Eberhard Karls University of Tübingen
Wilhelm-Schickard-Institute for Computer Science

Bachelor's Thesis Computer Science

Learning Dynamics in Models for Topographic Evolution

Author

Michel Jubke

Examiner

Prof. Phillip Hennig
Wilhelm-Schickard-Institute for Computer Science
University of Tübingen

Supervisor

Jonathan Schmidt
Maria-von-Linden-Straße 6, 2.OG, 20-30/A15
University of Tübingen

Jubke, Michel
Learning Dynamics in Models for Topographic Evolution
Bachelor's Thesis Computer Science
Eberhard Karls University of Tübingen
Time allowed for completion: 17.5.2022 - 9.9.2022

Abstract

Erosion shapes landscapes and impacts the lives of people all around the globe. We present a semi-automatic method that uses Landlab — a toolkit to numerically model earth surface dynamics — to generate data that describes water related erosion in mountainous areas. This data is used to train different *graph neural networks* (GNN) in order to examine, if they are capable of generating correct erosion simulations from unseen initial conditions. We succeed in finding GNN architectures that solve this task on 900-, 2500-, and 6400-node voronoi grids where the resolution of the model simulations decreases as the number of grid nodes increases. Further, two directions are pointed to firstly refine the results of this study and to secondly scale the proposed method up to potentially huge grids.

Zusammenfassung

Erosion formt Landschaften und beinflusst das Leben von Menschen auf der ganzen Erde. Wir präsentieren eine halb-automatisceh Methode, um Daten zu generieren, die wasserbedingte Erosion in gebirgigem Gelände beschreiben. Diese Methode nutzt Landlab, ein Toolkit um numeriscche Modelle der Erdoberflächendynamik zu erstellen. Unsere Daten nutzen wir, um verschiede *Graph Neural Networks* (GNN) zu trainieren, mit dem Ziel heraus zu finden, ob diese auch aus unbekannten Anfangsbedingungen korrekte Erosions-Simulationen erzeugen können. Erfolgreich präsentieren wir Netzwerk-Architekturen, die diese Aufgabe auf 900-, 2500- und 6400-Knoten Voronoi-Grids lösen. Allerdings nimmt die Auflösung der Simulationen ab, wenn die Anzahl der Knoten steigt. Deshalb zeigen wir zwei Richtngen auf, um erstens die Ergebnisse dieser Untersuchnug zu verbessern und um zweitens die vorgestellte Methode auf potentiell riesige Grids zu skalieren.

Acknowledgements

First, I want to thank Professor Phillip Hennig and my supervisor Jonathan Schmidt for accompanying this study over the last months and for giving me advise and fortification whenever I needed it.
Further, I want to thank Franziska Weiler for always having the right answers when it came to the organizational part of this project.
Also I want to thank Yannick Streicher for his kind and valuable words in times of despondence.

But most importantly I want to thank Caya, Ida and Imi for always having my back, giving me love and motivation and cheering me up after a hard day of work. Without you and the rest of the family, none of this would have been possible.

Contents

List of Figures vii

List of Tables ix

Abbreviations xi

1 Introduction **1**

1.1 Related Work . . . 3

1.1.1 Numerical Approaches . . . 4

1.1.2 Data Driven Approaches . . . 4

1.2 Goal . . . 5

1.3 Structure . . . 5

2 Theoretical Background **7**

2.1 Landlab . . . 7

2.1.1 Landlab Grids . . . 8

2.1.2 Landlab Components . . . 10

2.2 Graph Neural Networks . . . 13

2.2.1 Graph Embedding . . . 13

2.2.2 Graph Convolution . . . 14

2.2.3 Message Passing . . . 16

2.2.4 GCNConv . . . 17

2.2.5 SAGEConv . . . 19

3 Method **23**

3.1 Data . . . 23

3.1.1 Grid Type and Size . . . 23

3.1.2 Initial Topographies . . . 24

3.1.3 Erosion Timeseries . . . 26

3.1.4 Encoding . . . 26

3.2 Models . . . 26

3.2.1 Goal . . . 27

3.2.2 32-layer Architecture . . . 27

3.2.3 48-layer Architecture . . . 27

3.2.4 80-layer Architecture . . . 28

3.3 Training . . . 28

4 Results **31**

4.1 Questions . . . 31

4.2 Evaluation Metrics . . . 31

4.3 900-Node Grids . . . 32

4.3.1 GCN32 on Raster900 . . . 32

4.3.2 SAGE32 on Raster900 . . . 33

4.3.3 GCN32 on Voronoi900 . . . 34

4.3.4 SAGE32 on Voronoi900 . . . 35

4.3.5 Evaluation . . . 36

4.4 2500-Node grids . . . 37

4.4.1 SAGE32 on Voronoi2500 . . . 37

4.4.2 SAGE48 on Voronoi2500 . . . 38

4.4.3 Evaluation . . . 38

4.5 6400-Node grids . . . 39

4.5.1 SAGE48 on Voronoi6400 . . . 39

4.5.2 SAGE80 on Voronoi6400 . . . 40

4.5.3 Evaluation . . . 40

4.6 Further Remarks 41

5 Conclusion **43**

5.1 Discussion 43

5.2 Outlook 44

Bibliography **45**

List of Figures

8

2.2 Example of a graph with scalar node features 15

3.1 Example initial topographies 25

4.1 GCN32 on 900-node raster grid 33
4.2 SAGE32 on 900-node raster grid 34
4.3 GCN32 on 900-node voronoi grid 35
4.4 SAGE32 on 900-node voronoi grid 36
4.5 SAGE32 on 2500-node voronoi grid 37
4.6 SAGE48 on 2500-node voronoi grid 38
4.7 SAGE48 on 6400-node voronoi grid 40
4.8 Exemplary losses of models that do not train 41

List of Tables

4.1 Losses and required training time of GCN32 on Raster900 . . . 32
4.2 Losses and required training time of SAGE32 on Raster900 . . . 33
4.3 Losses and required training time of GCN32 on Voronoi900 . . . 34
4.4 Losses and required training time of SAGE32 on Voronoi900 . . 35
4.5 Simulation losses on 900-node grids 36
4.6 Losses and required training time of SAGE32 on Voronoi2500 . 37
4.7 Losses and required training time of SAGE48 on Voronoi2500 . 38
4.8 Simulation losses on 2500-node grids 39
4.9 Losses and required training time of SAGE48 on Voronoi6400 . 39

Abbreviations

SPL	Stream Power Law
SPE	Stream Power Equation
PyG	PyTorch Geometric
PDE	Partial Differential Equation
CNN	Convolutional Neural Network
GNN	Graph Neural Network
VL	Validation Loss
MSL	Mean Simulation Loss

Chapter 1

Introduction

The evolution of the Earth's surface topography is an omnipresent and never ending process. Besides phenomena like landslides, volcanic eruptions, earthquakes or topographic uplift due to tectonic activity, it is *erosion* that shapes landscapes all across the globe and thus impacts the lives of humans in a subtle yet crucial manner.
It is the discipline of geomorphology that tries to firstly understand such erosion processes and to secondly make erosion related predictions both forward and backward in time. While such backward predictions of earth surface dynamics might mainly satisfy the general human curiosity in understanding the physical laws governing our planet, the forward predictions are of much bigger practical use: just think of infrastructure projects or urban development which — in the best case — should be inter-coordinated with the evolution of the surrounding landscape on a long term time scale.

Erosion can be described as *sediment transportation and deposition* (Braun and Willett 2013) and it is a whole range of phenomena that can be made responsible for this processes.
However, the impact of the single actors varies and depends a lot on the specific setting. In dry and rather flat areas like deserts for example, aeolian processes (i.e., wind) play the biggest role, in other instances it could be hillslope processes like soil creep that mainly cause erosion (Braun and Willett 2013).
In this study, we will examine erosion in *mountainous* landscapes that are characterized by relatively large mean slopes. In such environments, it is water coming from rain or glaciers that is responsible for the transportation of sediment.
The water gathers in channels that incise into the bedrock with time. This channel incision destabilizes neighbouring hillsides and results in a gravity driven movement of soil and rock towards the bottom of the channels. From there, the sediment is then transported towards lower elevations by the water

flowing in the channels (Braun and Willett 2013; Whipple and G. E. Tucker 1999; Whipple 2004).

In order to build reliable erosion prediction models for such environments, it is thus a crucial step to first understand the water runoff behaviour on surfaces. This task is commonly known as *flow routing*.
Once we are able to precisely quantify where which amounts of water are flowing on a given landscape (e.g. after rainfall), we can make use of physical theories to calculate the amounts of sediment being transported downhill.

