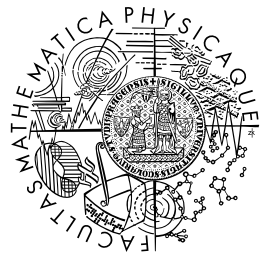

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Filip Sedlák

Remixing OSM maps using recurrent neural networks

Department of Software Engineering

Supervisor of the bachelor thesis: RNDr. Miroslav Kratochvíl
Study programme: Computer Science
Study branch: IPSS

Prague 2021

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank my supervisor Miroslav Kratochvíl for valuable advice. Next I would like to thank my family and friends for support during my study.

Title: Remixing OSM maps using recurrent neural networks

Author: Filip Sedlák

Department: Department of Software Engineering

Supervisor: RNDr. Miroslav Kratochvíl, Department of Software Engineering

Abstract: Generation of random realistic maps is a highly desirable content creation method for entertainment industry. Neural networks provide powerful computational capabilities proven useful in many fields. This thesis describes an algorithm that adapts real-world data to train Recurrent Neural Networks (RNNs) inspired by the pixel RNNs. An algorithm is constructed to generate a map of altitudes, roads, rivers and buildings. The results are tested and evaluated on multiple selected real-world regions. It shows the ability of RNNs to learn and create random realistic maps. Algorithm generates realistic altitude maps reflecting user input and training dataset. The creation of roads and rivers was met with weaker results. The creation of buildings was met with unsatisfactory results.

Keywords: recurrent neural networks, open street map, random generated media

Contents

Introduction

Generation of random realistic maps is a highly desirable content creation method for entertainment industry. For instance, open-world games or large scale movies often need to provide a map of size that is nearly impossible to be created manually.

There are multiple viable methods for generation including noise functions [1], erosion algorithms [1] or genetic algorithms. [9] These algorithms generate or improve terrain in the form of an altitude map. Realism of the output is usually improved by using real-world data as a basis for map generation, such as freely available OpenStreetMap (OSM) data. It is difficult for an artist to capture complex natural features by hand. A helpful alternative might be an algorithm that generates maps based on features learned from real-world training examples. This thesis explores the possibilities of applications of neural networks in the field of terrain generation with additional objects, such as roads, rivers and buildings, providing reasonable user control.

Recurrent neural networks (RNNs) provide powerful widely used method for deriving random remixed content from training datasets. However, direct application of OSM data is complicated due unfitting vector and graph based data representation and complexity of the OSM annotations. This is a problem, because neural networks expect input as a simple series of numbers.

This thesis describes an algorithm that adapts OSM data to RNN input. Networks are inspired by pixel recurrent neural networks, as published by Van Oord, Kalchbrenner and Kavukcuoglu[14]. Next, these networks are used to generate an altitude map with iterative addition of other elements. Roads and rivers are generated based on the altitudes. Buildings are generated based on altitudes, roads and rivers. The deep learning approach is popular in the field of machine learning. However, we have decided to avoid it because of its performance, which conflicts with our goal of terrain generation being interactive.

This approach is successful in creation of realistically looking altitude maps that reflect training dataset and user input. It was less successful in creation of more complicated features.

This thesis is organized as follows. The first chapter is composed of intro-

duction to neural networks, recurrent neural networks and overview of existing map generation algorithms. In the second chapter, this thesis introduces nature of available data and describes the broad design of the map generation algorithm. The third chapter describes specifics of implementation regarding RNNs structure and used software tools. In the final chapter, RNNs are tested and evaluated on multiple selected OSM regions and user inputs.

Chapter 1

Neural Networks for map generation

This chapter describes what is a neural network and recurrent neural network. It also describes existing map generation methods.

1.1 Artificial Neural Networks

Definition 1 (Artificial neuron)**.** *Artificial neuron, or neuron for short, is a mathematical function based on biological neurons. [10] Neuron is defined by weights* $(w_0, ..., w_n)$*, activation function* A *and bias* b*.* A *usually takes form of any nonlinear function. Given inputs* $(x_0, ..., x_n)$*, the neuron computes the output as follows:*

$$y = A(b + \sum_{i=0}^{n} x_i w_i)$$

Definition 2 (Artificial neural network)**.** *Artificial neural network, or neural network for short, is a circuit of neurons. [10] These neurons are organized into layers* $(L_0, ..., L_n)$*. Layer* L_0 *is an input layer. Layer* L_n *is an output layer. Layers between the input and the output layer are called hidden layers. Outputs of each hidden layer* L_i *are used as inputs for layer* L_{i+1}*.*

More appropriate and simpler view of neural networks in relation to this thesis is to think of them as function approximations. Zainuddin and Pauline[15] describes good explanation of this view. To summarize, real world problems often present themselves in a form of a mapping between an input and an output space with a set of input-output data. However, often there is no known explicit formula to describe the function f to solve such problem. There is only an available subset of input and corresponding output data. Function approximation is

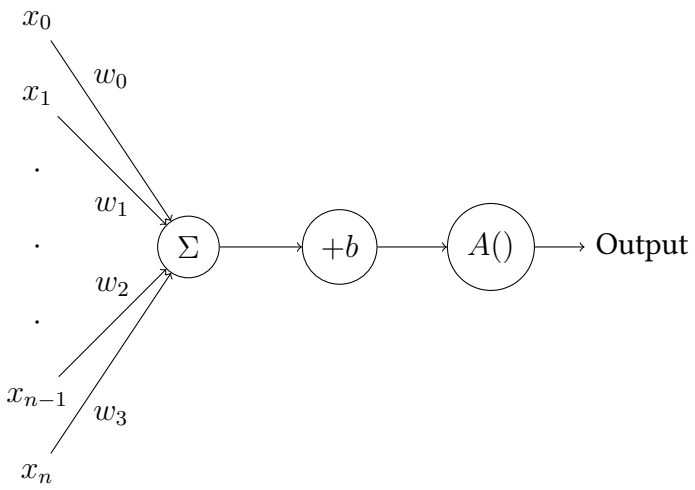

Figure 1.1 This figure shows computation of a single neuron.

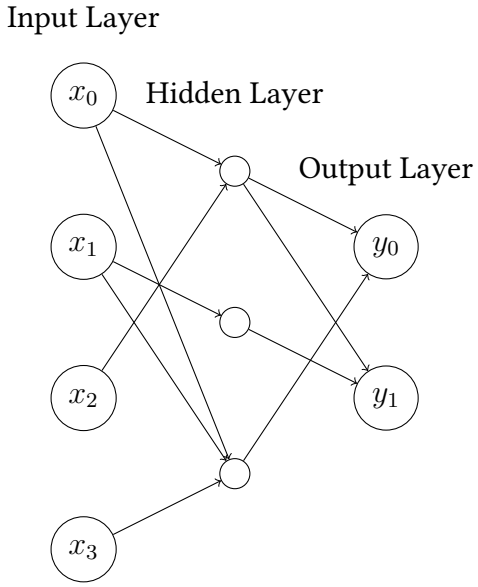

Figure 1.2 This figure shows the structure of a simple neural network. x_0 .. x_3 are inputs. y_0 and y_1 are outputs. Each circle represents a neuron.

a method to approximate function f. Given a set of input data I and output data O, an error function $E(f)$ is selected. E calculates how far f is from the perfect function for the given dataset $I \rightarrow O$. The error function is also called loss function in context of neural networks. The goal of the approximation is to minimize this error for all kinds of data. There are many ways to minimize the loss for neural networks. One popular approach is Stochastic Gradient Descent.

Theorem 1 (Universal Approximation Theorem)**.** *Given a real continuous function f and $\epsilon > 0$ there exists a neural network with one hidden layer that approximates f within the precision of ϵ.* [1]

Neural networks are one such function approximation method. According to Zainuddin and Pauline[15], they are capable to approximate arbitrary continuous function with any desired accuracy without predetermined models. This is tied to Universal approximation theorem1 proven by Cybenko[3]. Such ability makes them powerful among other approximation techniques. But they also require much more computational power. Neural networks are frequently used to solve various complex problems, such as:

- As time series predictors, they are able to predict the future of sales in business for instance. The approximated function is $f(x) = y$, where x are sales from the past and y are sales in the near future. [13]
- They are being used to solve problems relating to visual data. An example of this is image classification, where x is a rasterized image and y is a certain class. One problem of this nature is hand written digit classification. [7]

1.2 Recurrent Neural Networks

Definition 3 (Recurrent neural networks)**.** *Recurrent neural networks are neural networks that use their own previous outputs as inputs.*

One iteration on previous input is called a step. These steps in computation are usually called timesteps. Such network structure allows for processing inputs of arbitrary length without expanding its size. They also allow, but do not demand, capturing historical information of the previous inputs using hidden layers. This allows capturing dependencies in a sequence of inputs, where each input is dependent on previous ones. Although, the history is limited. Drawback is that the training is very slow compared to other networks. "The butterfly-effect" is one reason. A small change in early timestep can create very large

[1] For a more formal treatment of the Universal Approximation Theorem refer to Cybenko[3].

changes many iterations later. This implies that the derivative of the loss function at one time can be exponentially large with respect to the hidden activations at much earlier time. RNNs also suffer from the vanishing gradient problem. This means that in some cases the gradient is vanishingly small to affect weights and thus preventing further training. Sutskever [12] describes all of this and expands on further complexity of the structure of RNNs. This thesis only uses the most basic idea of recurrence, as defined above in Definition 3.

