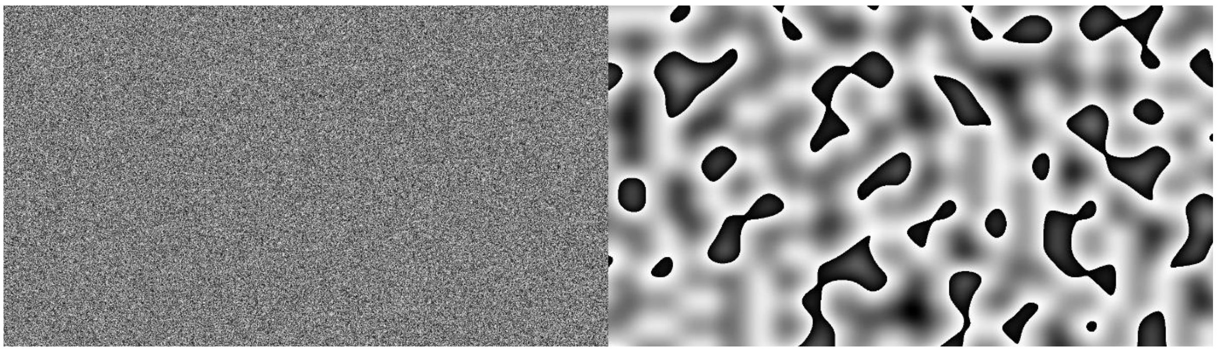

Figure 2.2.1: Uniform noise (left) vs Perlin's noise (right)

As can be seen, the brightness of pixels on the image generated using the procedural generation algorithm changes very slightly with the change of input coordinates (the function behaves as if it were a continuous one). In contrast points on the image generated using standard noise are completely unrelated (Figure 2.2.1). Despite the fine properties of the noise on the right side of (Figure 2.2.1), it still does not resemble natural landscapes. In order to improve the quality of the resulting image, the function generating noise is being called iteratively (single iteration is called as “octave”) with growing frequency (coordinates multiplier – increasing function’s variation) and decreasing amplitude (maximum value of the output). Adding outputs of every iteration together results in the final image. By changing the number of octaves, the rate of frequency growth (lacunarity), and amplitude decrease (persistence), one can control the process of noise creation.

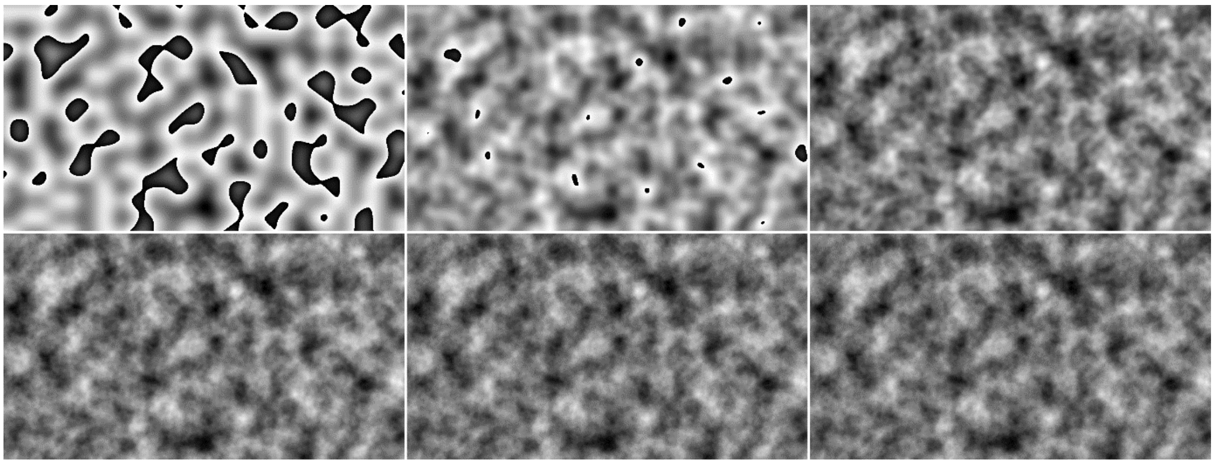

Figure 2.2.2: Noise generation using from 1 to 6 octaves

With this iterative approach, generated maps look much better, but at the cost of greatly increased computational complexity. (Figure 2.2.2).

After the grayscale heightmap has been generated, it can be easily transformed into a terrain-like map by replacing the chosen percentage of the darkest pixels with the black ones (representing the sea) and rescaling the others into the 1-255 range, representing elevation over sea level. This approach is the core of most of Perlin’s Noise based Procedural Terrain Generation algorithms.

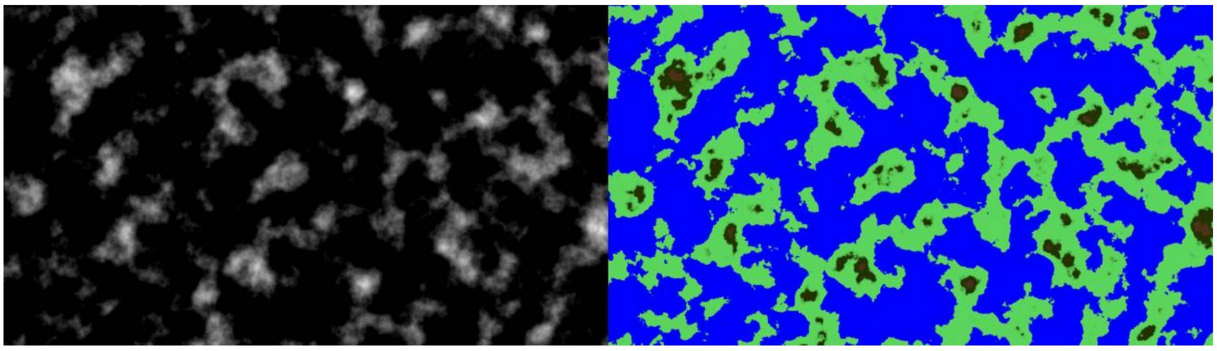

Figure 2.2.3: Archipelagos created with Perlin’s Noise in grayscale (left) and after simple coloring (right)

The map visible in (Figure 2.2.3) is the last image from (Figure 2.2.2) processed using exactly the approach described above. This archipelago-like map can be further processed to suit the user’s needs better. For example, instead of generating a map full of small islands, it is possible to generate a map with a single continent (or a few) by adding to the heightmap a function of distance from chosen points (these points would be centers of continents). There is a number of other methods of changing the map appearance, like replacing the map edges with low-level pixels only or deleting tiny islands. However, they are not the subject of this work. The interested reader can find more information about the topic over there [3].

Perlin's Noise is one of the most famous terrain generation methods and is still widely used today as the baseline for other algorithms. The spectacularity of the maps created with it is admirable, but that does not mean the method is flawless. One of the major drawbacks of the function is its inability to create naturally-looking mountain ranges. Because mountains have been created in the process of tectonic plates collisions, they tend to line up, while the algorithm creates spots of high-valued pixels. In addition to mountains, the noise does not handle creating rivers, so they have to be added in the post-processing stage. Adding all of these features (and many others) in combination with a map creation using multiple iterations (octaves) results in a very high computing complexity of the process. Finding an algorithm that does not require so much post-processing and does not take so much time would be very beneficial for the field of procedural generation.

2.3. Cellular Automata

A Cellular Automaton is a discrete computational system consisting of a grid of elementary computing objects, called cells, interacting with each other. Theoretically, there are no limitations in the shape of a grid and its dimensionality, but the most commonly encountered are those of dimensionality of two or three. The grid consists of cells (mostly square ones, but the choice of shape is arbitral), all of which have a set of possible states and an initial state. There also should be rules for defining a neighborhood of a cell and for the state transitions. The automaton carries computations in iterations, starting at the moment t=0 with every cell in its initial state and then updating each cell's state based on the current states of its neighbors. The number of possible usages of this simple model is surprisingly big. With a proper set of possible states and state-transition rules, one can simulate an enormous amount of real processes. Cellular automata are being widely used in physics, biology, chemistry, or even social sciences.

It comes as no surprise that with so many possibilities, a cellular automaton can also be used in the process of terrain generation. There are premises that this approach may grant a user with event better results than noise-based ones. However, this requires using an automaton with a very complex set of rules simulating soil erosion's natural processes, which needs an enormous number of iterations to be computed. A real-world use case of a cellular automaton deals with this problem by combining the soil-forming simulations with some noise functions [12]. Due to the high complexity, computational and logical alike, cellular automata are being rather rarely used for terrain generation. However, they are used for a procedural generation of some kind of game levels, like caves (Figure 2.3.1), labyrinths, or cities [10].

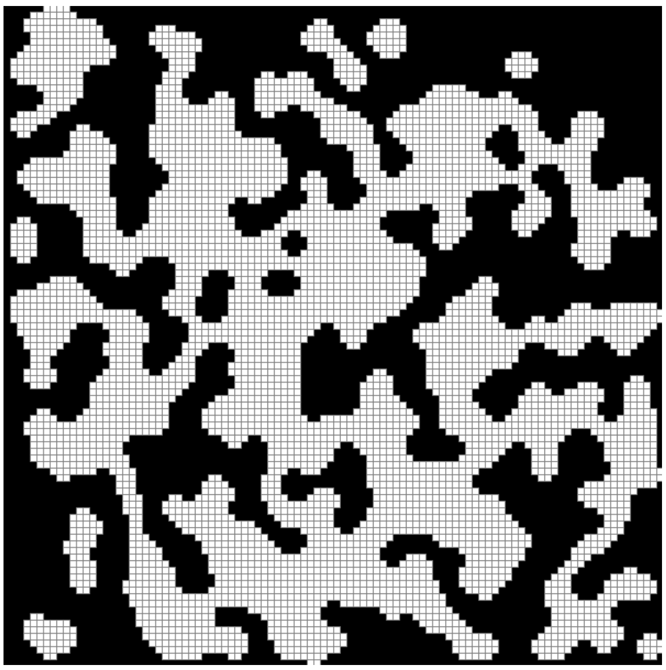

Figure 2.3.1: An example of a cave generated using a cellular automata [1]

Cellular automata are promising computational models applied in numerous fields, but in the case of Procedural Terrain Generation, some methods grant users with better results at a lower computational cost. Despite the fact that automata can be easily adapted to the task of a binary coastline generation, it cannot be applied to creating a heightmap without a set of very complex rules or combination with a noise function, which makes it unprofitable to use. This is the reason why this model is being used mostly for simple tasks or to post-process the map generated using other methods (it may be a good idea to implement a river-creating algorithm based on a cellular automaton).