Landscape evolution processes usually take place in very large time scales: we are talking about at least tens or hundreds but often thousands or even millions of years (G. Tucker 2015). Therefor it is only to a certain degree, that theories describing the laws governing erosion can be empirically substantiated. However, one theory that is broadly accepted as governing water related erosion processes is the *stream power law* (SPL) (Yuan et al. 2019). The SPL assumes, that the rate of channel incision is proportional to the hydraulic shear stress that a river is exerting to its channel bed (Braun and Willett 2013). In its simplest form the SPL can be expressed as

$$\frac{\partial h}{\partial t} = -KA^m S^n, \tag{1.1}$$

which is also known as the *stream power equation* (SPE). In this equation h is topographic elevation, t is time, K is the fluvial erosion coefficient — a constant depending on a range of properties of the river bed (Howard and Kerby 1983; Whipple and G. E. Tucker 1999) — A is the contributing upstream drainage area, S is the local channel slope and m, n are the SPL exponents (Yuan et al. 2019).
Even though it is well known, that the SPL might be oversimplified because it does not take into account several important processes acting in river channels (Lague 2014), it is the de facto standard for modeling fluvial incision related erosion (Yuan et al. 2019).
The SPE is a *hyperbolic partial differential equation* (PDE). To model stream power governed erosion, it has to be solved in each time step. To solve this kind of equations, numerical approximations are most often applied.

So bringing together a solid understanding of flow routing and both effective and stable methods to solve the stream power equation lets us already build simple numerical models for erosion prediction — at least from a theoretical point of view.

One state of the art toolkit for building erosion models as described so far, is the Python library called Landlab (Barnhart et al. 2020, Hobley et al. 2017). Landlab provides (i) suitable graph encoded data structures to represent landscapes in a efficient and discretized manner and (ii) so called components to simulate — among many other processes — both flow routing and stream power governed erosion.
On the other hand there is an extension to the widespread Python deep learning library PyTorch (Paszke et al. 2019) with the name *PyTorch Geometric* (PyG) (Fey and Lenssen 2019). It extends the well known PyTorch functionalities and allows for *geometric deep learning* (Bronstein et al. 2017), meaning: Convolutional neural network (CNN) like deep learning on graphs and point clouds.

Landlab is the starting point for this study. We use Landlab to generate data that describes flow routing based, stream power governed erosion. This data forms the ground truth for a series of experiments: We want to use it to train different *graph neural networks* (GNNs) (Scarselli et al. 2009) in order to learn erosion dynamics in a data driven end-to-end manner. The predictive performance of the trained GNNs can then be tested against Landlab data again.

Two problems of this attempt can be addressed straight ahead: (i) We do not use real world data in order to learn a real world process. This is a benefit in the way that we have nearly unlimited amounts of data at our disposal but it also puts a lot of trust in the reliability and correctness of the existing methods of which we actually know, that they are based on insufficient assumptions (Lague 2014) (ii) The nature of this thesis is very explorative: From the starting point of view it is neither clear if and if yes, how it is possible to achieve satisfying results.

1.1 Related Work

On the one hand, there are the (classical) numerical approaches for modeling dynamic earth surface processes as they are implemented in Landlab for example.
On the other hand, there are data driven (i.e., machine learning) approaches to model a variety of geomorphological phenomena including soil erosion (Yavari, Maroufpoor, and Shiri 2017). However, we found only one contribution from recent years, that approaches the task of landscape deformation prediction from a graph based (i.e., grid based) perspective (Zhou, Li, and Zhang 2021).

1.1.1 Numerical Approaches

The numerical approaches provide general purpose landscape evolution models with a great flexibility and user friendliness. They mainly differ in (*i*) the way they solve the SPE (equation 1.1) and (*ii*) which factors are included in the computation of the SPE's fluvial erosion coefficient K.
The approach that is implemented in Landlab follows (Braun and Willett 2013). To effectively approximate a solution to the SPE, the authors first build a temporal data structure that indicates, in which order the nodes of the underlying grid have to be visited during computation. The exploitation of this node ordering results in an $\mathcal{O}(n)$ algorithm where n is the number of grid nodes that are used to discretize the landscape. The algorithm is *stable*, *implicit* and *parallel* and can effectively handle very large grids with up to 10^8 nodes.
As mentioned above, some authors criticize the SPL as being over simplified. (Davy and Lague 2009) address this issue and present an equation that explicitly takes into account a mass balance equation for the streamflow (i.e., sediment erosion and deposition) but they do not provide a effective method to solve the equation.
(Yuan et al. 2019) follow the approach taken in (Davy and Lague 2009) and can provide a efficient (i.e., implicit and stable) $\mathcal{O}(n)$ algorithm to solve the equation.

1.1.2 Data Driven Approaches

In (Zhou, Li, and Zhang 2021) the authors use a GNN to predict landscape deformation (i.e., hillslides) in a very specific scenario: They use InSAR (i.e., satellite based) technology to collect landscape data on two sides of a large-scale hydropower dam for a time period of about one year. In this time period several minor landslides took place in the surveyed area. The InSAR data has the form of 3D point clouds which are first transformed into nearest neighbour graphs and then used to train a slope aware GNN in order to predict future landslides.
The authors consider themselves as being one of the first team ever to present a graph-based deep learning approach to modeling landscape evolution.

1.2 Goal

As we have seen, the only graph-based deep learning approach in the field of landscape evolution models deals with a very specific use case and relies on real world data that has to be collected first in a lavish an expensive procedure.

In the current study, we mitigate this problem by using Landlab as inexhaustible source of graph-based erosion data. This gives us the possibility to build more general erosion prediction models that we can test on unseen data rather then performing prediction s within the training domain as it is done in (Zhou, Li, and Zhang 2021).

Still, this study should not be regarded as the attempt to build a fully working general purpose deep learning framework for flow routing based erosion prediction. It is rather an attempt to combine the two worlds of existing numerical methods on one side and existing deep learning techniques that suite this domain on the other hand in order to get a intuition, if this is a promising direction at all.

1.3 Structure

The further structure of this thesis is as follows: Chapter 2 gives a introduction to the Landlab functionalities that are used in this thesis as well as a theoretical background for the different PyG convolution layers that come into action. In chapter 3 we present our method including data generation with Landlab. The results alongside their evaluation follow in chapter 4. The thesis is finally concluded by a discussion and a short outlook in chapter 5.

Chapter 2

Theoretical Background

Before we present the method proposed in this thesis, we introduce all the Landlab functionalities that come into action as well as the theoretical background for the two PyG convolution layers that are being employed later on.

2.1 Landlab

When building landscape evolution models of any flavour, a crucial yet error-prone part is the development of data structures (i.e., *grids*) that hold the rich and mostly continuous landscape related information at ones disposal in a consistent and discrete manner. Especially when irregular grids are applied, as it is the case in a lot of real world applications, this can be very time consuming (Hobley et al. 2017).

Landlab is designed to build numerical models for a wide range of earth surface dynamics without having to deal with these kind of issues. It lets scientists and students start their work exactly where they normally want to start: With setting up or loading a topographic profile and then simulating all kinds of geoscientific processes on it.
In a nutshell, Landlab consists of (i) a powerful gridding engine that defines the model domain and (ii) a wide range of so called *components*, each of which simulating one specific geoscientific process. All the components share a standardized and simple interface.
Landlab follows a very modular plug-and-play style design: The different grid elements (i.e., nodes, links, cells, ...) can hold arbitrary numerical information, dependent of the specific task. On such a loaded grid, we can simulate different earth surface processes by repeatedly running one or multiple of the components. In each iteration, the components update the numerical information stored on the grid (Hobley et al. 2017).

2.1.1 Landlab Grids

As already introduced, a grid forms the basis for all Landlab models. Landlab supports different grid structures. The ones that are used in this study are regular *raster grids* and irregular *delaunay triangulated voronoi cell grids.*

Landlab grids can be divided into different elements. Each of the grid elements can hold an arbitrary number of numerical data fields to represent any kind of landscape related data.
The elements of a Landlab grid are *nodes*, *links*, *cells* and *faces* as well as the less frequently used *corners*, *patches* and *junctions.*

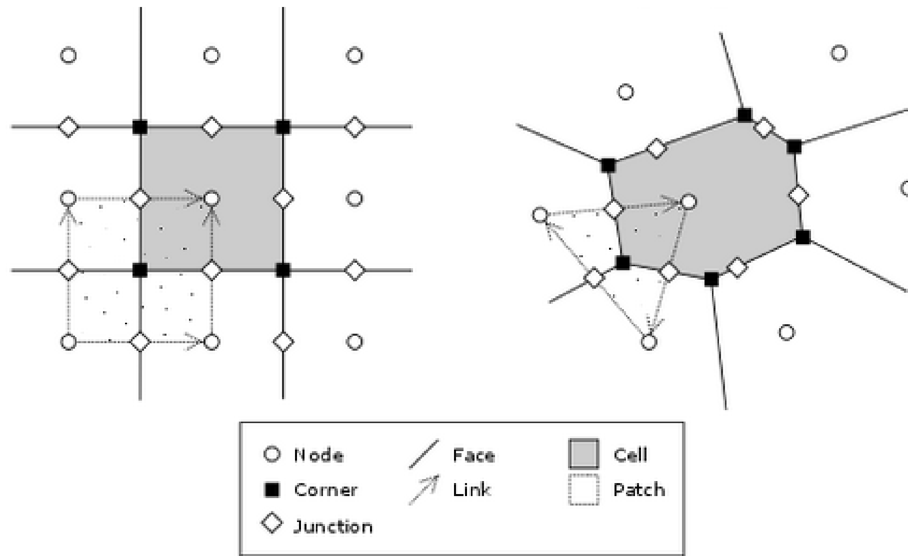

Figure 2.1: Landlab grid elements for regular and irregular grids [1]

Nodes can be further distinguished into *core* nodes and *boundary* nodes depending on their position within the grid. Links can be further distinguished into *active* links and *inactive* links, depending on their capability of transporting flux (i.e., soil, water, ...) (Barnhart et al. 2020). In this study only nodes and links are being used.