RNNs have proven useful in fields of speech recognition [2], natural language processing, and even language generation. [6] Van Oord, Kalchbrenner and Kavukcuoglu [14] also describe a way to use RNNs for generating partial images. This approach is called pixel recurrent neural network. This is important, as it is used as an inspiration for this thesis in further chapters. Given a part of some image, this algorithm tries to generate the rest of the image in a row by row fashion. By giving the network a certain area that has already been drawn, it tries to guess what the next pixel should be. It then works on its own outputs to generate the rest of the image row by row.

1.3 Map Generation Methods

Video games with very large amounts of content are often in need of creating content procedurally for various reasons. In games, it can be a source of randomness, to always give the player a new experience. Viewing map generation as a construction of an altitude map image, algorithms for generating graphical data can also be used effectively. There are multiple viable methods of generating a map. This section sums up and describes some of the most relevant ones.

Noise functions: In the field of computer science, noise is used for procedural generation of 2D and 3D models. Some examples of this are texture models and, more relevant for us, terrain models (especially landscapes). The downside of this approach is that it is difficult for the user to control. The speed of the algorithm varies. Fastest algorithms can generate terrain in real time using a chunk system.[2] Most recognizable noise functions are Perlin Noise and Simplex Noise. [1]

Erosion Algorithms: Erosion algorithms are used to further modify a model to give it more worn and realistic appearance. Such algorithms usually work by transferring material (soil) at steep slopes to make changes in inclination. Some examples of this are hydraulic and thermal erosion. [8]

[2] `https://minecraft.fandom.com/wiki/Seed_(level_generation)`

Wave Function Collapse: WFC functions are somewhat similar to neural networks. WFC learns from a small input image. Algorithm then generates patterns based on that input.

The learning process consists of two steps. The first step is dissecting and dividing the the image into small samples. The second step is recognizing rules, describing which samples can be placed next to each other. The map is then filled based on the rules gathered.

As Opposed to neural networks, they have restrictive and deterministic learning. WFC can be also highly controlled.

This algorithm is largely effective for generation of pictures that have repeatable patterns. It is also effective for generation of tile based maps, such as with repeatable objects, buildings, rooms and hallways.

However, it is not very suitable for natural terrain generation. [11]

Genetic Algorithms: As an example, Ong et al. [9] describes a genetic algorithm that allows for terrain generation, where it is possible to give the user enough control without overwhelming them or requiring knowledge about the algorithm. There are, however, limits to what genetic algorithms can create regarding natural structures.

Cellular Automata: Cellular Automata are often used in graphics as an erosion algorithm as specified above. They are also used for environmental systems, like fire, fluid flow etc. Johnson, Yannakakis and Togelius used it to generate complete 2D maps. It has shown capabilities of generating natural cave-like levels. This approach is fast and can generate map in real time. This is especially handy in rogue-like games, where is an infinite number of levels and need to be generated on demand. [4]

Chapter 2

Generating maps with RNNs

This chapter specifies the problem and introduces the proposed solution.

2.1 Problem specification

This thesis explores the possibilities of application of neural networks in the field of terrain generation with additional features, such as roads, rivers and buildings, providing reasonable user control.

Procedural map generation, closely tied with image generation, is required in large games, in which content is needed to be generated as it is almost impossible to be created manually. There are multiple algorithms to solve this problem mentioned earlier.

Realism of the output is usually improved by using real-world data as a basis for generation, such as the freely available OpenStreetMap (OSM) data. As map generation is closely tied to image generation, Pixel recurrent neural networks, published by Van Oord, Kalchbrenner and Kavukcuoglu[14] provides a good basis for such algorithm. The direct application of OSM data is difficult due to its vector and graph representation. The first problem is how to transform the data into a rasterized image well suited for pixel RNNs.

With transformed data, this thesis focuses on constructing simple pixel RNNs to generate an altitude map. As RNNs are difficult to train well, multiple approaches are described to further preprocess the data. This makes it easier for the networks to digest the input. Similarly, roads, rivers, and buildings are iteratively generated.

2.2 Available data

OpenStreetMap is a collaborative project to create free editable map of the world [1]. This means that it contains large open source collection of world map data. In particular, it contains information about things on Earth's surface, such as streets, roads, buildings, lakes, etc. This information is encoded in an OSM file. The OSM file is specific kind of XML file. [2] Such file is composed of elements.

There are 3 important elements:

- **Node** represents a single point on the map. It contains the unique identifier of the node, latitude and longitude of the represented point and additional information about it. [3]

- **Way** contains a list of unique identifiers of nodes that represent either an open way, a closed way or a polygon of an area, depending on tags present in the way's definition. It can be also implied by the type of the element. [4]

- **Relation** contains a list of unique identifiers of nodes, ways and other relations. Relations are used to to define logical or geographic relationships between other elements. [5]

However, OSM data do not contain altitudes. OpenTopography is an NSF EAR project that provides community access to high resolution Earth topography data. [6] In particular, it provides us with an altitude map. This can be used as a replacement for missing altitude data in OSM data.

OSM contains overwhelming amount of data. The type of map attempted to be created in this thesis is a map of cities based on other map elements. Altitudes being the obvious one, cities are mostly built near a source of water. Rivers are chosen for this reason. Historically, cities were built on routes, to be connected to civilization. Therefore, roads are the next logical choice. And lastly buildings are chosen to represent cities or settlements. There are other possibilities to consider, however, these 4 types are enough for demonstration purposes. It is also enough in a lot of games.

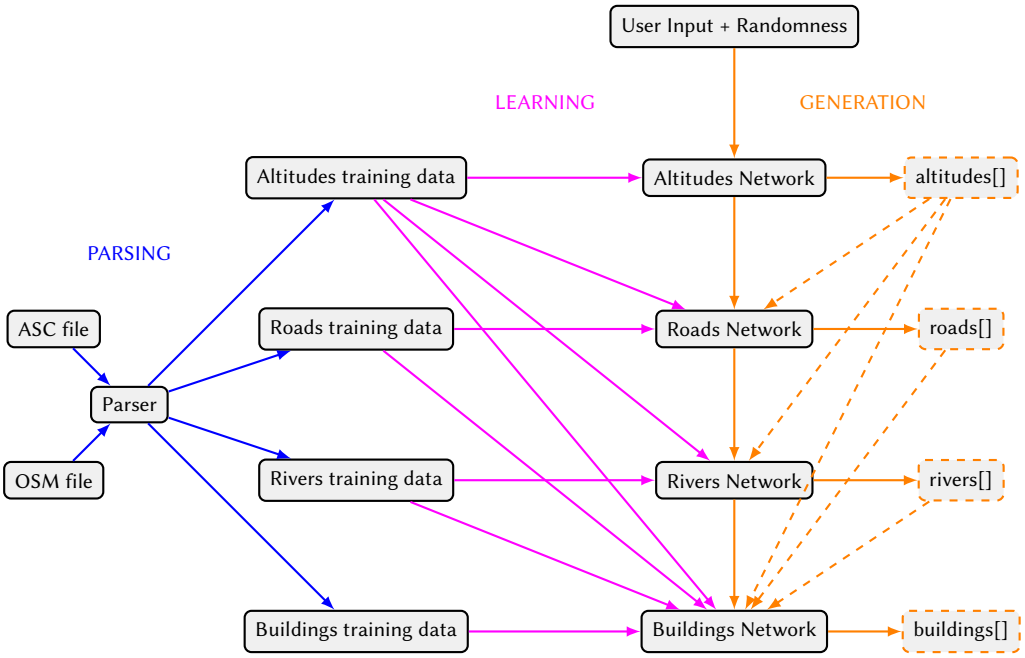

Figure 2.1 This figure shows broad representation of the entire algorithm used in this thesis. Every row in Generation phase is a stage. Each stage contains neural networks. Networks need to be trained first. The output of each stage is then also used as an input for the next stage. The exception is that generation of rivers does not take roads as its input.

2.3 Network design

The output of the algorithm is generated in stages. These consist of generating visual images of altitudes, rivers, roads, and buildings. The Figure 2.1 shows this structure. Each stage contains multiple neural networks. The final output is a set of the outputs of the stages.

The common structure of each network can be seen in Figure 2.2. The input consists of a context and a recurrent input. Given a part of an image, for example an altitude map, the recurrent input represents a narrow area of the already generated part. This area shall be called the window. The output of the network is a single pixel. The window moves in a row by row fashion, generating the map pixel by pixel. The map being generated is called generation matrix in this thesis. Figure 2.3 shows this process, as well as the shape of the recurrent window and the output. This design is inspired by pixel RNNs. [14] The context represents the wider area, so the network has a better understanding of what to generate in larger scales. This is explained in the next section.