3. Theoretical analysis

The term "Neural Network" is a broad concept these days. There are plenty of different architectures that suit different needs and problems. A properly designed network can be used as a solution for almost any type of machine learning problems, such as recognizing people's faces, translating sentences, text-to-speech synthesis, or even art generation. The area develops so rapidly it is almost impossible to remember every type of neural network designed or to track the newest developments in the field. Not only is there a great variety of architectures, but also every neural model can be adjusted to suit one's needs by using different training procedures, data preparation techniques, or by fitting a vast diversity of parameters. It is also possible to combine a few nets together, so the possibilities are almost infinite. This chapter will introduce neural networks basics in general and discuss certain architectures that are being used in the work.

3.1. Neural Networks basics

3.1.1. Definition

A neural network is a computing system that was invented as a human brain simulation model. It consists of a set of simple computing units – called neurons, organized into interconnected layers (Figure 3.1.1). Every neuron consists of an arbitrary number of inputs, each assigned a value called weight, and a single output. When a neuron receives a signal, it computes the output as some function - called the activation function, of the sum of weighted inputs. Formally, it can be described as:

$$o = f(\sum_{i=1}^{n} w_i a_i) \qquad (3.1.1)$$

Where:

$$n \in \mathbb{Z} - number\ of\ inputs,$$
$$a_i \in \mathbb{R} - the\ i^{th}\ input,$$
$$w_i \in \mathbb{R} - weight\ of\ the\ i^{th}\ input,$$
$$f: \mathbb{R} \to \mathbb{R} - activation\ function$$

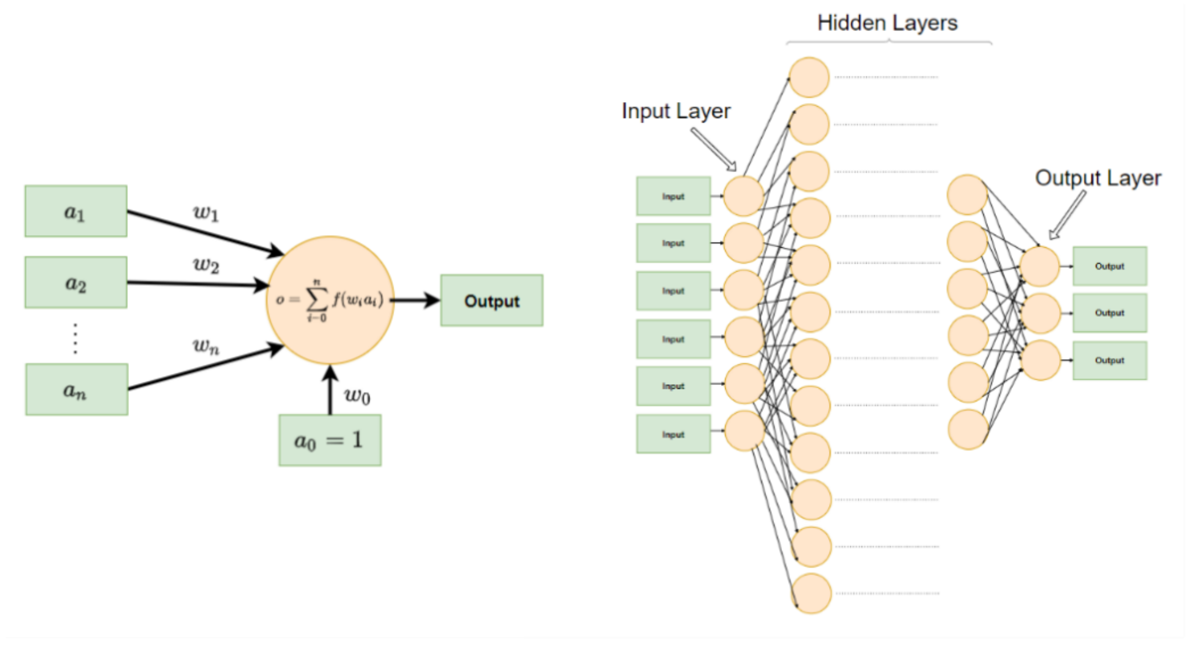

Figure 3.1.1: Visualizations of a single neuron (left) and a whole network (right).

Sometimes, a neuron can have an additional input that, instead of taking the outside signal, is actuated with the constant one. This input act as the intercept of the approximated function and is called the bias. The formula (3.1.1) can be adjusted to take the bias into account by simply starting the summation from i = 0 instead of i = 1 (with a_0 always equal to 1).

Neuron’s properties, and hence the network's properties, depend strongly on the activation function type, which can be chosen almost arbitrarily. However, as the training algorithm requires calculating the derivative, differentiable activation functions are preferred. In the early days of neural nets, it was a simple linear function allowing for training the network quickly while restricting its approximation abilities to the linear functions only. The alternative was a step function converting the output to a binary format with activated/deactivated neuron - hence the name activation function. As these classical functions entail either an information loss or an impediment to the training process, new ones were developed. These modern functions are mostly nonlinear ones like ReLU or a logistic function (Figure 3.1.2).

As a single neuron is not capable of performing complicated computations, there is a need to combine many of them in a network structure that is typically organized into layers. Usually, neurons from the same layer are not connected, and only adjacent layers are connected in a way, that outputs of the previous layer become inputs of the next one. The first layer, referred to as the input layer, receives the data, and the last layer – the output layer, returns the computation results. Layers between them are referred to as the hidden layers (Figure 3.1.1).

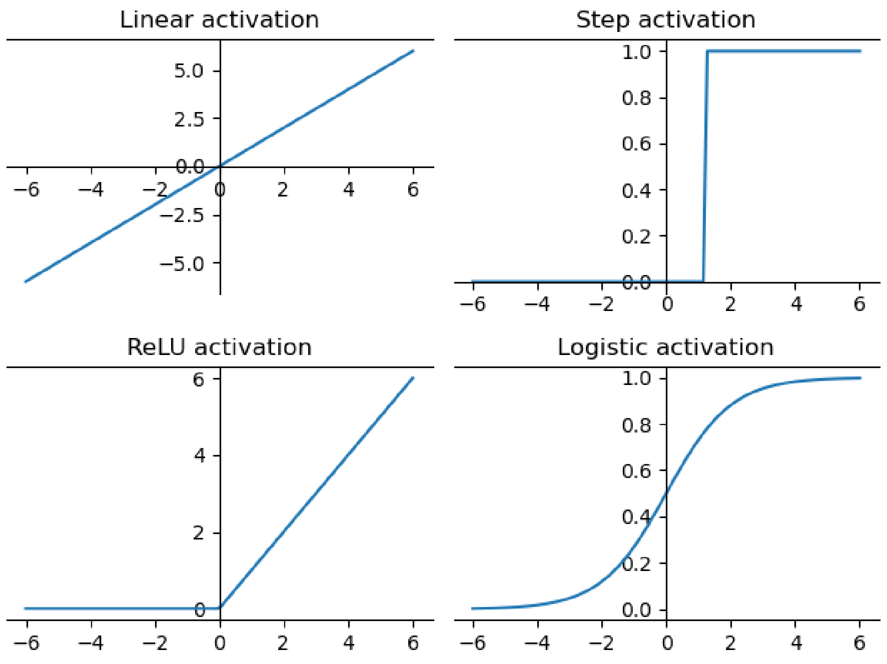

Figure 3.1.2: Example activation functions

Instead of using the formula (3.1.1) explicitly, the single layer's output can be described in a very compact way using a matrix multiplication manner described in (3.1.2).

$$o = f(Wa + b) \tag{3.1.2}$$

Where:

$$a \in \mathbb{R}^n - input\ vector,$$
$$o \in \mathbb{R}^m - output\ vector,$$
$$W \in \mathbb{R}^{m\,x\,n} - weights\ matrix,$$
$$b \in \mathbb{R}^m - bias\ vector,$$
$$f: \mathbb{R}^m \rightarrow \mathbb{R}^m - activation\ function$$

The bias term "b" can be omitted just as in the case of formula (3.1.1) by appending a "1" to the input vector and adding another row to the weight matrix. The formula described above not only improves the readability of the mathematical operations associated with neural networks but also allows to speed-up the computations, as the matrix multiplication algorithm is a very well-known and optimized one. A whole neural network can be described as a series of matrix multiplication operations (or just a single multiplication with a properly prepared matrix) using the (3.1.2), which dramatically improves the calculations.

The information flow in the simplest neural networks is straightforward, from left to right, from the input, through the hidden layers to the output without any feedback connections. However, there are neural network architectures that use the feedback connection - output from the layer connected as the input to the previous layer, or a neuron using its own output as one of its inputs. Those networks are called Recurrent Neural Networks (RNN), and they are instrumental in tasks requiring the keeping of input order (like in time series prediction or speech translation). Though they are also tricky to train, and they need special treatment. [5]

3.1.2. Training

Despite the simplicity of conducted operations, due to the enormous number of connections, the model described above is able to approximate any nonlinear, multidimensional function - provided there are enough layers and connections. It can be done by adjusting the weights of every connection properly. The task of choosing the right weight values is obviously beyond human capabilities (in most cases), but there are algorithms that allow us to set these values automatically. The process of optimizing the chosen weights is called the "training process".

The training is conducted iteratively. In the beginning, the network's weights are initialized randomly. Then, the network is shown some example input data with the corresponding expected output. The input is processed through the network, and the result is compared with the delivered output, using some metric function measuring the difference (loss function). The comparison error is then backpropagated [5] through the network which weights are updated accordingly. The training process for each layer, described with (3.1.2), can be presented formally as an optimization process (3.1.3):

$$\min_{W} L(Y', Y) = \min_{W} L(f(WX), Y) \tag{3.1.3}$$

Where:

$$X \in \mathbb{R}^{n} - input\ vector,$$
$$W \in R^{n\times o} - weight\ matrix,$$
$$f: \mathbb{R}^{o} \to \mathbb{R}^{o} - activation\ function$$
$$L: (\mathbb{R}^{o}, \mathbb{R}^{o}) \to \mathbb{R} - the\ loss\ function$$

Unfortunately, the equation (3.1.3) requires both input and output values to be known, which is the case only for the last layer in the neural network, as we do not know how the output of the hidden layers should look like. In order to fully train the network, one needs to deliver a method of determining the loss of the inner layers, which in most cases is the backpropagation algorithm. The method is based on the accumulation of the neuron's errors with respect to every input's weight and projecting them to the previous layers where they can be used in a slightly modified version of the formula (3.1.3).