Internally, Landlab grids are represented as *dual graphs.* Both of these graphs consist of a set of vertices connected with edges and both are *planar*, meaning edges within one graph do not cross. For the first graph, the vertices are the grid nodes each of which is centered at a grid cell and the edges are the grid links that outline the grid patches. For the second graph, the vertices

[1] This figure is taken from https://landlab.readthedocs.io/en/master/user_guide/grid.html

are the grid corners and the edges are the grid faces that outline the grid cells. The points where edges of both graphs cross define the grid junctions (Barnhart et al. 2020; Hobley et al. 2017).

This simple and yet expressive graph encoded model domain already gives a small glance on the possibilities of applying geometric deep learning methods in this field.

Regular Raster Grid

Raster grids are the simplest grids provided by Landlab. They consist of n rows and m columns of nodes resulting in $m \cdot n$ nodes in total. The spacing between the nodes can be freely chosen both in x and in y direction.
Indeed, this grid structure is so simple, it does not enforce the use of geometric deep learning methods (i.e., GNNs). If we want to operate on grid nodes only, we could easily apply regular CNNs in order to process raster grids. However, as soon as we want to include edge weights (e.g., topographic slopes) into our training process, CNNs reach their limitations.

Irregular Voronoi-Delaunay Grid

A bit more complex are the Voronoi-Delaunay grids. They consist of n nodes arbitrarily distributed in the 2D plane. On these nodes, a Delaunay triangulation[2] is performed. From this triangulation, the voronoi grid[3] can be easily computed: The centers of the circumstances of the Delaunay triangles form the vertices of the voronoi grid.
In such a voronoi grid, not all core nodes have the same number of neighbours as it is the case in regular raster grids. Therefor, this grid structure is nothing a regular CNN can process anymore and it enforces the use of other methods such as GNNs.

Boundary Control

An important question when building earth surface dynamics models is how to handle the boundaries of the underlying grid. Landlab provides the possibility to close certain boundary nodes, meaning no flux can enter or leave the grid over these nodes. By default, all links of a Landlab grid are considered being

[2]https://en.wikipedia.org/wiki/Delaunay_triangulation
[3]https://en.wikipedia.org/wiki/Voronoi_diagram

active links. If we now close a boundary node, Landlab simply sets all links connecting core nodes with this boundary node to inactive (Hobley et al. 2017).

Since this study aims towards generating a general understanding of the effectiveness of deep learning methods in the domain of erosion simulation, it is of minor importance to us how exactly we choose the grid boundary conditions. Still this should be mentioned because at the latest when it comes to designing applications for real world use cases, one has to deal with this issue.

2.1.2 Landlab Components

As already mentioned, each Landlab component is designed to simulate one specific geoscientific process. Each component needs a grid to be instantiated on and — depending on the component — certain data fields already loaded on the grid.
All the components share a simple and standardized interface: There is a method `run_one_step()` that each component has to implement. This methods simulates the component on the grid for one time step (if the simulated process is time dependant). The duration of such a time step can be freely chosen by the user.

In this study three Landlab components come into operation for the simulation of flow routing based stream power governed erosion and two more for the semi-automated generation of initial topographies.

Flow-Routing

Flow routing is the process of determining where and in which quantities water (i.e., rain) runs off and gathers on a surface (i.e., a landscape).
Landlab implements flow routing via two separate components. The first component is referred to as *flow director*, the second one is referred to as *flow accumulator*.

The flow director operates on node level: for each node it computes those neighbours that receive flow and, potentially, splits the complete cell outflow among these neighbours. There are two general approaches to implement flow directors: the *route-to-one* methods where all outflow of one cell is directed to exactly one neighbouring cell and the *route-to-many* methods where the outflow of one cell is being divided and directed to (potentially) more than

one neighbouring cell. The criterion in both cases is the topographic slope of the links between neighbouring cells.
When operating on raster grids, flow directing methods can be further separated into D4 and D8 methods, referring to the size of the neighbourhood of each core cell.

The flow accumulator on the other hand operates on grid level: It combines all the node level computations of the flow director to compute for each node of the grid a value referred to as `drainage__area`. This value describes the size of the area surrounding a node from which flow arrives at this specific node (value A in the SPE (equation 1.1)).

For this study we choose the `FlowDirectorSteepest` as our flow director which needs a field `topographic__elevation` being present on the grid it is instantiated on.
It routes all outflow of a cell along the link with the steepest slope (*route-to-one*) taking into account the D4 neighbourhood when operating on raster grids.
The name of Landlab's flow accumulator is `FlowAccumulator`.

Both the `FlowDirectorSteepest` and the `FlowAccumulator` operate time invariant — they compute their output deterministically for a given topographic profile. Therefor, their `run_one_step()` method does not require an additional argument to determine the temporal step size. Only after some erosion has taken place (i.e., the `topographic__elevation` field of the grid has changed), the values computed by `FlowDirectorSteepest` and `FlowAccumulator` may change.

Erosion

There are different Landlab components available to simulate erosion. They mainly differ in the way, they solve the SPE, which is, as already introduced, the de facto law governing all water related erosion processes on earth.

We choose the `StreamPowerEroder` for our contribution. It needs a field `drainage__area` being present on the grid it is instantiated on as it is accomplished by the `FlowAccumulator`.
It's `run_one_step()` method requires an argument `dt` which determines the time period for which to simulate erosion. Running the `StreamPowerEroder` results in an updated `topographic__elevation` field being saved on the grid nodes.

A complete flow routing based erosion model in Landlab can now be realized in a few lines of code where we assume that `grid` is a already instantiated Landlab grid with a already initialized `topographic__elevation` field being present.

```
from landlab.components import FlowDirectorSteepest
from landlab.components import FlowAccumulator
from landlab.components import StreamPowerEroder

fd = FlowDirectorSteepest(grid)
fa = FlowAccumulator(grid, flow_director=fd)
se = StreamPowerEroder(grid)

for _ in range(timestes):
    fd.run_one_step()
    fa.run_one_step()
    se.run_one_step(dt)
```

Generation of initial Topographies

Before we can use Landlab as a generative model for erosion related data, we need a set of topographies that we can use as *initial conditions* for the erosion simulation described above. Handcrafting such topographies is very time consuming and — since we need quite a lot of data — not feasible. Therefor we developed a semi automatic method to generate such initial topographies. Besides the already introduced Landlab components for flow routing based erosion, it uses the components `NormalFault` and `LinearDiffuser`.

The `NormalFault` component is designed to simulate so called *fault traces* — uplifts of the terrain along certain lines as they can occur after earthquakes for examples. For us this means topographic uplifts on one side of a line connecting two arbitrary boundary nodes of a grid. Beside the `run_one_step()` method, `NormalFault` has a method called `run_one_earthquake()`. For the sake of simplicity, it is the latter one that we use in our method, since it lets us time-invariantly uplift the terrain by a fix rate we can freely choose.

The `LinearDiffuser` component applies *linear diffusion* to the topography loaded on a grid, i.e., it transports soil from higher elevations to lower elevations. It's `run_one_step()` method requires an additional argument `dt` to determine the temporal step size. If time steps are chosen too big, applying the `LinearDiffuser` results in completely flat landscapes, if time steps are chosen small enough, it only smoothens the edges of a topographic profile.

2.2 Graph Neural Networks

In the last years, CNNs gained a lot of attention since they achieved remarkable results on a whole range of tasks. But despite their striking performance in some domains, CNNs are completely unavailable in other domains because they can only operate on rectangular data such as vectors, matrices or cubes.

However, a lot of real world data is not rectangular structured at all. It comes encoded in graphs, trees (that can be seen as a subset of graphs) or multi dimensional point clouds (that can be easily transformed into, e.g., k-nearest neighbour graphs). All these data structures are nothing a regular CNN can easily process.
GNNs can be seen as the attempt to close this gap by introducing a convolution like operator for graph structured data.