Neural networks in general are effective working with ranges $(0, 1)$ or $(-1, 1)$. Because of this, images are represented by matrices with numbers between 0 and 1. In the case of an altitude map, such floating point number represents the altitude at a given location. The value 1 represents the maximum altitude in the original image. The value 0 represents the minimum. In the case of other features, roads for example, the value represents the probability of the feature being there. Once a window is cut out from the matrix, it is flattened into an array to be used as an input for a network. There are more ways to process such input in order to reduce potential work the hidden layers might have to do. In the case of altitudes, network needs to be able to generate certain patterns with respect to any current altitude level. For instance, to generate a recess for a lake in the mountains as well as in some lower placed meadows. This is achieved by taking the average of the given input window and making it as only absolute value. This value is subtracted from all the values of the window, making them relative. The absolute value is appended to the input to retain knowledge about the current altitude.

The values themselves can still be processed further. Unary encoding of length L is used for segmentation of the value range into multiple neurons. This makes the neural network much more aware of the value. [5] Some values behave

[1] `https://www.openstreetmap.org/`
[2] `https://wiki.openstreetmap.org/wiki/OSM_XML`
[3] `https://wiki.openstreetmap.org/wiki/Node`
[4] `https://wiki.openstreetmap.org/wiki/Way`
[5] `https://wiki.openstreetmap.org/wiki/Relation`
[6] `https://opentopography.org/`

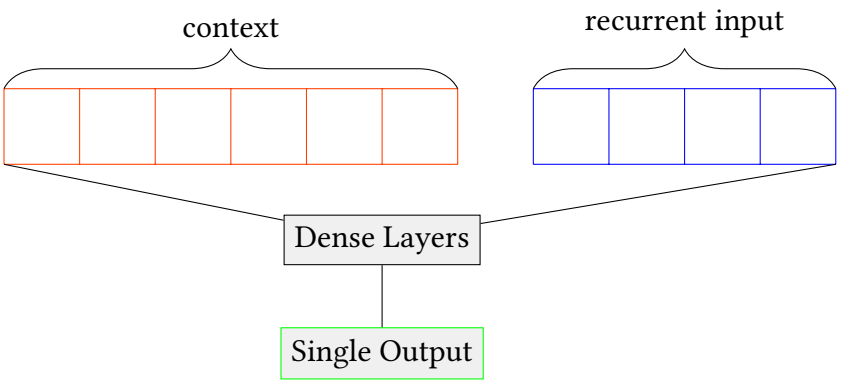

Figure 2.2 This figure shows the general structure of networks used in this thesis. Input consists of a context and the recurrent input. Context gives the network general idea of what to generate. The outputs from previous stages are also included here.. The recurrent input consists of previous outputs, making the network recurrent.

exponentially. For this reason, relative altitudes and probabilities are encoded logarithmically: $x = -\log_2(\texttt{value})$. For example: $3.32 \approx -\log_2(0.1)$. The value x is then unary-encoded as an array of length L: $[1, 1, 1, 0.32, 0, 0]$. Negative values are simply encoded as an array with negative numbers instead of positive. However, this encoding has certain reverse effect. The further the values in the array are from 0, the closer the original value was to 0. This resulted in generated images containing artifacts (for example sudden changes in altitude). Therefore, reverse encoding is used: $[0, 0, 0, 0.68, 1, 1]$. Some other values, like the absolute altitude, behave linearly. These are unary-encoded as well, except for taking the logarithm, they are multiplied by $L : x = \texttt{value} \cdot L$. In this thesis, $L = 8$. This choice is explained in the next section. However, there arises a problem with reconstruction of a value from the network. Networks generate an array of floats, which can be far from proper encoded form. Given a generated array $\texttt{arr}$, the exponential reconstruction is as follows: $\texttt{result} = 2^{-|\sum(\texttt{arr})|}$. This works quite well for features except for altitudes. Altitudes can also generate negative values and are encoded in reverse. The following algorithm provided results without any artifacts: $\texttt{result} = \Sigma_{i=0}^{L} \texttt{sign}(\texttt{arr}[i]) \cdot (-2^{-|\texttt{arr}[i]-i|} + 2^{-i})$. This simply reconstructs each value separately by its position in the array and sums the results.

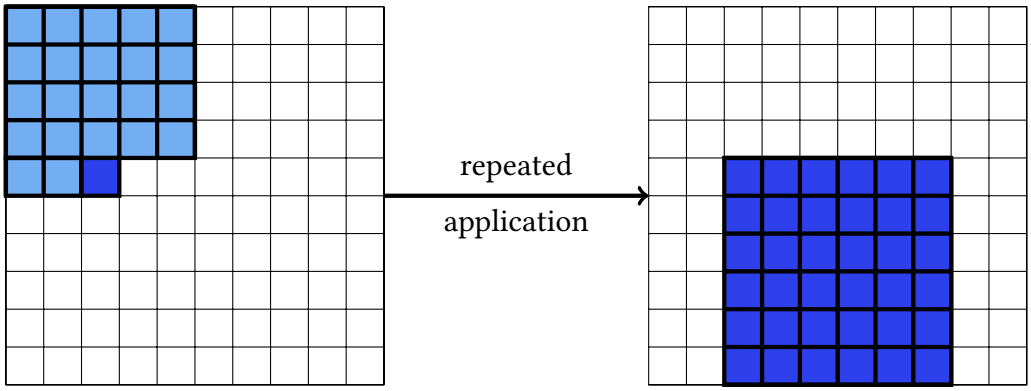

Figure 2.3 This figure shows the basic generation algorithm using pixel RNNs. On the left hand side, light blue cells are used as the recurrent input for the network. The blue cell is the output of a single step. This window then moves in a row by row fashion, filling the matrix with the outputs. On the right hand side, the blue cells are the output of the algorithm. This sub-matrix is cut out. White cells on the right hand side were only read from. These need to be somehow initialized in advance.

2.4 Additional Layers

Definition 4 (Layer-Nx)**.** *The set of inputs representing an area N-times wider than the base area is called Layer-Nx.*

In general, this thesis uses layers with N being a power of 4. Each stage is generated in multiple layers. First, the user provides a rough sketch of what to generate. This is in form of an image I. An image representing Layer-16x is generated from Layer-64x. Generation matrix needs to be created to begin the generation. This is done by scaling down I using linear interpolation. Then the reflection padding is added to fill in the edges. Demonstration of this is shown in Figure 2.3. Next, Layer-64x is created from I, taking 320×320 windows and scaling them down to 5×5. For this purpose, I is also padded with the reflection padding. As the output should be image representing Layer-16x, it is 16-times smaller than I. Therefore, the contexts are taken only at every 16th cell. Finally, using algorithm described in Figure 2.3, output is generated. It is called L_{16}. Figure 2.4 shows this process.

Next, as shown in Figure 2.5, Layer-4x is generated. Networks generating this layer are taking both Layer-16x and Layer-64x as context. As Layer-16x is already generated, we can omit I and focus on recurrent generation. As the result should be an image that is 4-times greater than L_{16}, L_{16} is scaled up four times to provide sufficient number of contexts. Layer-64x at this point is represented by the 16-times larger (80×80) windows from scaled up L_{16}. Similarly, Layer-16x is

represented by (20×20) windows. The resulting image of this process is called L_4.

At last, L_1 is generated. Networks generating this layer are taking Layer-16x and Layer-4x as context. This is achieved by scaling up L_4 with value of 4, and using the same algorithm as before. The resulting image of this process is the final image in full resolution called L_1. In the case of altitudes, random noise is added to I, L_{16}, and L_4 after scaling them up. This noise has exponential distribution with value of 20% of the distance between maximum and minimum of the image.

Roads, rivers, and buildings also take context from previous stages in a similar manner. For instance, the network generating L_{16} road map does not only take Layer-64x of roads as its context, but also Layer-64x and Layer-16x of altitudes. Refer to Figure 2.1 for map of context dependencies. As generating these features appeared to be too difficult problem for the networks to handle, an additional layer was added. Generating of L_1 is split into 2 steps. First, $L_{blurred}$ is generated, creating rough sketch of probabilities. Then an additional network generates L_{sharp} based only on $L_{blurred}$. This results in roads, rivers, and buildings being actual lines and dots instead of smudges on the map.

The highest layer used in this thesis is Layer-64x. As the original images of roads, rivers, and buildings contain only zeros and ones, the smallest number that can appear in Layer-64x is $1/2^7$. For this reason, the choice of the length of unary encoding is 8. This way, logarithmic unary encoding can represent the smallest value of $1/2^8$.