The process of upgrading weights to minimize the formula (3.1.3) is carried out with the use of a stochastic gradient algorithm, which allows for very slight changes of the weights in every iteration (every example showed), which grants the network the ability to fit every example as good as possible.

3.1.3. Designing a network

In the training process, the network fits itself to the problem by modifying the weights of the connections, but even the most prolonged training cannot improve the performance in the case the layout has been designed poorly. A famous example is the XOR function, which could not be learned by any single-layer network, no matter the delivered data. This is the reason why the size, the number of layers, and types of connections have to be determined carefully at the design stage for the network's ability to remember and training effectiveness strongly depends on it. Although it is possible to simply connect every neuron in a layer with everything in another layer (a layer like this is called a fully connected layer) and let the training algorithm reduce the weights of unimportant connections to almost zero, it is not always the best solution. Fully connecting all of the layers in the network results in a tremendous number of parameters (number of connections grows exponentially with the number of neurons), optimization of which requires an abundance of data and computing time. In massive networks, it is advised to connect a neuron's output with only a few closest inputs or choose a completely different strategy (like in the convolutional networks).

Aside from choosing the connection types, it is also important to determine the best possible number of layers to suit the problem. Although it has been proven that a network with three layers can approximate any function, provided it consists of a sufficient number of neurons, the "sufficient number" is the key because using too many neurons leads to infeasible computations. Shallow network architectures can be simplified by using a set of hand-crafted and pre-computed features instead of the raw data as the net's input. A simple example is using a Fourier Transformation instead of a raw signal, but designing features like this often requires extensive knowledge of the field. Choosing a thinner and deeper neural network architecture can save the effort, as every layer of a network like this creates a higher-level abstractive representation of the data, which is exactly the feature choosing procedure. This approach not only requires a lesser workload at the stage of data preprocessing, but it also grants the user savings in memory and computation time.

The rose in computing capabilities and an enormous increase in the amount of data collected and available to the scientists made it possible to use deeper networks on a larger scale. No, or only a little of domain knowledge requirements allowed a greater community to experiment with this kind of networks, which caused a swift development of the field, which continues today.

3.2. Convolutional Neural Networks

A Convolutional Neural Network is a special kind of a deep neural network adapted mainly to processing large, two-dimensional data inputs, like, for instance – images. However, the architecture's dimensionality can be arbitrary. For example, single-dimensional convolutional networks are being used in a time series processing. The advantage of a convolutional neural network is the reduction in the number of connections, which strongly lowers the number of parameters to train and, what follows, decreases the computational complexity of the training process.

It is achieved with the usage of so-called convolutional layers, that instead of processing the data through the weighted connections in the matrix multiplication manner described in (3.1.2) use the convolution operation - a linear operation over two (in case of neural networks - discrete) functions, defined as:

$$f * g = \sum_{i=0}^{n} f(i)g(n-i) \qquad (3.2.1)$$

Where functions "f" and "g" can be interpreted as the layer's input and the convolution kernel. The operation (3.2.1) can be defined as a multiplication of some kind of sparse matrices, where most of the elements are equal to zero, due to the fact that the typical convolution kernel is much smaller than the input data. However, the (3.2.1) form is much faster than computing the product of two matrices, as the operations, including the zero value, are omitted by default.

Using a convolution over some input may be interpreted as shifting the convolution kernel over the data and calculating the dot product of the kernel and the overlapping piece of data (Figure 3.2.1), which results in an array of a usually smaller size than the input data, containing aggregated information about the local data specification. Thus, the convolutional layers work as excellent feature extractors. For example, a convolutional layer with a kernel of size 3x3 might be used to extract from the input image the information about the lines it contains (horizontal, vertical, or in any direction), and the next layer might detect edges made of these lines.

What a convolutional layer is able to detect and what is the size of the output depends on the layer's parameters, the most important of which are: kernel size, stride, and padding. The kernel is usually of the same dimensionality as the input data and of a much smaller size. In the case of images, the most commonly encountered are kernels of size 3x3 or 5x5. The bigger the kernel, the more advanced features it can detect, but also the more computations it needs. It is recommended to keep the kernel's size relatively small in order to extract low-level features and keep the computations simple. The stride defines how many positions the kernel is shifted after each operation – for the stride = 1, the computations are done for adjacent windows. Higher stride values might be used to decrease the size of the output data and prevent the layer from extracting highly dependent features. The output size from the layer can also be controlled by the padding (or zero-padding) parameter. This parameter specifies how to surround the input data with the zero values border (the bigger the padding, the thicker the border). The convolution using all of the described parameters is visualized in (Figure 3.2.1).

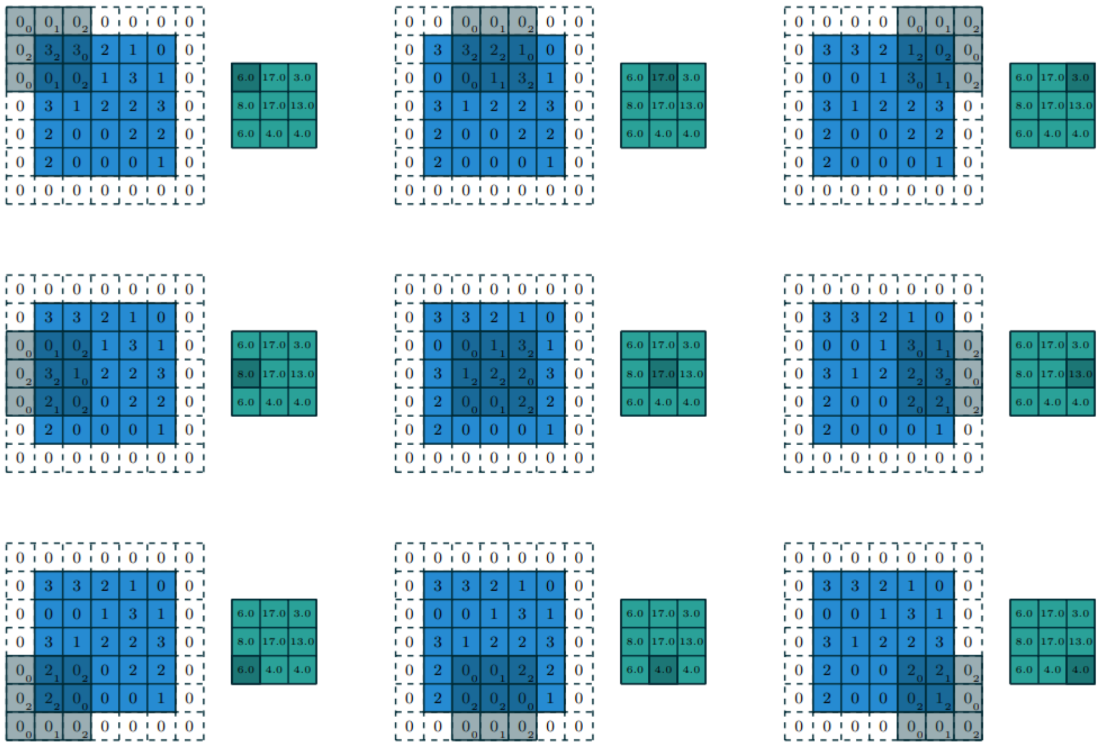

Figure 3.2.1: Example of a convolution with the input data of size (5,5), the kernel of size (3,3), stride = 2 and padding = 1. [17]

As the convolution operation does not interact between axes, the formula describing the output's size can be derived for only a single dimension without the loss of generality. The equation that takes into consideration all of the parameters described above can be found below:

$$o = \left\lfloor \frac{i + 2p - k}{s} \right\rfloor + 1 \tag{3.2.2}$$

Where:

$$o\ -\ size\ of\ the\ output$$
$$i\ -\ size\ of\ the\ input$$
$$k\ -\ size\ of\ the\ kernel$$
$$p\ -\ padding\ value$$
$$s\ -\ stride\ value$$

In addition to standard convolution layers, there are also other types of neural network layers using a convolution operation. One of them that matters under the terms of this work is a transposed convolutional layer. A transposed layer follows exactly the same rules as the standard convolutional layer, but the other way. A transposed layer uses some higher-level abstraction (features) as the input and derives from it lower-level data. It can be used to increase the dimensionality of the data and, for example, to create images from sets of features.

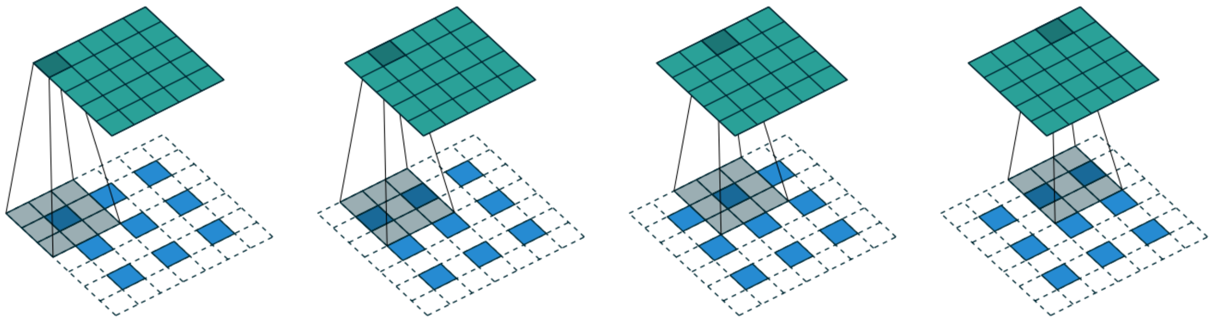
Figure 3.2.2: Example of a transposed convolution with the input data of size (5,5), the kernel of size (3,3), stride = 2 and padding = 1. [17]

As one can see, in the transposed convolution layer, the information from a single feature is being dispersed into the piece of output of kernel size (overlapping windows sums up), where the stride is responsible for the width of free space between output cells.