The input to a simple GNN consists of (i) n-dimensional node level feature vectors (ii) the graph's topology, e.g. in form of an adjacency matrix and (iii) optionally edge level features or weights.
Given this information, GNNs can produce output either on graph-, node- or edge-level what makes them applicable in a lot of scenarios. These reach from graph classification to edge prediction to node regression (Scarselli et al. 2009).

As we have seen in section 2.1.1, Landlab uses graphs as the data structure to represent the model domain.
One has to mention, that in the case of regular raster grids, these graphs could be regarded as rectangular structured and if only node features would be taken into account, they could be perfectly processed by regular CNNs. But as soon as edge features (i.e., weights) should used as well or at the latest when irregular voronoi grids are used, CNNs reach their limitations and it seems to be perfectly reasonable to go for GNNs.

Before describing in more detail the two graph convolution layers that have been investigated in this study, let us introduce some general terms and concepts that are related to GNNs.

2.2.1 Graph Embedding

[4] Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph with one feature vector $\mathbf{x}_i \in \mathbb{R}^n$ being associated to each node $v_i \in \mathcal{V}$, then n is called the *dimension* of the graph.

[4] This passage follows the explanations in https://towardsdatascience.com/node-embeddings-for-beginners-554ab1625d98

If we apply a function f that maps each feature vector $\mathbf{x}_i$ to a new feature vector $\mathbf{x}'_i \in \mathbb{R}^m$ usually with $m < n$ these vectors $\mathbf{x}'_i$ are called an *embedding* of $\mathcal{G}$.
Let $\mathcal{M}$ be such an embedding. In order to allow inference from $\mathcal{M}$ to the original graph $\mathcal{G}$, it is desired, that nodes being similar in $\mathcal{G}$ are similar in $\mathcal{M}$ as well, where the measure of similarity depends on the specific task to be performed. In order to preserve existing similarities, the embedding function should capture (i) the topology of the graph i.e., the connectivity of nodes with edges and (ii) all the provided node- and edge-features (i.e., weights). How exactly this is accomplished, depends on the embedding function (Grover and Leskovec 2016; Perozzi, Al-Rfou, and Skiena 2014).
GNNs can be seen as a general and yet powerful group of graph embedding functions.

2.2.2 Graph Convolution

[5] If in regular CNNs convolution is applied on e.g. an image, this means that for each pixel, information of the neighbouring pixels is *aggregated* by weighting it and summing it up. The size of this neighbourhood is defined by the size of the applied convolution filter. Since CNNs rely on rectangular structured data, the convolution filters also are of rectangular shapes, namely vectors, matrices or (hyper)-cubes.
GNNs attempt to generalize such neighbourhood aggregation onto graph structured data. The shortly introduced graph convolution operator does the exact same thing as it happens in CNNs: For each node in a given graph it aggregates information of neighbouring nodes. However, this time we do not know in advance, how many neighbours are present for each node and thus we need a different approach than sliding fixed-size convolution filters over the input data. One way to solve this problem, is to make use of the graphs *adjacency matrix.*
Again, let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph with adjacency matrix $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$. Further, let $\mathbf{X}$ be the graph's *node feature matrix* — each row of $\mathbf{X}$ represents one n-dimensional node feature vector, hence, $\mathbf{X}$ is of shape $|\mathcal{V}| \times n$. If we now multiply adjacency matrix $\mathbf{A}$ with node feature matrix $\mathbf{X}$, we obtain a new node feature matrix $\mathbf{X}'$, again of shape $|\mathcal{V}| \times n$, that aggregates for each node the information of its direct neighbours' node features.

Let us look at an example[6] to make things clear — for the sake of simplicity we use scalar node features:

[5]This passage follows the explanations in https://towardsdatascience.com/the-intuition-behind-graph-convolutions-and-message-passing-6dcd0ebf0063

[6]This example together with figure 2.2 are taken from https://towardsdatascience.com/

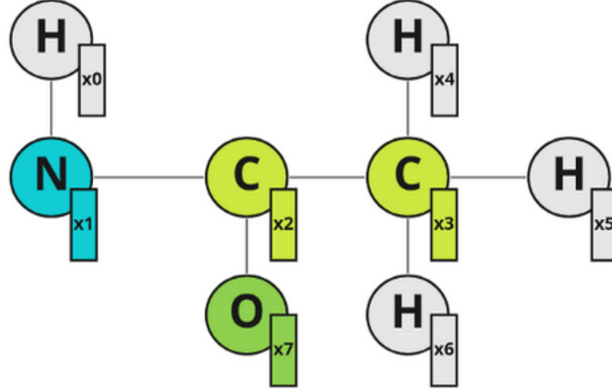

Figure 2.2: Example of a graph with scalar node features

Let us now see, what happens, if we apply the procedure described above:

$$\underbrace{\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{\mathbf{A}} \times \underbrace{\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix}}_{\mathbf{X}} = \underbrace{\begin{bmatrix} x_1 \\ x_0 + x_2 \\ x_1 + x_3 + x_7 \\ x_2 + x_4 + x_5 + x_6 \\ x_3 \\ x_3 \\ x_3 \\ x_2 \end{bmatrix}}_{\mathbf{X}'} \tag{2.1}$$

Note, that if we want to include the own feature values of a node into this neighbourhood aggregation, we have to set the entries on the diagonal of the adjacency matrix to 1. Further, if we want to include edge weights into this process, we can use them instead of having 0 and 1 entries in the adjacency matrix.

So far, for each node we aggregate information from neighbours in a 1-hop distance meaning nodes that can be reached using a single edge. But how can we collect information of nodes, that are further away like we would in CNNs by choosing a bigger kernel size? We simply use higher powers of $\mathbf{A}$ to be multiplied with $\mathbf{X}$, where using $\mathbf{A}^h$ means collecting information of nodes that are h hops away. Let us demonstrate this for $h = 2$:

the-intuition-behind-graph-convolutions-and-message-passing-6dcd0ebf0063

$$\underbrace{\begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 3 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 4 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}}_{\mathbf{A}^2} \times \underbrace{\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix}}_{\mathbf{X}} = \underbrace{\begin{bmatrix} x_0 + x_2 \\ 2x_1 + x_3 + x_7 \\ x_0 + 3x_2 + x_4 + x_5 + x_6 \\ x_1 + 4x_3 + x_7 \\ x_2 + x_4 + x_5 + x_6 \\ x_2 + x_4 + x_5 + x_6 \\ x_2 + x_4 + x_5 + x_6 \\ x_1 + x_3 + x_7 \end{bmatrix}}_{\mathbf{X}'} \tag{2.2}$$

Putting it all together, a general graph convolution filter $\mathbf{P}$ of size h can be formally expressed as

$$\mathbf{P} = w_0\mathbf{A}^0 + w_1\mathbf{A}^1 + w_2\mathbf{A}^2 + \cdots + w_h\mathbf{A}^h \tag{2.3}$$

so that

$$\mathbf{X}' = \mathbf{PX}. \tag{2.4}$$

The w_i are weights, that determine the influence of the node features in i-hop distance in the updated node feature matrix $\mathbf{X}'$.

2.2.3 Message Passing

[7] Considering a 1-hop neighbourhood, equation 2.3 could be also expressed as

$$\mathbf{x}_i' = \mathit{update}(\mathbf{x}_i, \mathit{aggregate}(\mathbf{x}_j, j \in \mathcal{N}(i))) \tag{2.5}$$

where $N(i)$ represents the neighbourhood of node i. This way of looking onto graph convolution is called a *message passing* mechanism. In the basic graph convolution operator as described in equation 2.3, *update* and *aggregate* are simple summation functions. But in fact, all permutation invariant functions like *sum*, *mean*, *min*, *max* or more sophisticated handcrafted functions could be used. So the expressive power and flexibility of graph convolution is highly increased, if we look at it as a message passing process.

[7] This passage follows the explanations in https://towardsdatascience.com/the-intuition-behind-graph-convolutions-and-message-passing-6dcd0ebf0063

Further, we can add learnable weights $\mathbf{W}_1$ and $\mathbf{W}_2$ to equation 2.5 where $\mathbf{W}_1$ weights the components of a nodes own feature vector and $\mathbf{W}_2$ weights the components of the nodes neighbours feature vectors what leads us to

$$\mathbf{x}_i' = update(\mathbf{W}_1\mathbf{x}_i, aggregate(\mathbf{W}_2\mathbf{x}_j, j \in \mathcal{N}(i))). \tag{2.6}$$

This equation forms the theoretical basis of all GNNs. We can iteratively apply it to the nodes of a graph and use gradient descent techniques to adjust the weights $\mathbf{W}_1$ and $\mathbf{W}_2$ after each iteration.

PyTorch Geometric provides a wide range of graph convolution layers. All of them implement equation 2.6 in a unique and use-case dependant way. In fact, all classes representing such graph convolution layers have to inherit from a base class called `MessagePassing`.