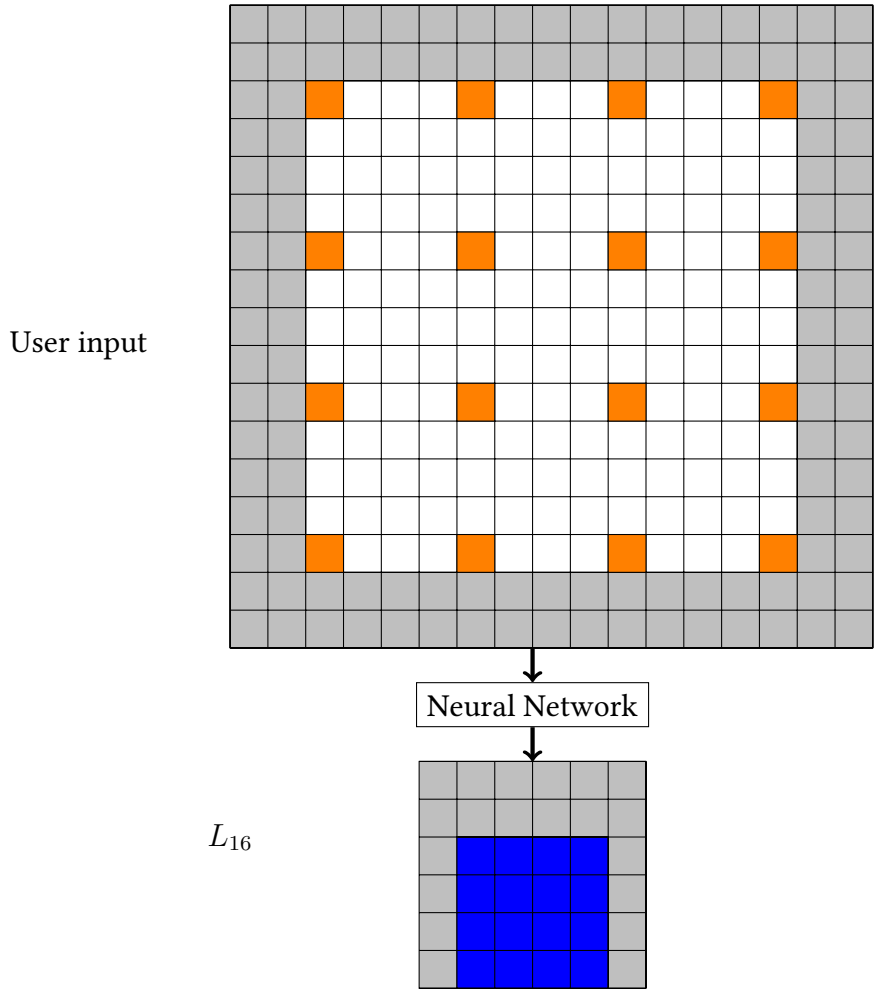

Figure 2.4 This figure shows generation of L_{16}. Values are smaller for demonstration purposes. User input has 'reflection' padding. This is represented by gray colour. L_{16} is the generation matrix. Its edges are initialized by scaled down user input. This is also represented by gray colour. Layer-64x is taken at orange cells at user input. This is every 16th cell in reality. The blue part of L_{16} is generated by using the neural network as shown in Figure 2.3 and is cut out.

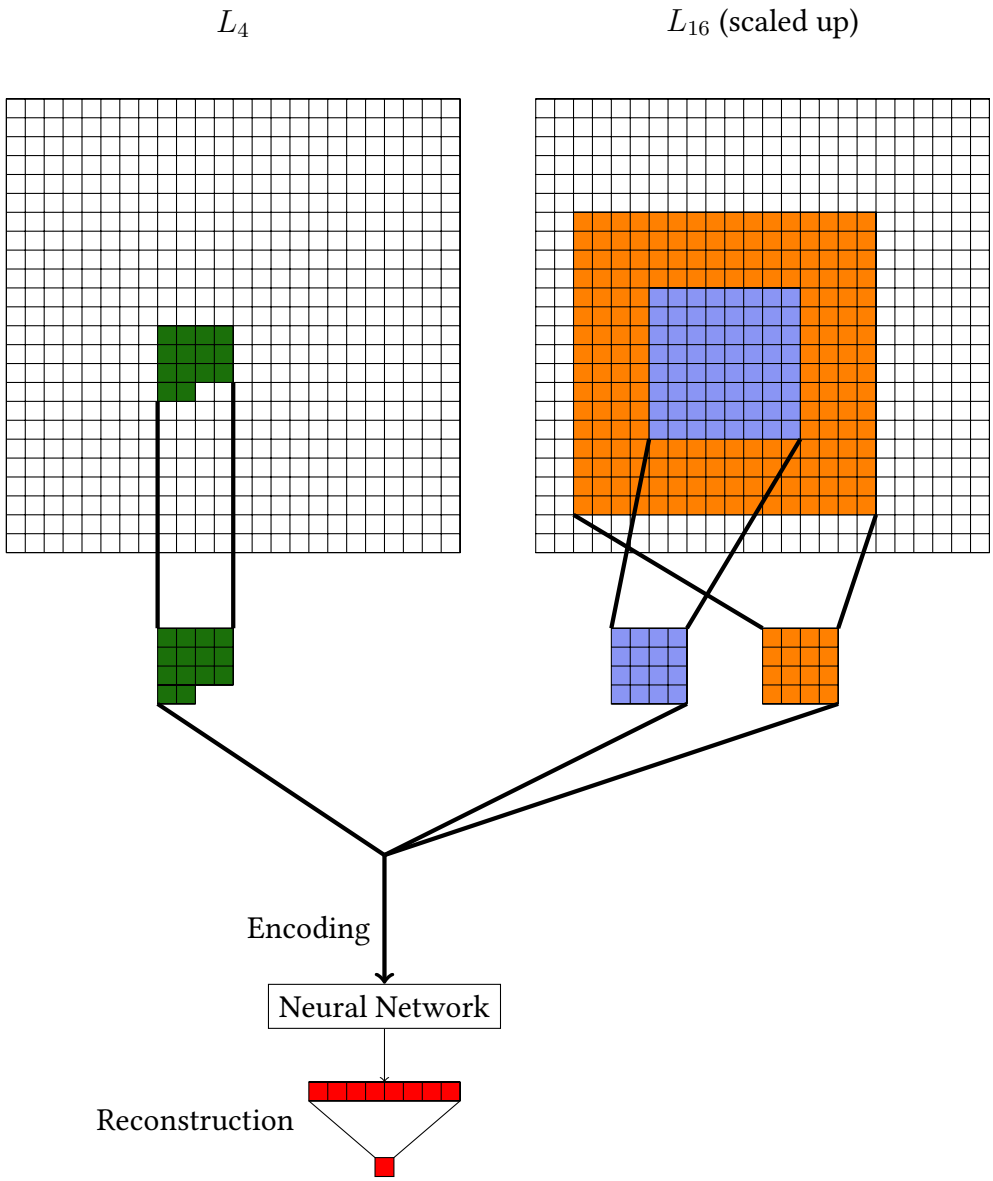

Figure 2.5 This figure shows processing of a single input in generation of L_4 using L_{16} as context. Smaller windows are used for demonstration. The middle position of all windows is the same. In practice, windows are of size 5×5, 20×20 and 80×80. The green window is the recurrent input. The blue window is Layer-16x context. The orange window is Layer-64x context. These 2 windows are scaled down to 5×5, as shown above.

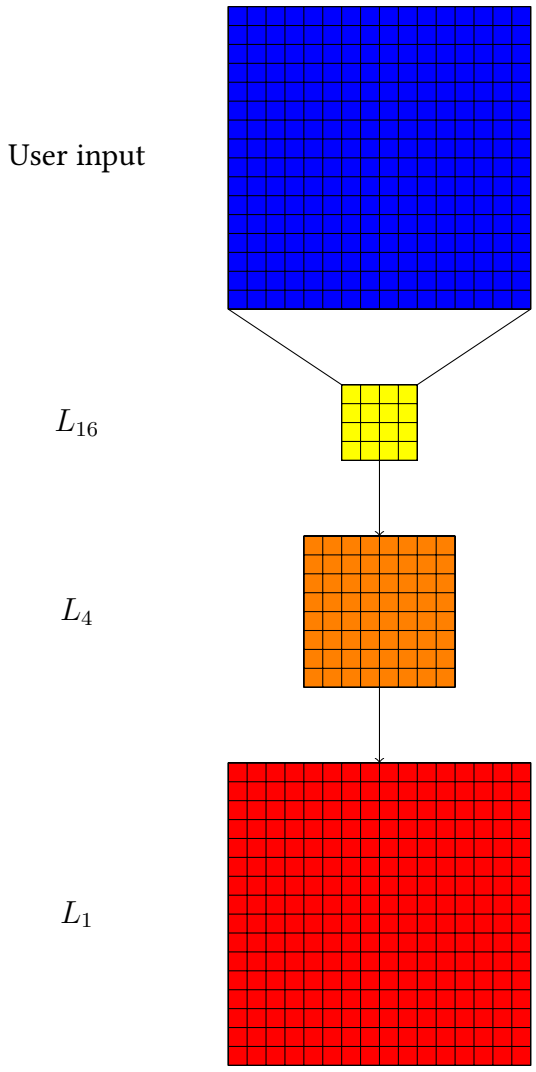

Figure 2.6 This figure shows the idea behind generation of an entire stage. Generating L_{16} from the user input results in loss of data, capturing only high level features. Then the networks expand on previous layer and generate a more detailed version. This results in remixed content filled with details that networks picked up on during training.

Chapter 3

Applying the RNN generation to real world data

3.1 Input datasets

A general area of Czech Republic was selected due to its diversity of terrain, good size and high quantity of map features. Table 4.1 shows its Earth coordinates. In both, OSM and OpenTopography, the data is downloaded using rectangular selection, specifying corners with longitude and latitude coordinates. This allows for easy matching of data from both sites. However, the OpenStreetMap site has limits on the size of the exported area. QGIS is a free and open source geographic information system [1]. QGIS 2.18.28 with OSMDownloader plugin [2] solves this problem by allowing unlimited downloads. This plugin offers option to select area by rectangle. However, it is only compatible with older versions of QGIS.