A typical convolutional neural network (Figure 3.2.3) starts with a few convolution layers intertwined with pooling layers (down-sampling layers responsible for reducing the data dimensionality) and ends with fully connected layers. Such a structure allows the network to extract the features from the input data and then conduct the real computations on these features, which results in a better outcome with even lover computing complexity. A convolutional network resembles two stacked networks, the first of which is responsible for the feature extraction, and the second seeks for the problem's solution using the information delivered by the convolutional part of the network.

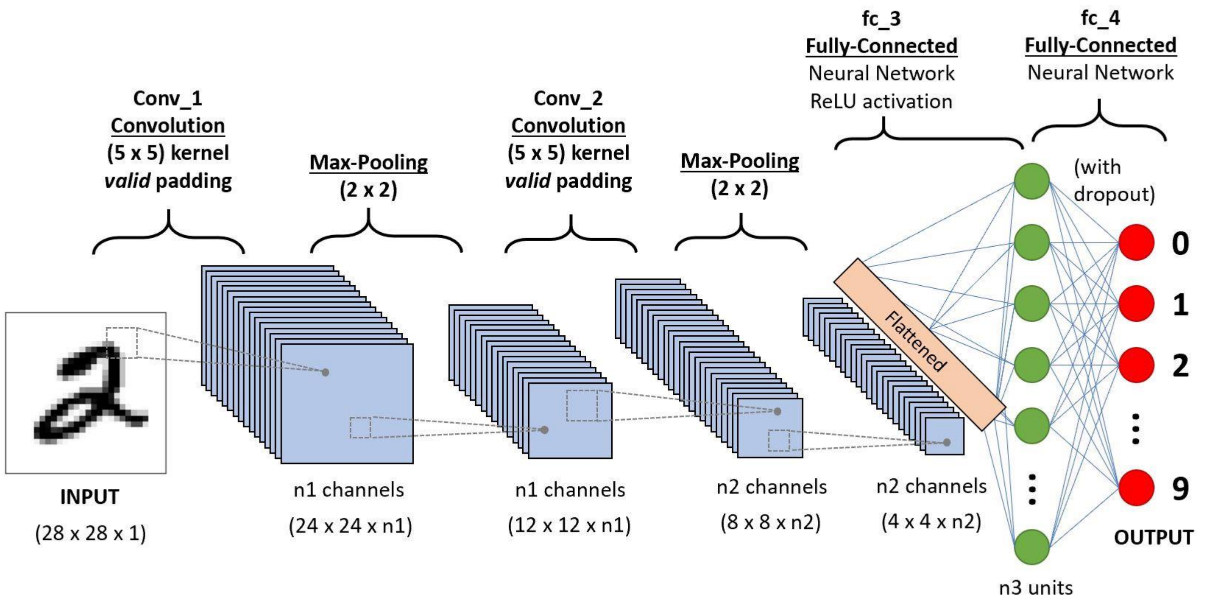

Figure 3.2.3: An example architecture of a convolutional neural network. It starts with the convolutional layers intertwined with pooling layers, and ends with the fully connected network.

A more detailed introduction to convolutional neural networks can be found in [5].

3.3. Generative Adversarial Networks

A GAN (Generative Adversarial Networks) is a specific type of neural network architecture. It follows an unusual approach of setting two different networks (of any type) in a competition where the first network tries to fool the other one. This kind of game allows both models to learn from each other, which results in high-quality outcomes with a significantly lower amount of training data needed. The GAN architecture has been created to solve the task of generating the data coming from the same probability distribution as the training data – for example, creating faces of nonexistent people [15].

The model consists of two neural networks – a generator and a discriminator, both of which can be combined with additional data processing elements like picture generators or objects performing morphological operations. Networks are coupled in a way that the input of the discriminator receives either the real data or the generator's output. The discriminator's task is to recognize whether its input is an authentic or a fake - created by the generator, which means it is a kind of a binary classifier. Simultaneously, the generator is attempting to fool its competitor by producing objects resembling the original data as much as possible.

The discriminator is trained to recognize the data produced by the generator. The output of the network can be either a binary classification – 0 for fake, 1 for real, or a continuous value being a metric of the authenticity of the input – the higher, the more probable it is that the data comes from the training distribution. The discriminator's training is conducted as the training of every other neural network type – by delivering the real and fake data in turns and upgrading the network's weights using the backpropagation algorithm. The objective of the discriminator can be formally described as the maximization of the difference between the results for the real and fake objects, or as the maximization of the output for the real ones:

$$maxf(D) = D(x) - D\big(G(r)\big) \tag{3.3.1}$$

$$max\, f(D) \ = \ \mathbb{E}_x[D(x)] \tag{3.3.2}$$

Where:

$D \ - \ discriminator\ model,$

$G \ - \ generator\ model,$

$x \ - \ real\ data\ sample,$

$r \ - \ random\ noise$

On the other hand, the generator is trying to produce fake data from the input that is mostly a noise (the type of noise used does not affect the quality of produced output), but in some cases, it can be meaningful data, for example, an image that has to be processed in a special way [6]. The network is being trained by generating the data – that at the beginning resembles only random noise, and then processing them through the discriminator and backpropagating the error (of the generator's loss function) as if they were a single neural network (during the process the discriminator's weights aren't updated). This method delivers information about the quality of the generated data despite the fact that it is almost impossible to measure the resemblance of generated structures as it is often subjective. The objective optimized by the generator is to maximize the discriminator's output for the fake object:

$$max\, f(G) = \mathbb{E}_r[D(G(r))] \tag{3.3.3}$$

Training the whole GAN network is a challenging process, for the task of data generation being way harder than simple classification. A discriminative model (an example of which is the discriminator) in order to recognize the input, only has to learn the boundaries between data types. In contrast, the generative model (the generator) has to model the exact data distribution, which requires much more information.

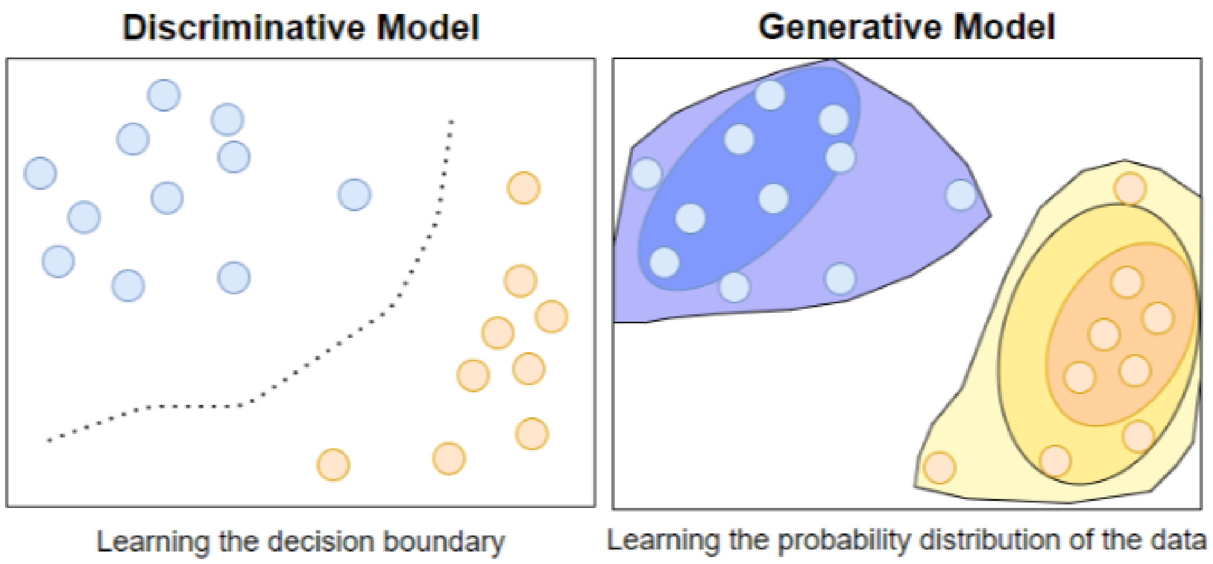
Discriminative Model
Generative Model
Learning the decision boundary
Learning the probability distribution of the data

Figure 3.3.1: A discriminative versus a generative model.

(Figure 3.3.31) presents a visualization of the difference in the amount of the information needed for a discriminative and a generative model. As the discriminator requires less information to complete the task, there is a risk that it will quickly achieve a one hundred percent accuracy, which will stop the generator from learning any further due to the lack of positive feedback. In order to prevent this from happening, a GAN network has to be trained in a particular manner.

Training the GAN network takes place in stages in order for the generative network not to have to fit the moving target. Every stage, built from a few steps, consists of processing a single batch of the data. The first step is the training of the discriminator by processing through it the real data and generator's output in turns (the discriminator's weights are frozen during the process). It is a common practice to "smooth" the data labels (instead of 0, 1 values, one can use a random noise with the mean close to 0, 1) to make the task harder for a discriminator and deliver more information to the generator. The next step is to freeze the discriminator's weights and to train the generator. The training process is considered completed when the discriminator's accuracy drops to near 50% because it means that it no longer can distinguish fake images from the real ones, but it does not guarantee that the results would be acceptable, as the quality of the object generated strongly depends on the quality of the classifier.

The training process described above can be formally represented by the combination of equations (3.3.3) and (3.3.2) / (3.3.1):

$$\min_G \max_D V(D,G) = \mathbb{E}_x[D(x)] + \mathbb{E}_r[1 - D(G(r))] \qquad (3.3.4)$$

The equation (3.3.4) has a global optimum for the distribution of x equal to the distribution of G(r). [7]

GAN is a state-of-the-art solution in the world of data generation, and it is highly used in the case of image generation, resolution boosting, speech-to-image synthesis, and much more. The data created by a GAN network can not only be used as the final result of the processing but also as an intermediate product that's only a part of a greater machine learning application. A particularly promising application of the GAN network in an artificial intelligence pipeline is as a data augmentation module. The ability to create data samples coming from the same distribution as the training data allows the network to multiply the size of the training dataset with high-quality objects. The advantage of a GAN over standard methods (noise adding, rotation, translation) is that it doesn't affect the original data distribution at all. The dataset augmented in this way can then be used to train a neural network of higher complexity, like in [6], where a GAN delivers thousands of marked CAPTCHA images that are incredibly time-consuming to be labeled manually.