As discussed above, applying equation 2.6 aggregates information within a 1-hop neighbourhood only. In order to increase the size of the neighbourhood, we can use multiple instances of equation 2.6 in a row in each iteration, each instance enlarging the neighbourhood by 1 hop. In PyTorch Geometric, this means to simply build networks with multiple convolution layers.

So let us now introduce the two graph convolution layers that have been used in this study.

2.2.4 GCNConv

PyTorch Geometric's `GCNConv` layer implements equation 2.4 as it is described in *Semi-Supervised Classification with Graph Convolutional Networks* (Kipf and Welling 2016).

In essence, it is

$$\mathbf{X}' = \mathbf{AXW} \tag{2.7}$$

where again $\mathbf{A}$ is the graphs' adjacency matrix, $\mathbf{X}$ is the node feature matrix (or its representation for the hidden network layers) and $\mathbf{W}$ is the learnable weight matrix.

Note that the `GCNConv` layer supports the use of scalar edge weights. If present in the graph, the edge weights are used instead of the 1 entries in $\mathbf{A}$.

The model as simple as it is in described in equation 2.7 has two drawbacks: (i) Multiplying with $\mathbf{A}$ means that for each node only information of its neighbours is aggregated but not of the node itself unless the graph has self loops what is not the case in general and (ii) typically $\mathbf{A}$ is not normalized so multiplication with $\mathbf{A}$ completely changes the scale of the feature vectors in $\mathbf{X}$.

To solve problem (i), we can enforce self loops by adding the identity matrix to $\mathbf{A}$ what leads us to

$$\mathbf{X}' = \hat{\mathbf{A}}\mathbf{X}\mathbf{W} \tag{2.8}$$

with $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ where $\mathbf{I}$ is the identity matrix.

To solve problem (ii), matrix $\hat{\mathbf{A}}$ needs to be normalized. The authors suggest a symmetric normalization what finally leads us to

$$\mathbf{X}' = \hat{\mathbf{D}}^{-\frac{1}{2}}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-\frac{1}{2}}\mathbf{X}\mathbf{W} \tag{2.9}$$

where $\hat{\mathbf{D}}$ is the *node degree matrix*[8] of $\hat{\mathbf{A}}$.

The node-wise formulation of equation 2.9 is given by

$$\begin{aligned} \mathbf{x}'_i &= \mathbf{W} \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{e_{j,i}}{\sqrt{\hat{d}_j \hat{d}_i}} \mathbf{x}_j & (2.10) \\ &= \mathbf{W}\mathbf{x}_i e_{i,i} + \mathbf{W} \sum_{j \in \mathcal{N}(i)} \frac{e_{j,i}}{\sqrt{\hat{d}_j \hat{d}_i}} \mathbf{x}_j & (2.11) \end{aligned}$$

where $e_{j,i}$ denotes the weight of the edge from source node j to target node i and $\hat{d}_i$ denotes the degree of node i. Note that if edge weights are present, the degree is the sum of the weights of all incoming edges.

If we now compare equation 2.10 and equation 2.6, we can state that in the `GCNConv`-layer (i) *update* is a simple summation function (ii) *aggregate* computes for each node the normalized and potentially edge-weighted sum of the node features of neighbouring nodes and (iii) instead of two weight matrices $\mathbf{W}_1$ and $\mathbf{W}_2$ a shared weight matrix $\mathbf{W}$ is used.

[8]The *node degree matrix* of a graph is a diagonal matrix that contains for each node of the graph its degree. A nodes' degree is the number of (incoming) edges that are connected with this node

Limitations

This approach assumes, that all of the graphs present during training have the same number of nodes. During training, it is the embedding of these graphs that is being learned directly and not, e.g., a function thereof. Therefor, this approach fails, when it comes to generalizing to unseen nodes since only embeddings can be predicted for graphs that have the exact same number of nodes as the ones being present during training.
If we want to predict an embedding for a graph with only one node added, this means that we have to retrain the whole model, this time using graphs, that also have one node added.
However, a lot of real world data that comes in graph encoded form (i.e., citation networks, social media, the world wide web), is not that static. It is rather characterized by a ongoing fluctuation of nodes and edges. This means, that for a lot of real world applications, generalizing to unseen nodes is a very crucial property.

2.2.5 SAGEConv

PyTorch Geometric's `SAGEConv` layer implements graph convolution as it is described in *Inductive Representation Learning on Large Graphs* (Hamilton, Ying, and Leskovec 2017).

The goal of this approach is not to learn an embedding for fixed size graphs directly as it is done in (Kipf and Welling 2016) but rather to learn a function that generates embeddings for single nodes depending on their neighbourhood. This is a effective strategy in order to mitigate the limitations of the `GCNConv` layer described in the previous section. For example it is possible when using `SAGEConv`, to use a subset of a graph as the training data and to then generate meaningful embeddings for an other, unseen subset of the graph. Like this, generalizing to unseen nodes is not a problem any more what makes `SAGEConv` applicable in domains, where nodes are often added or deleted from a graph.

In fact, *GraphSAGE* as the authors call their approach, can be seen as a whole framework for different specific convolutional operators, as it allows for different types of neighbourhood aggregation. It generates embeddings using the algorithm at the top of the next page.

The intuition behind this algorithm is, that in each iteration of the outer loop (i.e., each layer of the neural network), each node aggregates information from its direct neighbours (line 4) using iteration specific (i.e., layer specific) aggregator functions. Some of the aggregator functions available

Algorithm 1: GraphSAGE embedding generation (i.e., forward propagation) algorithm

Data: Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, input features $\{\mathbf{x}_v \ \forall v \in \mathcal{V}\}$, depth K, weight matrices $\mathbf{W}_k \ \forall k \in \{1, \dots, K\}$, non-linearity σ, differentiable aggregator functions $AGGREGATE_k \ \forall k \in \{1, \dots, K\}$, neighborhood function $\mathcal{N} : v \mapsto 2^{\mathcal{V}}$

Result: Vector representations $\mathbf{z}_v \ \forall v \in \mathcal{V}$

1 $\mathbf{h}_v^0 \leftarrow \mathbf{x}_v \ \forall v \in \mathcal{V}$
2 **for** $k = 1..K$ **do**
3   **for** $v \in \mathcal{V}$ **do**
4         $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow AGGREGATE_k(\{\mathbf{h}_u^{k-1} \ \forall u \in \mathcal{N}(v)\})$
5         $\mathbf{h}_v^k \leftarrow \sigma\big(\mathbf{W}^k \cdot CONCAT(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\big)$
6   **end**
7     $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2 \ \forall v \in \mathcal{V}$
8 **end**
9 $\mathbf{z}_v \leftarrow \mathbf{h}_v^K \ \forall v \in \mathcal{V}$

are parametrized and thus can be learned during training. The aggregated information is then concatenated with the information of the node itself, weighted with iteration specific (i.e., layer specific) weight matrices and passed through a non-linear activation function (line 5). The *CONCAT* operation prevents nodes that have distinct feature vectors in the original vector space from being mapped to identical values in the embedding space. In line 7, the result of each iteration (i.e., layer) is being l_2 normalized. In the actual PyG implementation, this step is optional. Line 9 serves notational convenience only.

However, in this study we use the mean aggregator function and in contrast to the other aggregator functions available, the mean aggregator is not parametrized. This makes it a deterministic function that is not being learned during training. Further, the *CONCAT* operation in line 5 of the algorithm is not performed when using the mean aggregator. Instead, a nodes own feature vector is being weighted with a separate weight matrix and then added to the aggregated (and weighted) information of neighbouring nodes.

This leaves us with a node wise update rule that is given by

$$\mathbf{x}_i' = \mathbf{W}_1 \mathbf{x}_i + \mathbf{W}_2 \cdot \mathrm{mean}_{j \in \mathcal{N}(i)} \mathbf{x}_j \tag{2.12}$$

If we now compare equation 2.12 to the `GCNConv` update rule described in equation 2.10, we can point out four differences: (i) mean is used instead of sum for neighbourhood aggregation (ii) Two weight matrices are used

instead of one to determine the influence of the components of a nodes own feature vector and the components of its neighbours feature vectors (*iii*) `SAGEConv` does not provide the possibility to take edge weights into account (*iv*) The update rule itself does not produce normalized edge features. As already mentioned, `SAGEConv` provides the possibility to produce normalized embeddings by applying l_2 normalization to the already updated feature vectors.

If we want to express the update rule of equation 2.12 in terms of the message passing scheme of equation 2.6, *aggregate* is the mean function and *update* is the sum function.

Chapter 3

Method

In this chapter, we first describe in detail, how we use Landlab to generate grap-based erosion data as we need it to train and to test different types of PyG GNNs. We then continue by presenting three different GNN architectures, differing in the number of neural network layers being used. The Chapter is concluded by a brief description of the training setup for our experiments.