The first step is to parse the altitude matrix. The data can be downloaded in ESRI ArcASCII format, which is easily parsed and provides a complete matrix of altitudes. The second step is to parse the OSM file. OSM data is in a form of non-discrete points and polygons. This needs to be rasterized. That is, we need to create a matrix of the same size as the altitude matrix and translate the coordinates to discrete form. Finally, we need to simply draw the shapes into the created matrices. Value 0 means absence of a feature and value 1 means presence of that feature. Note that roads are just in the form of lines, while rivers tend to be wider and are often represented by polygons. The specific tags used to filter only needed features are as follows:

Roads: (key=highway, value=motorway|trunk|primary|secondary|tertiary)

[1] `https://www.qgis.org/en/site/`

[2] `https://github.com/lcoandrade/OSMDownloader`

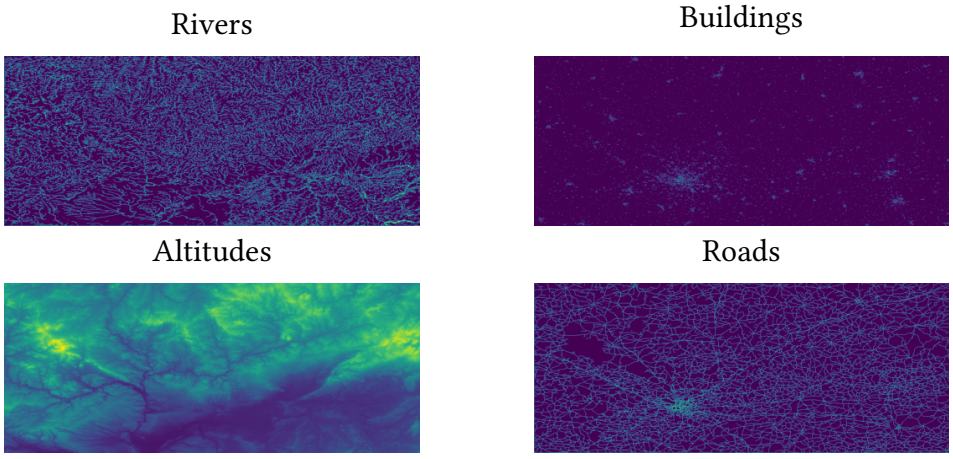

Figure 3.1 PGM data

Rivers: key=waterway, (key=water, value=river)

Buildings: key=building, key=amenity

For simplicity, C# provides XMLReader class for the large OSM file. Using a standard drawing library, polygons and lines can be easily drawn into the matrix, by converting it into an image. The output form of the data needs to be easily parsable and viewable. The PGM grayscale image format serves this purpose, as it is one of the simplest existing image formats.

The output of this algorithm consists of 4 PGM images for each type of data. Altitudes, roads, rivers, and buildings. They can be seen in Figure 3.1. These images are then processed into concrete inputs for the networks.

3.2 Training the Networks

Keras is the most widely used Python deep learning framework. It comes with simple and flexible API. [3] This framework is used to build the networks.

There are 3 networks for every stage. Networks that generate L_{16} from Layer-64x are denoted as: network $(64) \rightarrow 16$. Altitude generation stage requires following networks: $(64) \rightarrow 16$, $(64, 16) \rightarrow 4$ and $(16, 4) \rightarrow 1$. Other features have the last layer split into $(16, 4) \rightarrow blurry$ and $(blurry) \rightarrow sharp$. The blurry layer is simply created by using Gaussian blur with kernel shape of (5×5) on the original image.

The first step to prepare the training data is to cut out windows across the layers. The base size for a Layer-1x window is (5×5). Windows are taken at

[3] `https://keras.io/`

every pixel. Layer-Nx has a window N-times larger, which is then scaled down to (5×5). The result of this are lists of (5×5) windows for every layer. The standard score normalization is used for altitude data. For altitude generation, random noise is also applied to these layers to reduce the chance of overfitting.

As roads and rivers depend on the differences in altitude more than on actual altitudes, 2 more images are generated. Differences of altitudes in x and y direction. Given altitude matrix A, the matrix containing the differences of altitudes in x direction Dx is defined as: $Dx[row, column] = A[row, column] - A[row, column + 1]$. Similarly: $Dy[row, column] = A[row, column] - A[row + 1, column]$.

All of this results in 4 images parsed into layers, as explained in Section 2.4. Altitudes, altitude differences in x and y direction, roads, rivers, and buildings. The next step is to take the layers and create training examples for the networks. This is individual for every stage and every network. However, every training set consists of 150 000 random examples. Roads and rivers are processed in the same way and are addressed as paths. The inputs and outputs for each network are described as follows:

Altitudes $(64) \rightarrow 16$ **:** The input consists of Layer-64x context and recurrent input. Making the heights relative and encoding the inputs makes total size of input 392. The output is a single encoded pixel in 8 neurons. The loss is mean squared error.

Altitudes $(64, 16) \rightarrow 4$ **and** $(16, 4) \rightarrow 1$ **:** Both networks take 2 layers as context. This makes input length 600. The output length is 8. The loss is mean squared error.

Paths $(64) \rightarrow 16$ **:** The context consists of Layer-64x of paths and altitude differences. It also consists of Layer-16x altitude differences. This makes input length 1176. The output length is 8. The loss is mean squared error.

Paths $(64, 16) \rightarrow 4$ **:** The context consists of Layer-64x and Layer-16x of paths and altitude differences. It also consists of Layer-4x of altitude differences. This makes input length 1776. Output is 8. The loss is mean squared error.

Paths $(16, 4) \rightarrow blurry$ **:** The context consists of Layer 16x and Layer-4x of altitude differences. It also consists of Layer-1x of altitude differences. This makes input length 1776. The output length is 8. The loss is mean squared error.

Paths $(blurry) \rightarrow sharp$ **:** Sharpening network takes only blurry layer as context. This makes total length of input 222. The output length is 1. This

network simply tries to classify, wether there is a path or not. The loss is binary crossentropy.

Buildings $(64) \rightarrow 16$ **:** Buildings take roads, rivers and altitudes as context. It consists of Layer-64x of roads, rivers, altitudes and buildings. It also consists of Layer-16x of roads, rivers and altitudes. This makes input length 1592. The output length is 8. The loss is mean squared error.

Buildings $(64, 16) \rightarrow 4$ **:** The context consists of Layer-64x and Layer-16x of roads, rivers, buildings and altitudes. It also consists of Layer-4x of roads, rivers and buildings. This makes total length of input 2400. The output length is 8. The loss is mean squared error.

Buildings $(16, 4) \rightarrow blurry$ **:** The context consists of Layer-16x and Layer-4x of roads, rivers, buildings and altitudes. It also consists of Layer-1x of roads, rivers and buildings. Recurrent input is blurry Layer-1x of buildings. This makes total length of input 2400. The output length is 8. The loss is mean squared error.

Buildings $(blurry) \rightarrow sharp$ **:** This network takes only blurry Layer-1x as context. Input length is 222. The output length is 1. The loss is binary crossentropy.

Chapter 4

Results and discussion

4.1 Generation Results

Due to computational difficulty of training recurrent neural networks, their large quantity and large size of the created datasets, all networks have been trained up to 10 epochs. Even after 10 epochs they already received diminishing improvements of loss. Loss of all the networks was between 0.1 and 0.01.

Networks for generating altitudes were trained on multiple datasets shown in Figure 4.1. All user inputs used in this testing are in the shape of (100×100). Using a neural network trained on Czech Republic dataset, Figure 4.3 shows generation results on a simple hand drawn input of a valley. The output is visibly more complex. Figure 4.2 shows generation results on complex input from a real world map. The map retains some broad features of the original input. Figure 4.4 and Figure 4.5 show impact from training on Sahara dataset. The resulting images are visually close to sand dunes. Figure 4.6 and Figure 4.6 are results based on the Himalayas dataset. This network seems to generate vertical valleys, which share similarities with the Himalayas dataset. Sahara and Himalayas networks seem to generate maps that are very different from the user input.

Generation of other features was more difficult. It can be seen in Figure 4.8 and Figure 4.11 that the sharpening network was successful in creation of lines from the blurry layer. However, the blurry layer is simply too inaccurate to generate longer, more connected roads. Figure 4.9 and Figure 4.12 show very similar results in generation of rivers. As Figure 4.13 and Figure 4.10 show, the attempt at generating buildings was mostly unsuccessful. The Network generating the blurry layer generates only empty images.

Figure 4.15 and Figure 4.14 show more closely how roads and rivers respond to altitudes. In some cases, they follow edges of elevated levels.