4. Project

Applying state-of-the-art neural network architecture to the Procedural Terrain Generation task is much more complicated than merely using well-known noise-based methods. It requires not only implementing the algorithm itself but also obtaining and preprocessing the training data. Adequately prepared data is crucial for any machine learning task, so all the downloaded maps have to be analyzed thoroughly to choose the best ones for the task. Implementing the algorithm is also not as easy as the classical ones because it is not guaranteed that the model will even converge, not to talk about its performance. Multiple parameters, such as the network layout, the number of epochs, the batch size, and the learning rate, have to be fitted through numerous experiments. This raises the need to monitor the training process with plots and visualizations strictly. The results have to be somehow evaluated in comparison with classical methods, so at least one of them should also be implemented. In the end, a few sample maps should be generated and visualized in a more human-friendly form than in a gray-scale.

4.1. Data preparation

The data preparation is the most underrated part of implementing the artificial intelligence model, despite its greatly influencing the results. In order to teach the neural network how the landscapes on the Earth looks like, one has to deliver the data about terrain height. In this work, the data is delivered in the form of gray-scale heightmaps in a .tif format. Pixel's shading informs about the elevation of a 1km^2 piece of terrain, and this information was scaled into the 0-6400 meter range. The elevation has been measured by space-based radars measuring the time the signal needs to bounce back.

4.1.1. Obtaining the data

NASA is the data source for this project. The heightmaps can be obtained from the "Visible Earth" collection, specifically from the "Topography" topic of the "Blue Marble" album. [8] Heightmaps are organized into eight separate files of a resolution 10800 x 10800, so they can be easily downloaded manually.

4.1.2. Data preprocessing

Images of size (10800, 10800) obviously cannot be directly used as a neural network input. They have to be standardized to a proper shape, and they also need to be filtered out of outliers because a map that contains 90% of the water (pixels of value 0) cannot teach the model anything of use. To acquire a significant amount of training data, cropping images of a chosen shape is done with overlapping. Three parameters can be set to control the cropping process: cropped image width: w_c, cropped image height h_c, and jump size: j. Given the original image width and height as w and h, the number of cropped images - n, can be calculated from (4.1.1):

$$n = \left(\left\lfloor\frac{w - w_c}{j}\right\rfloor + 1\right) \cdot \left(\left\lfloor\frac{h - h_c}{j}\right\rfloor + 1\right) \tag{4.1.1}$$

Using the jump size of 128 pixels, one can acquire as much as 47432 images of size (1024, 1024), which is a reasonable number for neural network training. However, a vast amount of these pictures do not contain any land or contain only a few percentages of non-zero pixels. Those pictures should be rejected. In this work, a minimal land percentage threshold is 30%, which allows keeping 18 013 maps.

4.1.3. Data analysis

In order to determine the proper threshold for the map filtering and to choose the jump size that provides a sufficient number of maps without too much data reproduction, there were a few experiments conducted. For a set of arbitrarily chosen map sizes, a histogram of pixel brightness has been created. It allowed for the approximation of a meaningless data reproduction scale.

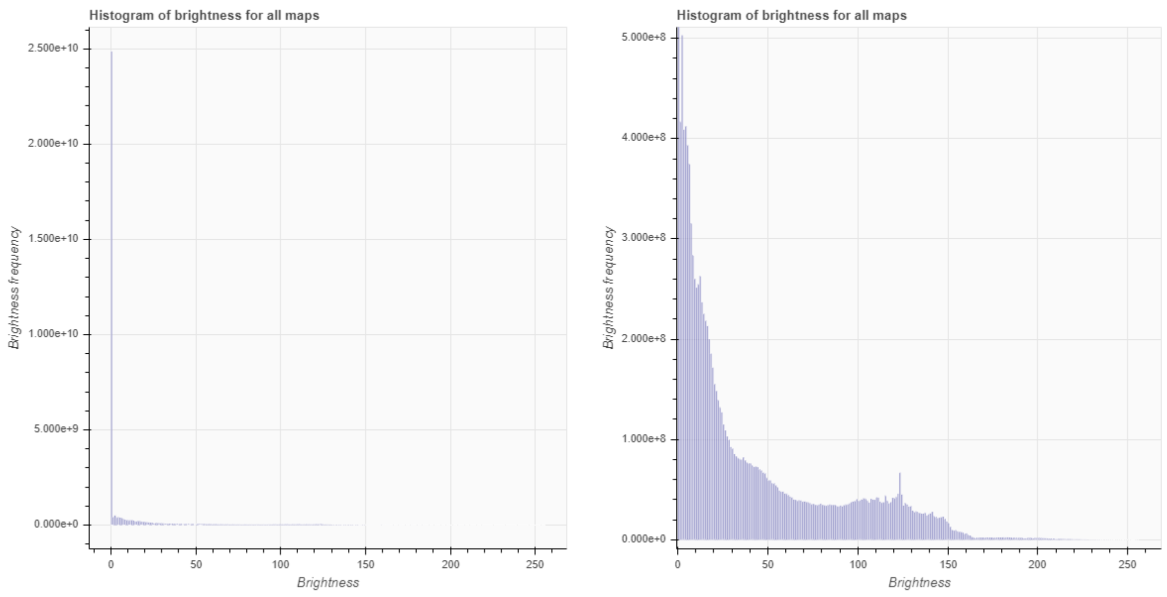

Figure 4.1.1: Example histogram of pixel's brightness (left) and same histogram without the 0 value (right).

(Figure 4.1.1) presents an example distribution of pixel brightness in the cropped data. The 0 value clearly dominates the data set, while values higher than 200 are rarely encountered. The ratio between the zero value count and the non-zero count indicates how many useless data points have been unnecessarily copied. The jump size has to be determined in a way that gives the lowest ratio possible while still providing enough training data.

A similar method was used to find the best possible minimal-land-percentage threshold. Instead of pixel brightness distribution, the land percentage on an image distribution has been plotted.

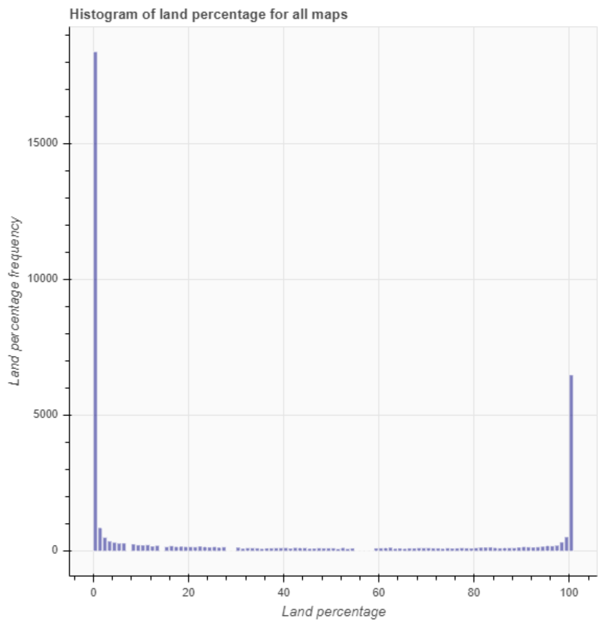

Figure 4.1.2: Example histogram of land percentage distribution over cropped maps.

(Figure 4.1.2) shows how many maps containing a certain percentage of non-zero pixels have been cropped. A vast majority of all maps (over 18 thousand) contains only a sea-level terrain (0 pixels), but there is also a large group of land-only pictures. It proves to be a problem because due to the unbalanced data set, a neural network might encounter troubles while learning the brightness distribution.

Using the combination of (Figures 4.1.1) and (Figure 4.1.2) and also a few different statistics, like mean land height on the maps calculated for a bunch of different cropping parameters, the best crop – in a subjective sense as there is no direct metric to measure it, can be chosen.

After cropping and filtering the maps out of the original data, they also have to be resized to a smaller size (in case of the final network – 128x128 pixels). Decreasing the size of the images is necessary because, in the picture of size 1024x1024 pixels, there is an immense information redundancy, which makes it harder for the network to learn. In a resized image, the number of information stays the same, but the required output's size is much smaller, which reduces the number of parameters needed. The drawback of this solution is that the output image after resizing it to the original size might be a little fuzzy, but there are solutions to this problem. However, they are not a part of this work.

4.2. Neural Network Model

4.2.1. Model architecture

As map generation's task is a generative task, the GAN architecture seemed to fit it perfectly. The GAN used in this work is composed of two convolutional-only networks – without the fully connected layers since there is no need to conduct complicated computations on a global terrain layout, and only adjacent terrain spots have to fit together locally.

The discriminator has been designed to process grayscale images of size 128x128 pixels. It comprises five convolutional layers intertwined with the BatchNorm objects and LeakyReLU objects, which means that every convolution output is normalized before being fed through the activation function. The BatchNorm normalization is a variation of a standard mean and variance normalization described by:

$$y = \frac{x - E[x]}{\sqrt{Var[x] + \varepsilon}} \cdot \gamma + \beta \tag{4.2.1}$$

Where parameters $\gamma \; and \; \beta$ are being learned through the process. The normalization (4.2.1) is used to limit the layer's output signal as the GAN networks tend to create very high results. This happens due to the fact that both networks would work on batches of the data that they are mostly right or mostly wrong about, which leads to the connection weights eruption. For the discriminator, the start of the training process is the moment when it acquires unusually high performance since its task is much easier. The activation function also has been chosen to prevent the overfitting of the network. The LeakyReLU is a modified version of a ReLU activation function presented in (Figure 3.1.2). It differs from the original one by the presence of the “leak” – a gently sloped line for the x < 0 instead of a constant 0 value.

$$f(x) = \begin{cases} x & if \; x > 0, \\ 0.01x & otherwise \end{cases} \tag{4.2.2}$$

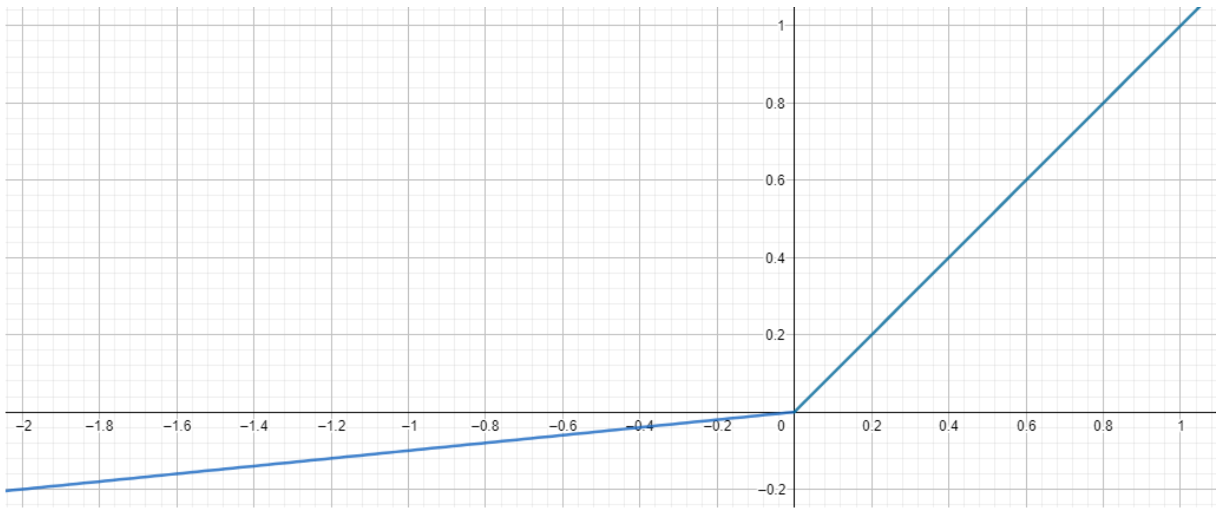

Figure 4.2.1: The LeakyReLU activation function.