3.1 Data

As already discussed in chapter 1, we do not use real world data for our experiments. Instead we use Landlab to generate all the data we need.
The complete data generation process can be divided into four steps: (*i*) Initializing a Landlab grid of the desired size an type (*ii*) Generating a initial topography and loading it on the grid (*iii*) Simulating erosion for the desired number of time steps and saving all the intermediate results (*iv*) Encoding these results in a way such that they can be processed by PyG GNNs.

3.1.1 Grid Type and Size

As grid types for our experiments, we use rectangular raster grids and voronoi grids as they are described in chapter 2.1.1.
The grid sizes in our experiments regarding number of grid nodes are $30 \times 30 = 900$, $50 \times 50 = 2500$ and $80 \times 80 = 6400$.
In the case of raster grids, we have a spacing of $1m$ between neighbouring nodes, so the areas described by these grids have a size in m^2 of about the number of grid nodes.
By design, this also holds for the voronoi grids — the number of nodes approximates in m^2 the area that is being described. However, this time the nodes are randomly distributed within this area so that there is no fixed spacing between neighbouring nodes. This also means, that the number of

neighbouring nodes for each node varies.

3.1.2 Initial Topographies

Handcrafting initial topographies is a very time consuming task and since we need quite a lot of them, this approach is highly infeasible.
But before we present in detail our semi-automatic method to solve this problem, let us have a look at the requirements, that such initial topographies have to fulfill.

Requirements

As we have outlined in chapter 1, stream power governed erosion is characteristic in areas with relatively large mean slopes (i.e., mountainous areas). Since this precondition is quite general, we make some further assumptions for the sake of simplicity: We always choose the *northern* boundary nodes to have the highest average topographic elevation and the *southern* boundary nodes to have the lowest average elevation. In the case of raster grids, we further close the *western*, *northern* and *eastern* boundaries so that flux generally moves from *north* to *south* and can only exit the grid over the *southern* grid boundary. In the case of voronoi grids, boundary control is limited because the algorithm that identifies the boundary nodes does not collect all of them[1]. Therefor we leave all grid boundaries open when operating on voronoi grids.

Method

We start by stacking several so called *fault scarps* over another. This can be done with Landlabs `NormalFault` component (see chapter 2.1.2). This component lets us define a line (i.e., a fault scarp) from one edge of a grid to an other and elevate the terrain on one side of that line.
In order to meet the requirements described above, we pre-define a set of fault scarps that elevate *western*, *north-western*, *northern*, *north-eastern* or *eastern* parts of the terrain. From this set $n \in \{5, 10, 15, 20\}$ fault scarps are randomly selected. This results in a topographic profile with very clear and sharp edges

Next, we apply some stream power based erosion since we can easily assume, that erosion has already taken place before we start our simulation. This is achieved using the `FlowDirectorSteepest`, the `FlowAccumulator`

[1] https://github.com/landlab/landlab/issues/885

and the `StreamPowerEroder` components (see chapter 2.1.2). The time period for which we apply erosion varies between 50 and 400 years depending on the grid size. We choose it by visually checking the results.

Now, we want to control the topographic slope. Therefor we center the topographic elevation we have so far around 0 by subtracting the mean topographic elevation of all nodes from each single nodes' topographic elevation. Now we can multiply the complete topographic elevation with a value $x \in (0, 2)$ where any $x < 1$ makes the slope smaller and any $x > 1$ makes it bigger. Suitable values for x are again chosen by visually checking the results. We choose $x \in \{0.3, 0.6, 0.9\}$.
After adjusting the steepness, we lift the topographic elevation up again until the lowest point is $20m$ above 0.

Next, we want to get rid of the sharp edges in our topographic profile that are still present from the first step. Therefor, we apply the `LinearDiffuser` component (see chapter 2.1.2). Here it is important, to choose the time period small enough because otherwise the result is a completely flat landscape. Depending on the grid size, we choose time periods between 10 and 400 years — again based on visually checking the results.

Finally, we add some random noise $\epsilon \in [0m, 1.5m]$ to the topographic elevation.

The results look like this (*f.l.t.r.*: Raster900, Voronoi900, Voronoi2500, Voronoi6400):

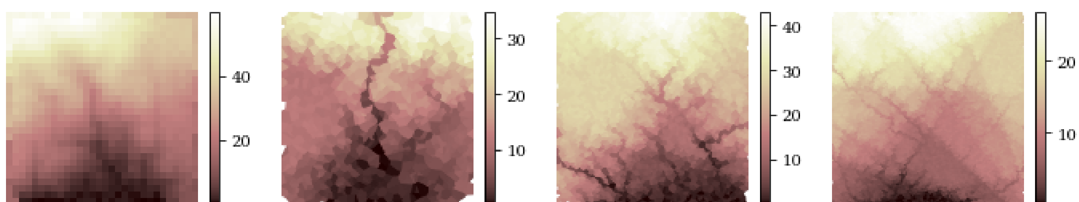

Figure 3.1: Example initial topographies

For each grid type and size, we generate 36 initial topographies for training and 36 initial topographies for testing our GNNs later on.
It has to be mentioned, that in the case of voronoi grids, not only each topographic profile is unique but also each underlying graph structure, because for each initial topography we randomly re-distribute the grid nodes.

3.1.3 Erosion Timeseries

For each initial topography, we now simulate stream power based erosion for 50 time steps using Landlab's `FowDirectorSteepest`, `FlowAccumulator` and `StreamPowerEroder` components (see chapter 2.1.2).
After these 50 time steps we want about 3/4 of the terrain to be gone, so we choose the size of the time steps accordingly. For the small grids each time step represents 100 years of erosion so in total we simulate for 5000 years. For the medium grids each time step represents 120 years of erosion so we simulate for 6000 years. For the large grids each time step represents 150 years of erosion so we simulate for 7500 years.

3.1.4 Encoding

While simulating erosion as described above, we safe the results of each iteration. Therefor we use PyG's `Data` object which is designed to represent arbitrary graph related information. Most importantly, it can hold n-dimensional node features, the graph structure (i.e., all the edges being present in the graph), n-dimensional edge features and a target to train against.
In each iteration, we instantiate a new `Data` object. As node features, we use the topographic elevation *before* running the flow routing and erosion components on the grid. The graph structure is represented by all the *active* links of the grid — this excludes links that touch at least one closed boundary node. Edge features are not used for our experiments although grid attributes like link slopes or lengths would be legitimate candidates.
As node-level target we use the *difference* of the topographic elevation *before* and *after* running the flow routing and erosion components.

So in total, we generate 50 unique data points (i.e., graphs) from each initial topography. This results in data sets with 1800 data points for each possible combination of grid size, grid type and mode (i.e., train or test).

3.2 Models

In this section, we present a 32-layer, a 48-layer and a 80-layer GNN architecture. All of them can be implemented either using the `GCNConv` layer (see 2.2.4) or using the `SAGEConv` layer (see 2.2.5).
But before we present our architectures, we want to briefly discuss the task we aim to solve with them.

3.2.1 Goal

The overall goal is to build a erosion simulation pipeline in which the three Landlab components `FlowDirectorSteepest`, `FlowAccumulator` and `StreamPowerEroder` are replaced with a trained GNN. As a first input, this GNN takes one of our initial topographies and predicts the change of topographic elevation on node level. With this information, we can compute an updated topographic profile, which we can then feed into the GNN again to compute the next update.
The task we are facing is thus a node level linear regression task: For each node of a graph we feed as input into our GNN we want to predict a small continuous value.

Since the nature of this study is very explorative, we do not know from the beginning, which network architectures fit best for the task we want to accomplish. Therefor, the GNN architectures we are about to present follow two simple design principles. (*i*) We use only graph convolution layers followed by a fully connected output layer — features like dropout or normalization layers are not applied (*ii*) The number of convolution layers is about the square root of the number of the input graph's nodes. Remember that the number of convolution layers in a GNN defines the size of the neighbourhood (in hops) from which each node can aggregate information (see chapter 2.2).

3.2.2 32-layer Architecture

This architecture aims towards processing our small, 900-node grids. It has 32 graph convolution layers that produce output with a different number of channels. The first five layers produce 256 output channels, followed by five layers with 128 output channels, five layers with 64 output channels, five layers with 32 output channels, five layers with 16 output channels, five layers with 8 output channels and two layers with 4 output channels. As non-linearity we choose the `ReLU` after each convolution layer.
The last layer that solves the regression task, is a fully connected linear layer with one output channel.
We refer to this architecture as GCN32 and SAGE32 respectively.

3.2.3 48-layer Architecture

This architecture aims towards processing our medium, 2500-node grids. It has 48 graph convolution layers that produce output with a different number of channels. The first seven layers produce 256 output channels, followed by

seven layers with 128 output channels, seven layers with 64 output channels, seven layers with 32 output channels, seven layers with 16 output channels, seven layers with 8 output channels and six layers with 4 output channels. As non-linearity we choose the `ReLU` after each convolution layer.
The last layer that solves the regression task, is a fully connected linear layer with one output channel.
We refer to this architecture as GCN48 and SAGE48 respectively.