Figure 4.16 is a result without any randomness. This picture resembles the

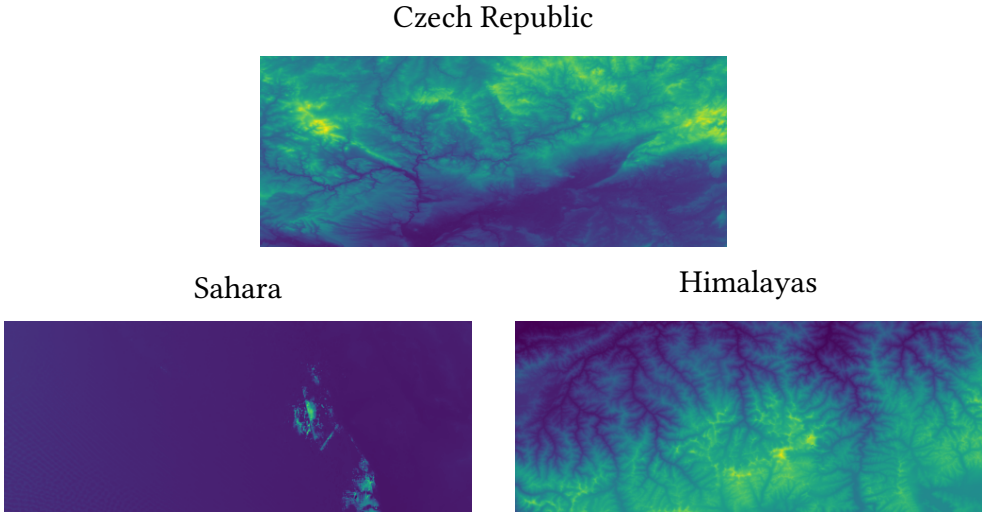

Figure 4.1 Training datasets of multiple regions. Their locations on Earth can be seen in Table 4.1.

Area	Left longitude	Right longitude	Bottom latitude	Top latitude
Czech Republic	13.500000000	16.165283203	49.28252211106	50.3701685954
Sahara	12.3310546875	14.9968893975	23.769305315898	24.8575011166
Himalayas	85.4110107421	88.0784912109	27.225447545489	28.3130940298

Table 4.1 This table shows Earth's coordinates of training datasets.

original valley. Compared to Figure 4.15, it shows impact of randomness. This picture has far less features than the original input. It is, however, more detailed. The problem is that the details may be visually pleasing, but are not result of learning from real world examples. Therefore, a lot of tweaking is needed to find the right amount of randomness. On the other hand, comparing Figure 4.18 and Figure 4.17, neither of them resemble the original input. This happens despite the fact that Figure 4.17 was generated without any randomness.

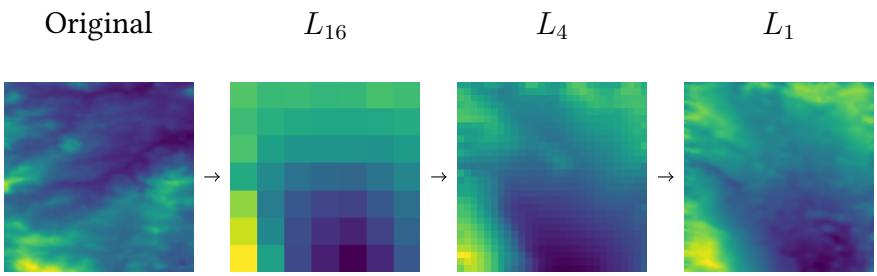

Figure 4.2 Generation of altitudes from complex input with Czech networks.

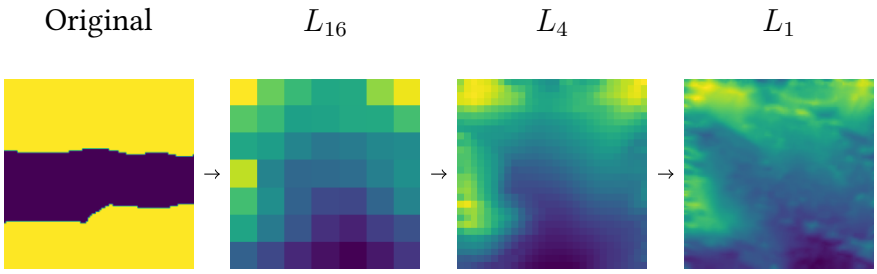

Figure 4.3 Generation of altitudes from simple drawn input with Czech networks.

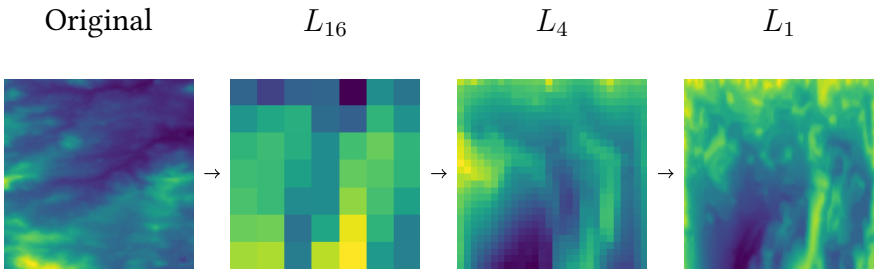

Figure 4.4 Generation of altitudes from complex input with Sahara networks.

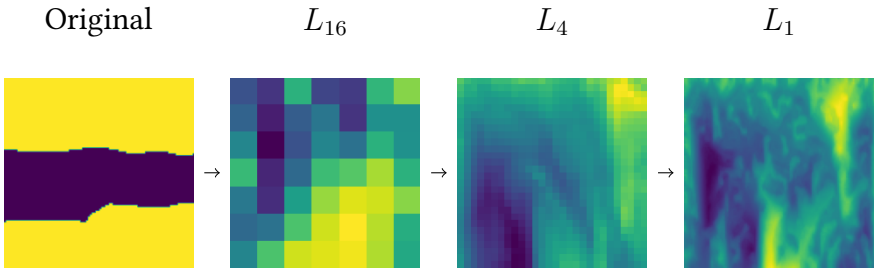

Figure 4.5 Generation of altitudes from simple input with Sahara networks.

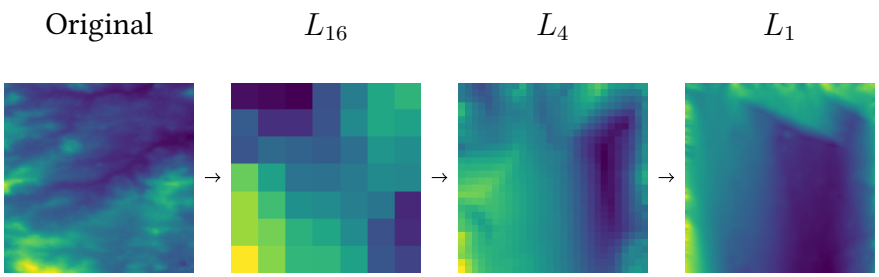

Figure 4.6 Generation of altitudes from complex input with Himalayas networks.

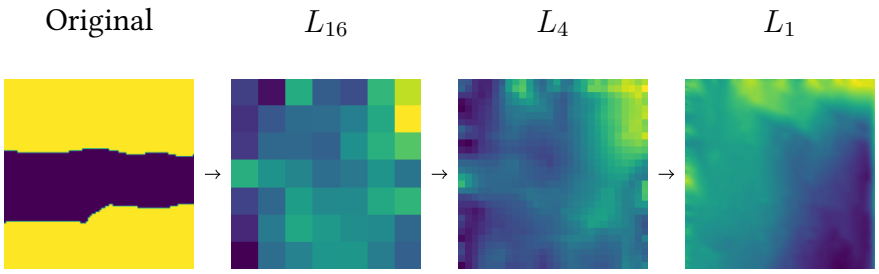

Figure 4.7 Generation of altitudes from simple input with Himalayas networks.

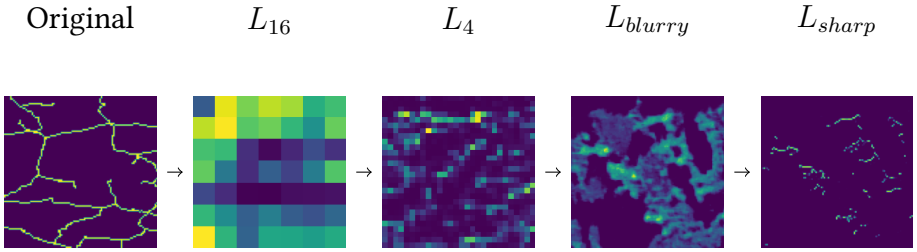

Figure 4.8 Generation of roads based on altitudes in Figure 4.2.

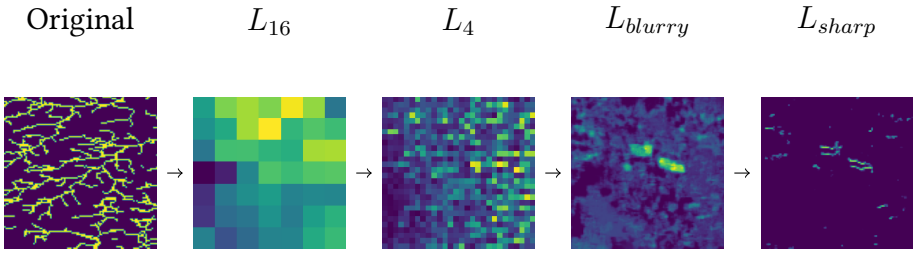

Figure 4.9 Generation of rivers based on altitudes in Figure 4.2.