This “leak” (Figure 4.2.1) allows a small, positive gradient even if the neuron is not active, which improves the network’s training progress on negative examples. The function used is not differentiable at zero (4.2.2) though it is everywhere else, and the value of the derivative at the zero point can be chosen arbitrarily.

Every convolution layer described above is used with the kernel of size 4, stride = 2, and with single zero-padding. This choice of parameters allows the network to halve the input size with every step, which means that the output is of size 4x4 ($128*(0.5)^5 = 4$). After these five convolutional layers, there is one last convolutional layer that is not normalized and uses a logistic activation function (Figure 3.1.2) instead of the LeakyReLU. This layer has no stride and no zero-padding, which means that the input of size 4x4 is converted using the kernel of size 4x4 into a single number. This number processed through the activation function is the label given by the network to the input image. A logistic activation function is used because, as one can see in (Figure 4.2.1), the LeakyReLU is not limited to the range (0,1), which is expected from a binary classifier.

The generator is a mirror image of the discriminator. It takes as the input a single channel random noise vector containing 128 samples and outputs an image of size 128x128. In order to acquire the data of proper size, the input vector is processed by transposed convolution layers. The network starts with a single layer with no stride and no zero-padding, which uses the information from the 128 input samples and transform the single channel into four channels. Then, similar to the discriminator, there are four transposed convolution layers with the kernel of size 4x4, stride = 2, and a single zero-padding, intertwined with the BatchNorm objects and activation function (in this case – a standard ReLU function). The normalization of the signal

(4.2.1) is needed in order to prevent the output of the network from "exploding" (similarly to the discriminator), and a ReLU (a constant 0 for $x < 0$ in formula (4.2.2)) is used instead of a LeakyReLU to simplify the signal processing further. Each of the transposed layers doubles the dimensionality of the processed data, which means that the output is of size 64x64 ($4*2^4 = 64$). After these layers, there is the last transposed convolution layer with the same properties as the previous ones, but without the normalization and with the hyperbolic tangent as an activation function. This activation function (Figure 4.2.2) limits the output to the range (-1, 1) - which can be easily rescaled, and distributes the data evenly in that range. As the last layer is parametrized identically as the previous four, it doubles the size of the input from 64x64 pixels into desired 128x128 picels.

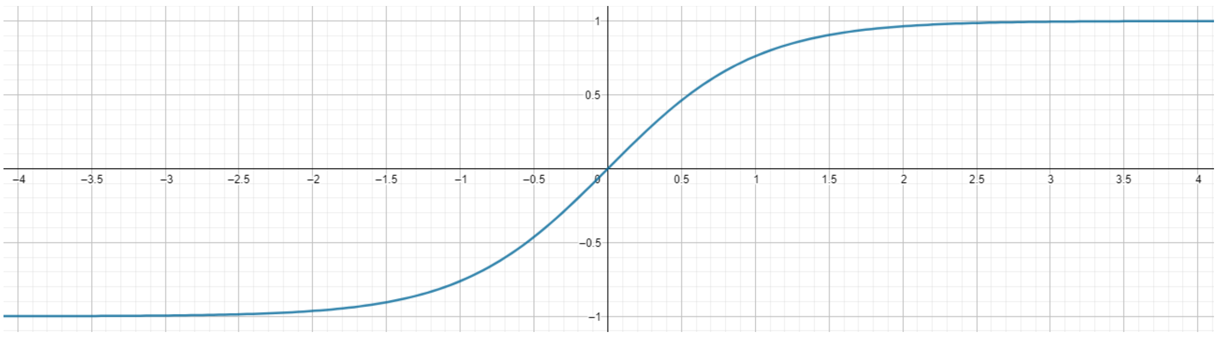

Figure 4.2.2: The hyperbolic tangent activation function.

Both networks do not use the bias connection as it is unnecessary in this kind of problem.

4.2.2. Training the model

The process of training the model requires multiple approaches to fit suitable training parameters. As the training process requires a few hours to be completed, it is impossible to choose the best possible set of parameters. Instead, one has to conduct a couple of experiments with different parameters and use a heuristic to evaluate the results and derive the rule of how the change in setting influences the results. Of course, there are some general rules of how these parameters should look like, but dependent on the problem – they may not be enough.

In the case of the problem raised in this work, the parameter fitting task is especially hard, as there is no way to compare the results of two different training processes other than comparing them manually since the results are very subjective.

In the case of the network architecture described in (Chapter 4.2.1), the simplest parameter to fit was the batch size (number of images processed through the network before the weight update). With the batch size of 64 images and higher, the network appeared not to learn at all.

The smaller the batch size, the longer the computations, but also the better the results. The final network was created using a batch size of 4 images.

Another simple parameter was the number of epochs (how many times the network should process all the training data). A range of 6-10 epochs seemed to produce the best output. However, it seemed that more extended training did not cause the network to overfit.

One of the most challenging options to choose was the learning rate – a parameter specifying the impact of every sample of the data. With the learning rate set to one, the connection's weight is updated precisely by the value of the gradient calculated from the error made. Usually, this parameter is set to a very small value, like 0.002, to "smooth" the training process and avoid overfitting (it is called "regularization"). Checking multiple combinations of this parameter for both networks is a costly process, as the values it can adopt are continuous, so only a few cases can be tested. It turns out, that in this case, the GAN works the best while both the networks learn with the same learning rate. The final results were created using the learning rate of 0.002.

4.2.3. Model evaluation

During the training, the results produced by the generator have to undergo evaluation in order to determine the moment to stop the training process and compare the results achieved between different training sessions. A standard procedure while training a GAN network is monitoring a discriminator and a generator performance because if the discriminator's performance reaches 100%, it means that the generator stopped learning anything, and the chosen architecture or the parameters do not fit the task. From the other side, the discriminator's performance at the level of 50% means that the discriminator is no longer able to recognize fakes, and the learning process has stopped. This is the reason behind using the discriminator's performance as a stop condition for the training, but this is not always the case. In the case of this work, the performance of the discriminator was steadily growing during the process, which means that the discriminator was learning faster than the generator, but it does not mean that the training failed.

As visualized in (Figure 4.2.3), the discriminator overwhelmed the generator, but it did not achieve 100% performance, so the training proceeded, and the quality of the images generated continued to grow. Since the trend presented in (Figure 4.2.3) continued for even up to 40 epochs, there was a need to stop the training manually, and in order to choose the best moment, the images generated themselves were used.

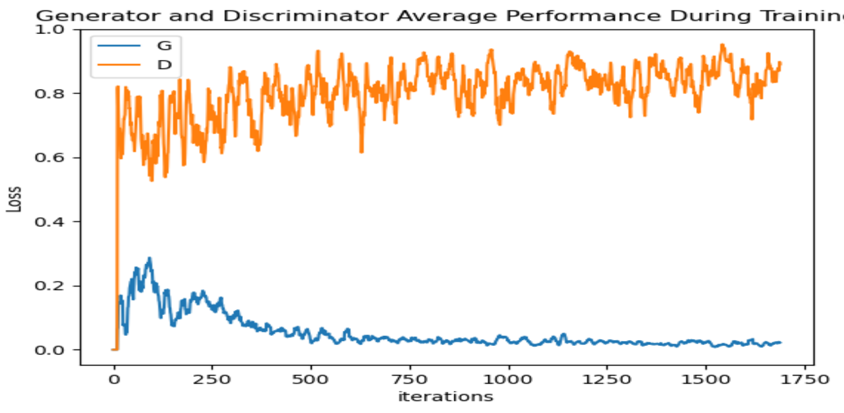

Figure 4.2.3: The performance of the final GAN.

In order to subjectively rate the quality of the results, at the beginning of the training process, a set of 16 different noise input vectors were created. These vectors, which can be called a validation set, were fed to the network every few hundred iterations. As the validation set did not change throughout the learning process, it could be used to "measure" the progress the algorithm has made.