3.2.4 80-layer Architecture

This architecture aims towards processing our large, 6400-node grids. It has 80 graph convolution layers that produce output with a different number of channels. The first twelve layers produce 256 output channels, followed by twelve layers with 128 output channels, twelve layers with 64 output channels, twelve layers with 32 output channels, twelve layers with 16 output channels, twelve layers with 8 output channels and eight layers with 4 output channels. As non-linearity we choose the `ReLU` after each convolution layer.
The last layer that solves the regression task, is a fully connected linear layer with one output channel.
We refer to this architecture as GCN80 and SAGE80 respectively.

3.3 Training

In this section we briefly describe the training setup we use for our experiments.

The criterion we apply is the smooth L_1-loss. For network outputs $\hat{y}$, targets y and a batch size of N, this loss function can be formally described as

$$\ell(\hat{y}, y) = L = \{l_1, \dots, l_N\}^T \tag{3.1}$$

with

$$l_n = \begin{cases} 0.5 \cdot (\hat{y}_n - y_n)^2/\beta & \text{if } |\hat{y}_n - y_n| < \beta \\ |\hat{y}_n - y_n| - 0.5 \cdot \beta & \text{otherwise.} \end{cases} \tag{3.2}$$

The smooth L_1-loss thus behaves similar to L_1-loss (*mean absolute error*), if the error exceeds a threshold β and similar to L_2-loss (*mean squared error*) otherwise. We use batch sizes of 32 for the 6400-node grids, 64 for the 2500-node grids and 128 for the 900-node grids and stick to the convention to set $\beta = 1$.

The *reduced* batch error as it is finally used for optimization is computed as the mean error over all data points of a batch. Formally this is

$$\ell(\hat{y}, y)_{reduced} = \text{mean}(L) \tag{3.3}$$

All models are trained for 5000 epochs. We use the Adam optimizer with default parameters except for the learning rate. Figuring out the best learning rates for the different network architectures and grid sizes is one goal of the experiments we present in the next section. Therefor we train in each experimental condition with three different learning rates: 0.0001, 0.00005 and 0.00001 and pick the best result.

Each model was trained on a NVIDIA GeForce RTX 2080 Ti GPU. The training times vary depending on the grid size and the applied learning rate.

Chapter 4

Results

In this section, we present the results we obtain from applying the method proposed in the previous chapter on different grid types and grid sizes.
In each experimental condition, we train with three different learning rates to see, which one is suited best to solve the task. The applied learning rates are 0.0001, 0.00005 and 0.00001.

4.1 Questions

In chapter 3.2.1 we have described the goal of this study from a methodical point of view. So let us now have a look at the questions we want to find answers for by applying our method.
The three main questions are *(i)* Which one of the two graph convolution layers (`GCNConv` and `SAGEConv`) solves the task better? *(ii)* Does the grid type have any impact on the results? *(iii)* How does the approach scale?

As a first step, we try to answer question (*i*) and (*ii*) by applying the 32-layer network architectures on the 900-node grids.
We then only keep the better one of the two convolution layers and the better one of the two grid types for the proceeding experiments.
To answer question (*iii*), we apply the 32- and the 48-layer architecture on the 2500-node grids as well as the 48- and the 80-layer architecture on the 6400-node grids.

4.2 Evaluation Metrics

To evaluate the performance of our models, our first indicator is always a visual inspection of the model predictions in comparison to the original Landlab simulations.

As a second indicator we define a *simulation loss* for our models. It is realized using PyTorch's smooth L_1-loss — the same loss we use for optimization during training. If T is the number of data points (i.e., time steps) within each simulation, it can be formally described as

$$\ell(\hat{y}, y) = \frac{\sum_{t=0}^{T} l_t}{T} \tag{4.1}$$

with

$$l_t = \begin{cases} 0.5 \cdot (\hat{y}_t - y_t)^2/\beta & \text{if } |\hat{y}_t - y_t| < \beta \\ |\hat{y}_t - y_t| - 0.5 \cdot \beta & \text{otherwise,} \end{cases} \tag{4.2}$$

where $\hat{y}$ denotes the model predictions and y denotes the Landlab baseline. The hyper parameter β is set to 1.

4.3 900-Node Grids

After defining the metrics for the evaluation, we now present the results for our 900-node grids.

4.3.1 GCN32 on Raster900

After 5000 epochs, the smallest *validation loss* (VL) during training, the *mean simulation loss* (MSL) and the required training time for the three training conditions (i.e., learning rates) are:

	0.0001	0.00005	0.00001
VL	1.921	0.663	0.741
MSL	5.523	2.690	2.869
Time[1]	6:29	5:41	5:32

Table 4.1: Losses and required training time of GCN32 on Raster900

An exemplary simulation of the model with the smallest MSL (i.e., the one that was trained with a learning rate of 0.00005) compared to the Landlab baseline looks like this:

[1]Time is given in the form *hh:mm* in this and all the following tables

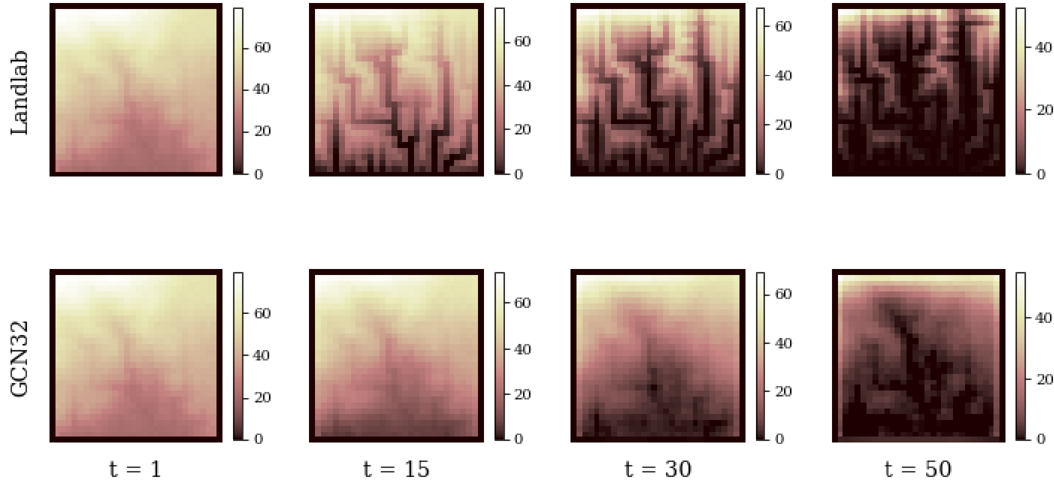

Figure 4.1: GCN32 on 900-node raster grid

As we can see, GCN32 captures the general tendency of the erosion process as the topographic elevation steadily decreases. Still it clearly fails in reproducing the fine grained structures of the channel incisions in the bedrock as Landlab produces them.

4.3.2 SAGE32 on Raster900

After 5000 epochs of training, the smallest VL, the MSL and the required training time for the three training conditions are:

	0.0001	0.00005	0.00001
VL	0.101	0.151	0.173
MSL	2.993	3.161	3.792
Time	4:25	4:35	4:24

Table 4.2: Losses and required training time of SAGE32 on Raster900

An exemplary simulation of the model with the smallest MSL (i.e., the one that was trained with a learning rate of 0.0001) compared to the baseline looks like this:

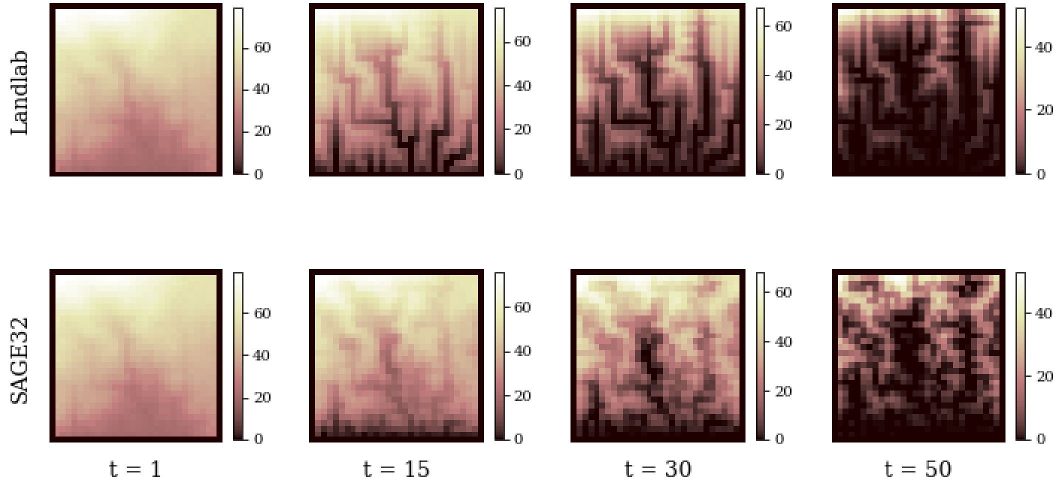

Figure 4.2: SAGE32 on 900-node raster grid

As we can see, SAGE32 also lets the topographic elevation decrease steadily and it produces a more fine grained channel structure in the bedrock as GCN32 does. Still, it does not accurately reproduce the Landlab baseline.