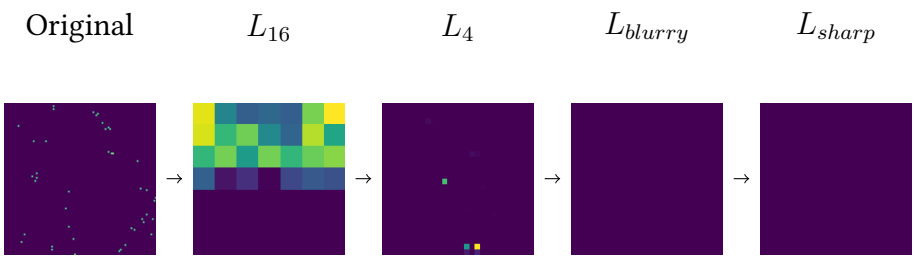

Figure 4.10 Generation of buildings based on Figure 4.8, Figure 4.9 and Figure 4.2.

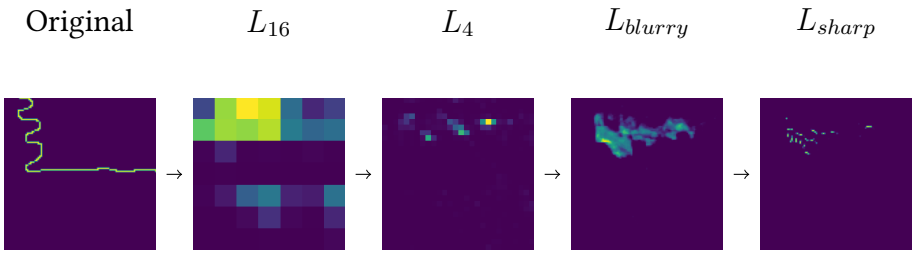

Figure 4.11 Generation of roads based on altitudes in Figure 4.3.

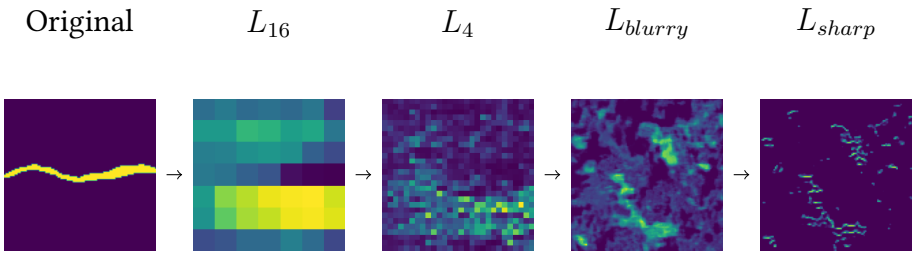

Figure 4.12 Generation of rivers based on altitudes in Figure 4.3.

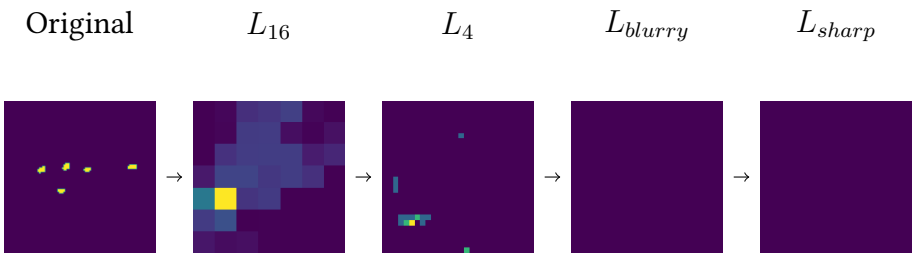

Figure 4.13 Generation of buildings based on Figure 4.11, Figure 4.12 and Figure 4.3.

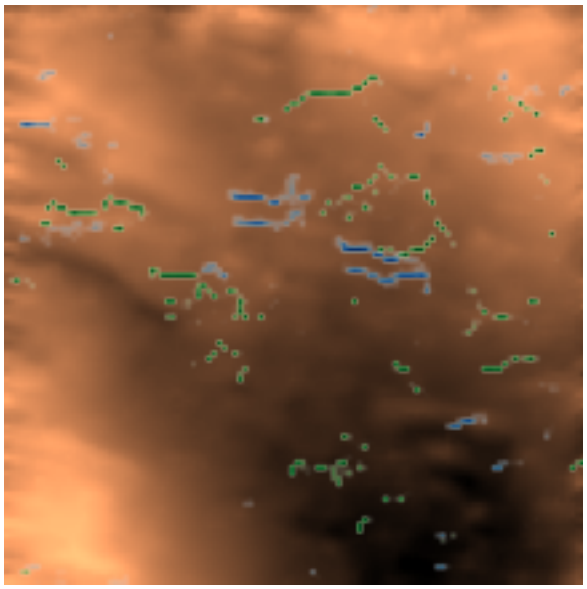

Figure 4.14 Map of all features generated from complex input by Czech networks. The blue colour represents locations of rivers. The green colour represents locations of roads.

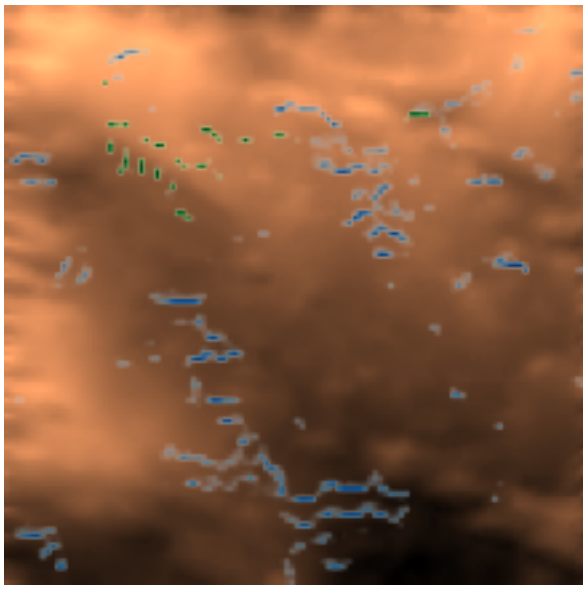

Figure 4.15 Map of all features generated from simple drawn input by Czech networks. The blue colour represents locations of rivers. The green colour represents locations of roads.

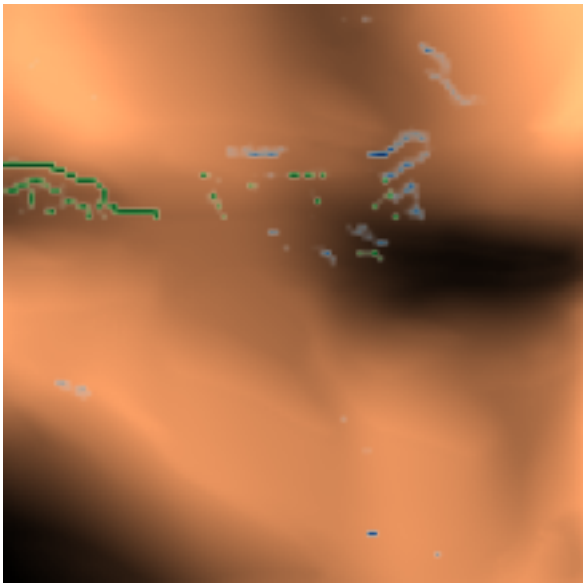

Figure 4.16 Map of all features generated from simple drawn input by Czech networks without any randomness. The blue colour represents locations of rivers. The green colour represents locations of roads.

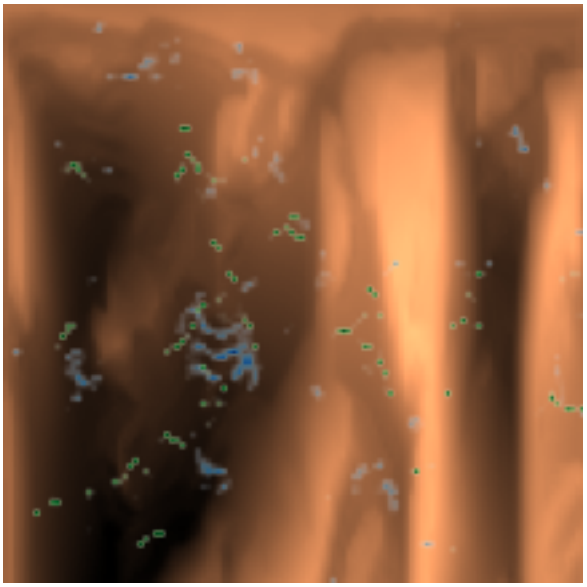

Figure 4.17 Map of all features generated from the complex input by Sahara networks without any randomness. The blue colour represents locations of rivers. The green colour represents locations of roads.