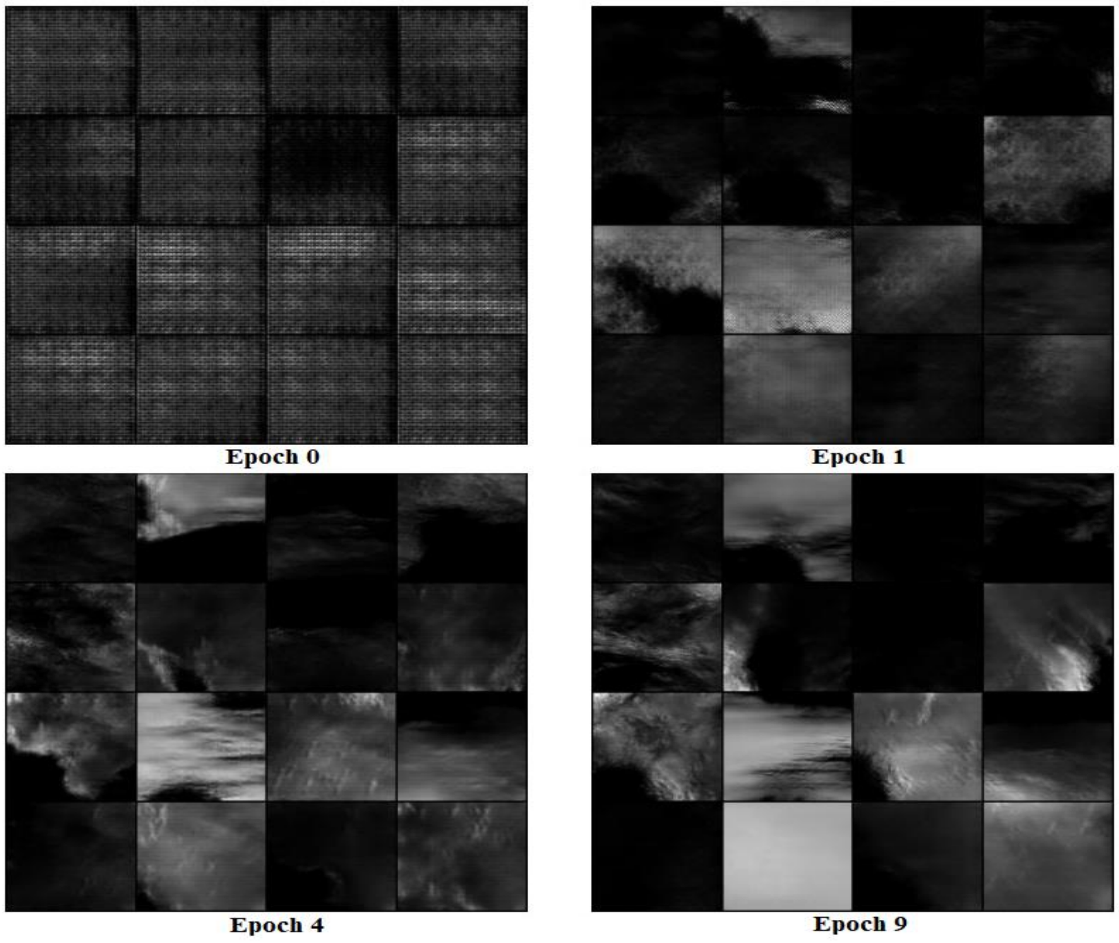

Figure 4.2.3: The results of feeding the fixed noise vector through the generator after 0 epochs (upper left), 1 epoch (upper right), 4 epochs (lower left) and 9 epochs (lower right).

Sets of images resembling these in (Figure 4.23) were created for every neural network trained while fitting the parameters. They turned out to be the best tool for rating the performance of the GAN as they present exactly what the network creates, how it looks, and whether the same output repeats for numerous input noises (if it does so, it might mean that the discriminator is being fooled by a single type of an image, and the output does not improve). By examining these pictures, one can compare two different network architectures and determine the end of the training. When the images generated does not change much between epochs, the training can be stopped.

The progress made by an example model is presented in (Figure 4.2.3). At the beginning of the training, the generator was producing only a strange kind of noise, but that changed as soon as after the first epoch when it started creating land-shaped pictures containing simple coastlines, but with no mountains and other land features. These pictures were also contaminated by some kind of artifacts in the form of visible squares of different brightness. After a few epochs, the network learned how to create a proper, naturally-looking coastline and started to add mountain ranges into images. The artifacts almost vanished. At the end of the training, the generator became proficient with creating well-shaped coasts and mountain ranges, which resulted in maps that can be easily confused with the training data.

4.3. Perlin's Noise model

In the project, there was also a Perlin's Noise generation model implemented to provide template images that can be used for comparison between the results obtained using a neural network approach and those generated with classical algorithms. The model was implemented with a Python library called "noise" that delivers out-of-the-box functions creating the noise with given parameters. The library allows for the generation of a two-dimensional Perlin's Noise in a single line of code. It is a convenient solution under the terms of this work, as the algorithm's implementational nuances are not important. What matters is to generate the images in a properly optimized manner, which the chosen tool accomplishes.

4.4. Visualization of the results

When the model is trained, it can generate heightmaps resembling the natural terrain arrangement. These heightmaps are created in the form of grayscale images, which can be easily used in further terrain processing or, for example, as a board in a strategic game. However, if the image itself is the target product, it has to be visualized in a more human-readable way. The easiest way of doing so is by coloring the achieved image (Figure 4.4.2). Even though colorful images are prettier and better separate the land from the water, due to the simplicity of the method used, they also bring less information, which can lead to the vanishing of some terrain features, and as a result, may complicate the quality comparison. This is the reason why raw grayscale images were used for a comparison between maps generated by a GAN, and these created using Perlin's Noise (Figure 4.4.1).

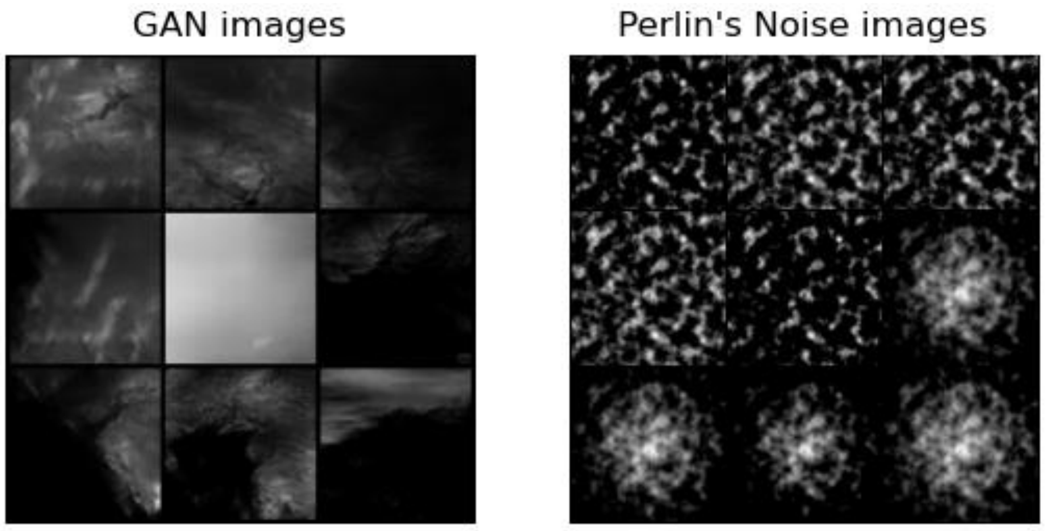

Figure 4.4.1: Images generated using a GAN network (left) and using a Perlin's Noise (right).

As one can see, Perlin's Noise, without further processing, creates very irregular structures that resemble archipelagos of islands with slightly more jagged coastlines than in reality. The noise can be easily processed to acquire a single island instead of archipelagos, with the result resembling a volcanic island, with a single mountain in the center. An example can be seen in the last four pictures in (Figure 4.4.1). The slope certainly has some natural features, but it resembles only a very simplified representation of real-world landscapes. The results obtained using a neural network are incomparably better. They truly mirror images delivered by the satellite, and it is almost impossible to tell one from the other.

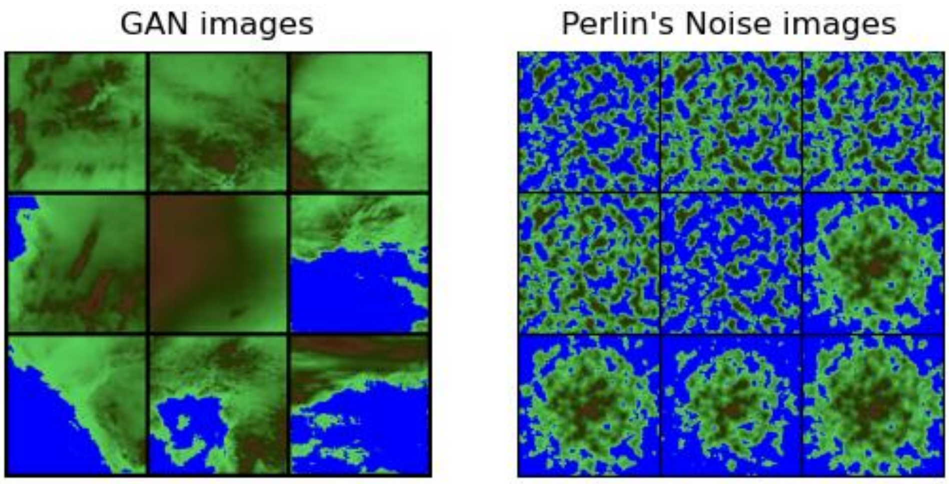

Figure 4.4.2: Colored comparison of images generated by a GAN network (left) and using Perlin's Noise (right).

(Figure 4.4.2) presents the same maps as (Figure 4.4.1), but colored using a simple, hand-crafted colormap. These colored pictures allow for easier observation of the coastline's shape and high-slope regions location than the grayscale ones but at the cost of a detail loss (mountain ranges, that line-up in generated pictures just as they tend to do in reality looks like spots of high-level pixels, just as in the case of Perlin's Noise).

Another way to visualize the generated map is to create a three-dimensional model that may be a more natural interpretation of a heightmap. An example model is presented in (Figure 4.4.3).