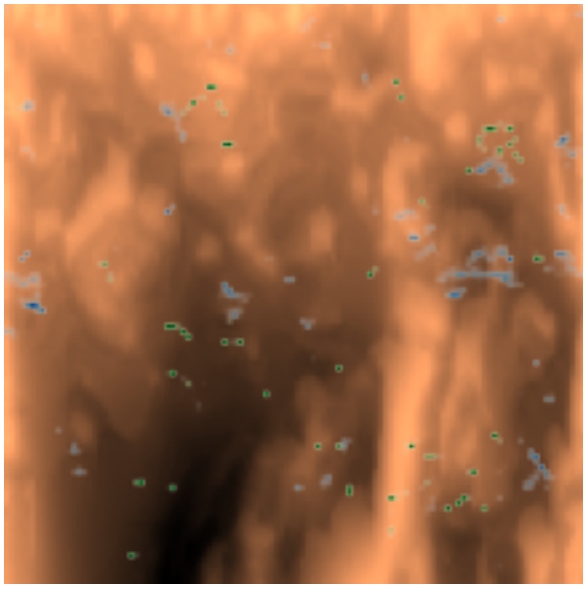

Figure 4.18 Map of all features generated from the complex input by Sahara networks. The blue colour represents locations of rivers. The green colour represents locations of roads.

4.2 Discussion

The altitude networks show applicable results. They proved capable of producing realistically looking images that are resembling the user input. The generated images show influence of training datasets. However, these images still lack fine details in comparison to training datasets. This is solved by random noise, but at the cost of user control. Maybe given much larger input image, less data would have been lost in the process of generation.

Roads and rivers were successful in generation of some path resembling images. They also show ability to respond to altitudes. However, the lines are too short and inconsistent. Improving the generation of the blurry layer could have the most impact on better generation. Maybe reducing the kernel size of Gaussian blur might help in future improvements.

Buildings networks are the most computationally demanding networks because of their large input. This makes them very time consuming to train. The network generating blurry layer seems to lose almost all of the data and generates empty images. Except for improving this network, processing the previous layer might give this network enough information. For example, rescaling values in L_4, so it contains higher probabilities.

In all cases, future work would include trying larger networks with more training. Altitude networks in particular could benefit from different unary re-

construction method to encourage more detailed, fractal generation.

Conclusion

In this thesis, OSM data together with OpenTopography were analyzed and successfully converted to a viable form for RNNs. An algorithm for generation of random maps using pixel RNNs was successfully constructed and implemented. The networks have shown the ability to create altitude maps, road maps, and river maps.

Altitude images generated by the algorithm look realistically and are not far from actual usable maps. The generation is controllable by simple user input and choice of the training dataset. Neither road nor river images generated by the algorithm are of applicable quality, but they could become useful upon further improvements to the algorithm. The building images have not been successfully generated.

Trained RNNs show the ability to generate random maps. The real world samples show influence on creating realistically looking maps. This is especially true for altitude maps. With the basic algorithm provided, it is possible to iterate on this and create better results by improving the neural networks.

For the future work, there are multiple areas to make the algorithm better:

- Testing the networks on larger inputs.
- Improving neural networks.
- Tweaking the noise added in training and generation phase to better reflect user input.
- Choice of different data encoding and reconstruction.
- Addition of more map features.

Bibliography

[1] Travis Archer. "Procedurally generating terrain". In: *44th annual midwest instruction and computing symposium, Duluth*. 2011, pp. 378–393.

[2] William Chan et al. *Listen, Attend and Spell*. 2015. arXiv: 1508.01211 [cs.CL].

[3] George Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314.

[4] Lawrence Johnson, Georgios N Yannakakis and Julian Togelius. "Cellular automata for real-time generation of infinite cave levels". In: *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. 2010, pp. 1–4.

[5] Subhash Kak. *Unary Coding for Neural Network Learning*. 2010. arXiv: 1009.4495 [cs.NE].

[6] Andrej Karpathy. *The Unreasonable Effectiveness of Recurrent Neural Networks*. https://karpathy.github.io/2015/05/21/rnn-effectiveness. 2015.

[7] Stefan Knerr, Léon Personnaz and Gérard Dreyfus. "Handwritten digit recognition by neural networks with single-layer training". In: *IEEE Transactions on neural networks* 3.6 (1992), pp. 962–968.

[8] Jacob Olsen. "Realtime procedural terrain generation". In: (2004).

[9] Teong Joo Ong et al. "Terrain generation using genetic algorithms". In: *Proceedings of the 7th annual conference on Genetic and evolutionary computation*. 2005, pp. 1463–1470.

[10] Raúl Rojas. *Neural networks: a systematic introduction*. Springer Science & Business Media, 2013.

[11] Dominik Scholz. "Tile-Based Procedural Terrain Generation". PhD thesis. Technische Universität Wien, 2019.

[12] Ilya Sutskever. *Training recurrent neural networks*. University of Toronto Toronto, Canada, 2013.

[13] Frank M Thiesing and Oliver Vornberger. "Sales forecasting using neural networks". In: *Proceedings of International Conference on Neural Networks (ICNN'97)*. Vol. 4. IEEE. 1997, pp. 2125–2128.

[14] Aaron Van Oord, Nal Kalchbrenner and Koray Kavukcuoglu. "Pixel recurrent neural networks". In: *International Conference on Machine Learning*. PMLR. 2016, pp. 1747–1756.

[15] Zarita Zainuddin and Ong Pauline. "Function approximation using artificial neural networks". In: *WSEAS Transactions on Mathematics* 7.6 (2008), pp. 333–338.

Appendix A

Using the software

The following dependencies are needed for the algorithm used in this thesis. These are the steps to get them on a windows based system:

- **.NET Core 3.0 :** Download with Visual Studio: `https://visualstudio.microsoft.com/downloads/`
- **System.Drawing.Common :** Download in Visual Studio in OSM_Parser project under NuGet Packages function.
- **Python 3+ :** Download from `https://www.python.org/downloads/`
- **Tensorflow 2.2.0 :** Use `python -m pip install tensorflow=2.2.0` in the command line or navigate to `https://www.tensorflow.org/install`). It needs to be this specific version.
- **NVCuda 10.1+ :** Follow the guide: `https://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/index.html`
- **cuDNN 7.6 :** Follow the guide: `https://docs.nvidia.com/deeplearning/cudnn/install-guide/index.html` It needs to be this specific version.
- **openCV :** `python -m pip install opencv-python`
- **numpy :** `python -m pip install numpy`
- **pyplot :** `python -m pip install matplotlib`
- **Jupyter Notebook :** `python -m pip install jupyterlab`

- **QGIS 2.18.28 (optional) :** Navigate to `https://qgis.org/downloads/` and download the correct version. Then through "manage plugins" feature, download OSMDownloader plugin.

The complete steps for the whole process of downloading and parsing the data, training the networks and generating the maps are described as follows:

1. Go to `https://portal.opentopography.org/datasets` and select the region you want to export by hand.

2. Under "Global Data" select Global Multi-Resolution Topography (GMRT) Data Synthesis option.

3. You can adjust the longitude and latitude values precisely at this point.

4. Select ESRI ArcASCII format and hight resolution. Download the data and extract them.

5. Navigate to `https://www.openstreetmap.org/` and use the export feature to download the same coordinates as before. If your area is too large, open QGIS and use OSMDownloader instead.

6. Open OSM_Parser/OSM_Parser.sln in visual studio.

7. Install the System.Drawing.Common library via NuGet packages function.

8. Build the solution in release mode.

9. Copy your osm and asc files into the release directory.

10. Parse the files using the program: `./OSM_Parser.exe file.asc file.osm`. This will result in 4 images: `heights.pgm`, `roads.pgm`, `rivers.pgm`, and `buildings.pgm`. If osm file is not specified, only heights file will be created.

11. Place the files in a folder of your choosing and navigate to "Neural Networks" folder in the command line.

12. Execute `jupyter notebook` in the command line and open "Example.ipynb" in the browser window that is now opened. This file contains rest of the instructions on how to parse the data, train the networks and generate new maps.

The selected area has to be quite large for the parsing to be successful. (720×720) is the bare minimum for the altitude (ArcASCII) map. The number of viable training examples will be less from such map as empty examples are omitted for roads, rivers, and buildings. Area of Czech Republic, as specified in Table 4.1, is recommended.

To summarize the python scripts, "nn_generator" folder contains 5 main scripts. Each of them contains a single function that does its functionality.

- **parse.py** parses the input image into layers. It is used to parse training images into Layers 64x, 16x, 4x and 1x. It also computes blurry layer and layers of altitude differences.
- **normalize_heights.py** uses standard score normalization on the height layers.
- **process_training_data.py** takes the given layers and processes them into training data for a single network.
- **train.py** is parametrized with a name of a file that specifies network structure. It builds the network based on this file, trains it based on given parameters and saves it into given location.
- **generate.py** is provided with configuration file "generation_config.txt", which specifies which networks are used for the generation. This script is given a name of a folder containing user input and generates the maps into the specified output folder.
- **libraries** folder contains libraries with helpful functions.
- **model_structures** folder contains files describing structures of each network.
- **generation_data** folder contains example files for the generation.
- **Example.ipynb** is, as mentioned above, example python notebook file containing script on how to perform the algorithm from parsing to generation.

It is important to note that these computations are very demanding and time consuming. Especially data used for training the network for generation of buildings. This network takes data from all the previous layers as an input. The software comes with prepared unparsed PGM images in **raw_data** folder. It also comes with pre-trained networks in **models** folder. This is for demonstration purposes.

The source code can be found on the github: `https://github.com/Ermith/Thesis-Public.git`