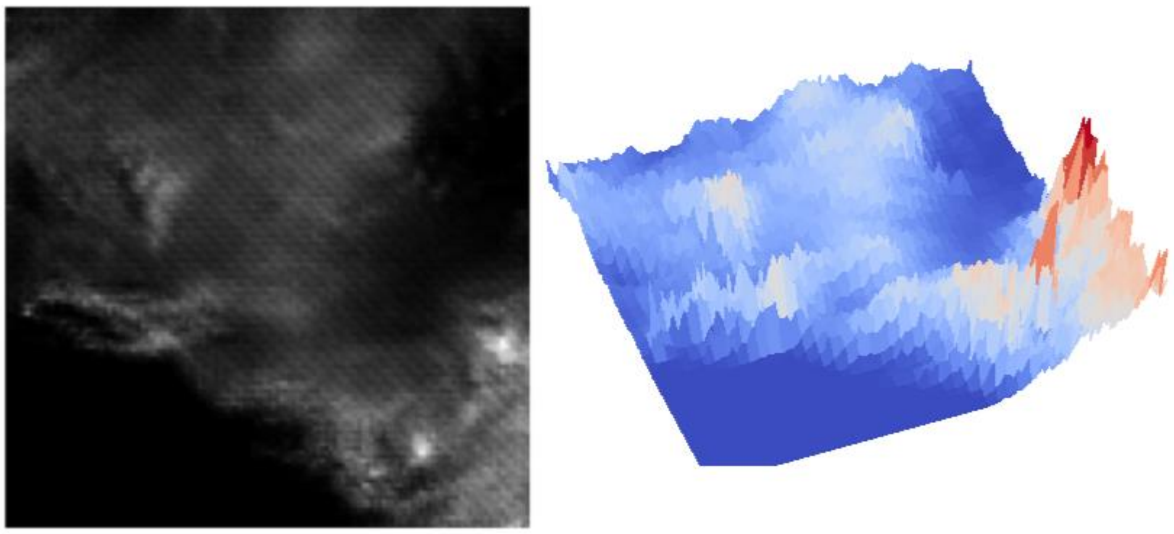

Figure 4.4.3: An example three-dimensional model (right) of a heightmap (left).

5. Conclusions

The GAN network designed and trained for the project meets the objectives of this work. It proved that imitating the land arrangement on the planet Earth is possible and can be done autonomously. The network not only learned how the coastline should look like, but it also is able to add some extra features like bays, peninsulas, fjords, and additional islands that further enrich the resulting terrain. It also managed to learn how to place mountains, and in some cases, even rivers, which results in maps that fully reflect the natural look of the Earth's landmasses. Some examples of the maps created by the algorithm can be seen in (Figure 5.1):

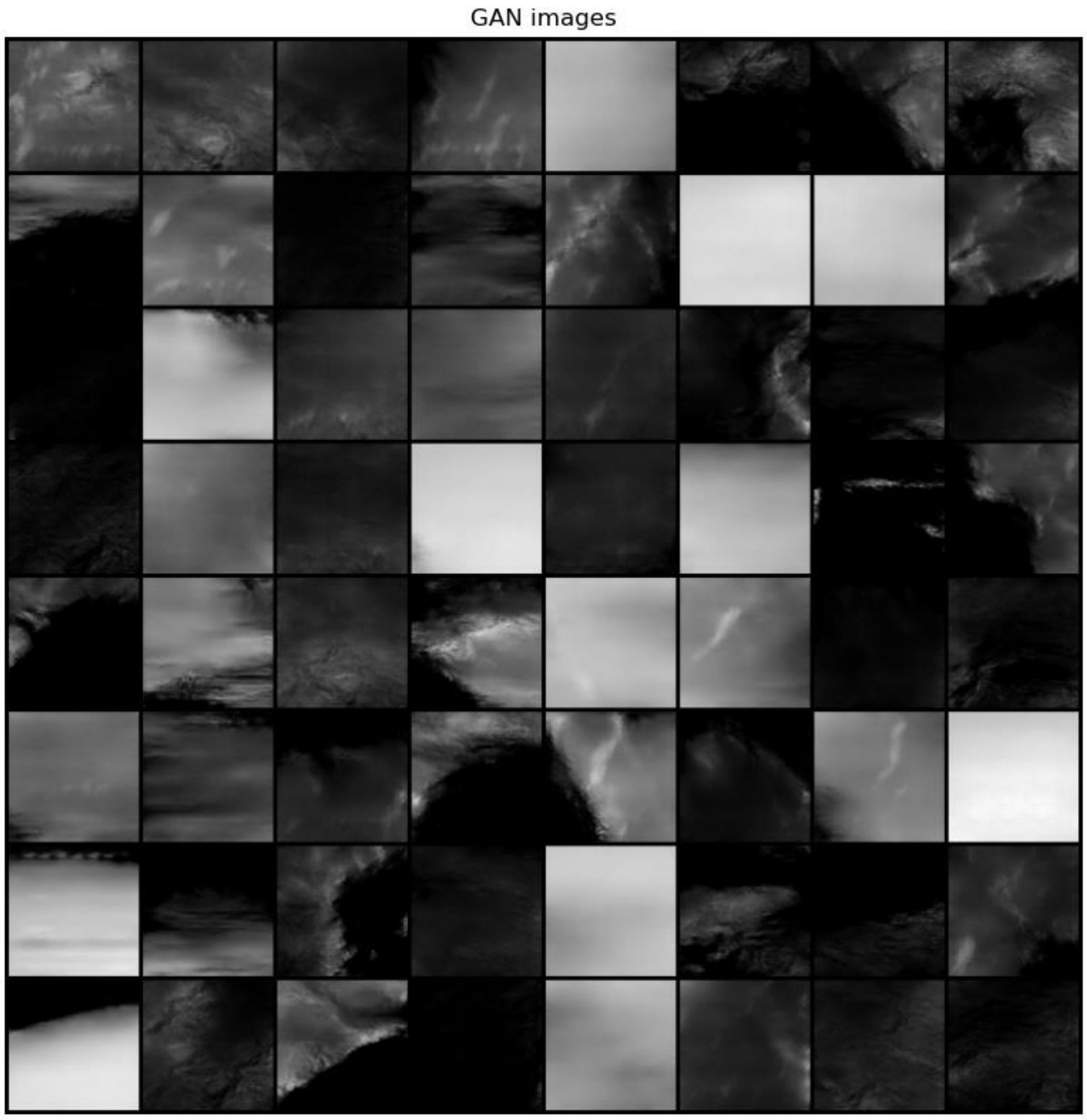

Figure 5.1: A collage of 64 example images generated by the final network.

Maps generated by the final network (Figure 5.1) compared to the results obtained using Perlin’s Noise algorithm look much more natural. More Earth-like coastlines characterize the GAN's results. Moreover, they do not need any further processing as all the terrain features, including mountains and rivers, are generated right away. Nevertheless, there are also drawbacks of the neural network approach. One of the biggest is the fact that the user has no control over the type of terrain generated as it is impossible to find out how changes in the input noise affect the average height in the output. Another one is the number of details the maps are created with. In many applications of the Procedural Terrain Generation, especially in the gamedev, detailed maps might be undesirable as, for example, something unnecessarily complicating the gameplay.

In terms of computational performance, the results are twofold. Creating a single map using Perlin’s Noise (around half a second) is faster than with a neural network (around a second), but due to the high parallelization ability of the neural nets – the more maps one wants to create, the faster the GAN is. For instance, generating one hundred maps at once using the network still takes around a second, while the Perlin’s Noise requires as much as 40 seconds. Costs of additional processing of the noise-based method results also have to be taken into consideration, which in some applications makes the neural-based algorithm faster.

Summing everything up, it is possible to use neural networks in the task of the Procedural Terrain Generation. The results obtained using this approach are of high quality, and they can be easily used in many fields, from art to the gamedev. However, there are some major drawbacks of the method, they can probably be dealt with, which leaves space for further research in the field, which – given the quality and performance gains, are worth to be conducted.

6. Bibliography

[1] Celusniak, Martin. 2019. *Cave-like Level Generation Using Cellular Automata.* March 19. https://www.linkedin.com/pulse/cave-like-level-generation-using-cellular-automata-martin-celusniak/.

[2] Christopher Beckham, Christopher Pal. 2017. *A step towards procedural terrain generation with GANs.* July 11. https://arxiv.org/abs/1707.03383.

[3] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. 2003. *Texturing & Modeling: A Procedural Approach, Third Edition.*

[4] Eric Guérin, Julie Digne, Eric Galin, Adrien Peytavie, Christian Wolf, Bedrich Benes, Benoît Martinez. 2017. *Interactive Example-Based Terrain Authoring with Conditional Generative Adversarial Networks.* September 28. https://hal.archives-ouvertes.fr/hal-01583706/file/tog.pdf.

[5] Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning.* MIT Press.

[6] Guixin Ye, Zhanyong Tang, Dingyi Fang, Zhanxing Zhu, Yansong Feng, Pengfei xu, Xiaojiang Chen, Zheng Wang. 2018. *Yet Another Text Captcha Solver: A Generative Adversarial Network Based Approach.* Toronto, October. https://www.researchgate.net/publication/328322917_Yet_Another_Text_Captcha_Solver_A_Generative_Adversarial_Network_Based_Approach.

[7] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio. 2014. *Generative Adversarial Nets.* Montreal, June 10. https://arxiv.org/pdf/1406.2661.pdf.

[8] Jesse Allen, NASA's Earth Observatory. 2005. *Topography.* July 21. https://visibleearth.nasa.gov/images/73934/topography.

[9] Julian Togelius, Noor Shaker, Mark J. Nelson. 2016. "Procedural Content Generation in Games." 1-2.

[10] Michalak, Grzegorz. 2012. *Algorytmy proceduralnego generowania rzeczywistości na przykładzie dwuwymiarowej gry cRPG.* Łódź. http://www.michalak.net.pl/download/Praca_magisterska.pdf.

[11] Moss, Richard. 2016. *7 uses of procedural generation that all developers should study.* January 1. http://www.gamasutra.com/view/news/262869/7_uses_of_procedural_generation_that_all_developers_should_study.php.

[12] Olsen, Jacob. 2004. *Realtime Procedural Terrain Generation.* October 31. https://web.mit.edu/cesium/Public/terrain.pdf.

[13] Perlin, K. 1985. "An image synthesizer." *SIGGRAPH '85.*

[14] Supasorn Suwajanakorn, Steven M. Seitz, Ira Kemelmacher-Shilzerman. 2017. *Synthesizing Obama: Learning Lip Sync from Audio.* Washington. https://grail.cs.washington.edu/projects/AudioToObama/siggraph17_obama.pdf.

[15] Tero Karras, Timo Aila, Samuli Laine, Jaakko Lehtinen. 2017. *Progressive Growing of GANs for Improved Quality, Stability, and Variation.* October 17. https://arxiv.org/abs/1710.10196.

[16] Togelius, J., Champandard, A.J., Lanzi, P.L., Mateas, M., Paiva, A., Preuss, M., Stanley, K.O. 2013. "Procedural content generation: Goals, challenges and actionable steps." *Dagstuhl Seminar 12191: Artificial and Computational Intelligence in Games.* Dagstuhl.

[17] Vincent Dumoulin, Francesco Visin. 2018. *A guide to convolution arithmetic for deep learning.* January 12. https://arxiv.org/abs/1603.07285.