4.3.3 GCN32 on Voronoi900

After 5000 epochs of training, the smallest VL, the MSL and the required training time for the three training conditions are:

	0.0001	0.00005	0.00001
VL	1.828	1.936	3.551
MSL	8.399	8.623	9.510
Time	9:39	7:19	7:19

Table 4.3: Losses and required training time of GCN32 on Voronoi900

In this experimental condition, all the losses above are comparatively high. An exemplary simulation of the model with the smallest MSL (i.e., the one that was trained with a learning rate of 0.0001) compared to the baseline looks like this:

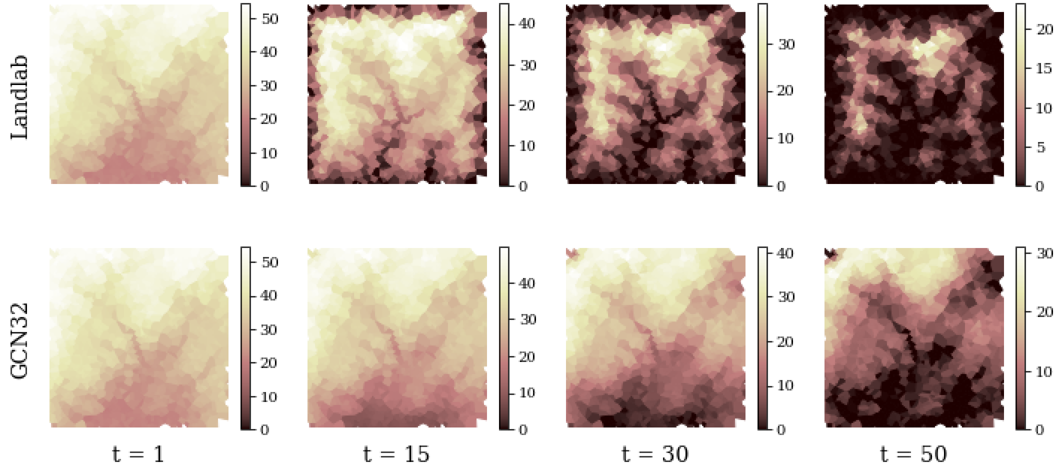

Figure 4.3: GCN32 on 900-node voronoi grid

As we can see, GCN32 lets the topographic elevation gradually decrease but the topographic pattern it produces does not mach the pattern of the baseline.

4.3.4 SAGE32 on Voronoi900

After 5000 epochs of training, the smallest VL, the MSL and the required training time for the three training conditions are:

	0.0001	0.00005	0.00001
VL	3.383	0.491	3.189
MSL	9.415	2.950	11.220
Time	5:26	5:32	5:31

Table 4.4: Losses and required training time of SAGE32 on Voronoi900

An exemplary simulation of the model with the smallest MSL (i.e., the one that was trained with a learning rate of 0.00005) compared to the baseline looks like this:

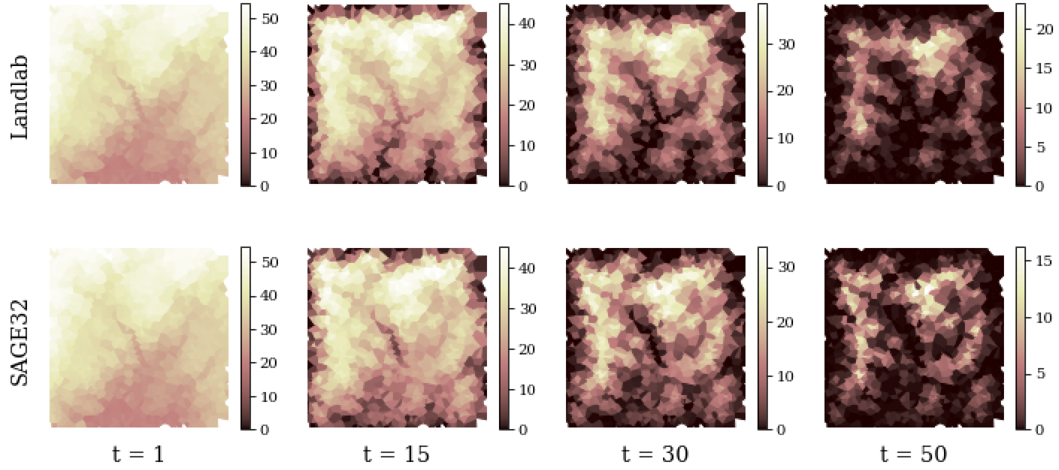

Figure 4.4: SAGE32 on 900-node voronoi grid

From all the experiments on 900-node grids, SAGE32 operating on voronoi grids produces the most accurate results. It is capable of generating a erosion pattern that highly resembles the Landlab baseline.

4.3.5 Evaluation

From visually inspecting the above results, we can state two things: (*i*) The `SAGEConv` layer clearly outperforms the `GCNConv` layer both on the raster and the voronoi grid (*ii*) The results obtained on voronoi grids are more promising than the ones obtained on raster grids.
We can further state that a comparably high MSL is a good indicator for a bad performing model. On the other hand, the best performing model (based on visual inspection) is not the one with the lowest MSL:

	`GCN32`	`SAGE32`
Raster	2.690	2.993
Voronoi	8.399	**2.950**

Table 4.5: Simulation losses on 900-node grids

As a consequence of these first results, we drop the `GCNConv` layer and the raster grids as conditions for our next experiments, in which we want to examine, how our approach scales to bigger grids.

4.4 2500-Node grids

To examine, how our approach scales to bigger grids, we now train a 32-layer and a 48-layer `SAGEConv` network on 2500-node voronoi grids.

4.4.1 SAGE32 on Voronoi2500

After 5000 epochs of training, the smallest VL, the MSL and the required training time for the three training conditions are:

	0.0001	0.00005	0.00001
VL	3.374	0.549	0.830
MSL	7.884	3.823	5.465
Time	15:37	15:41	17:47

Table 4.6: Losses and required training time of SAGE32 on Voronoi2500

An exemplary simulation of the model with the smallest MSL (i.e., the one that was trained with a learning rate of 0.00005) compared to the baseline looks like this:

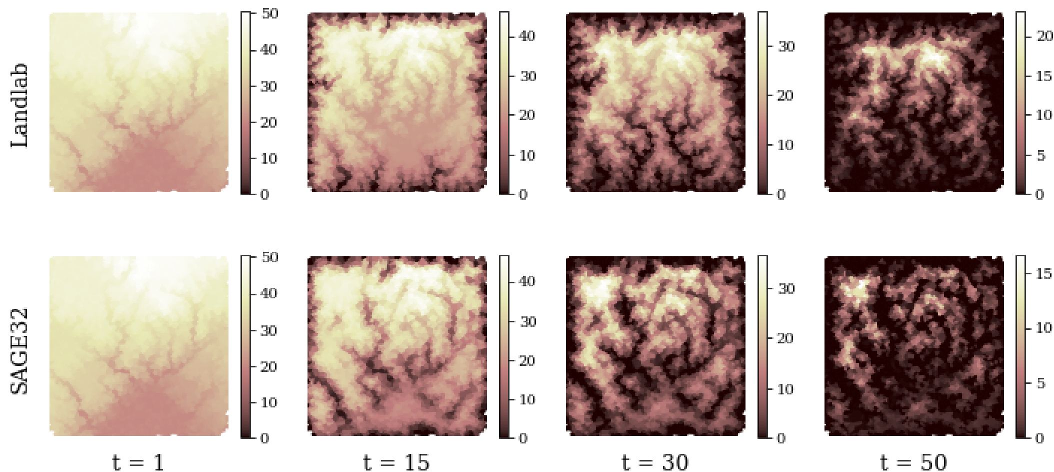

Figure 4.5: SAGE32 on 2500-node voronoi grid

As we can see, SAGE32 produces a topographic pattern that is related to the Landlab baseline. However, there are also distinct differences in the two simulation patterns. Further, in the model simulation the topographic elevation decreases slightly faster than in the baseline simulation.

4.4.2 SAGE48 on Voronoi2500

After 5000 epochs of training, the smallest VL, the MSL and the required training time for the three training conditions are:

	0.0001	0.00005	0.00001
VL	0.990	6.595	1.288
MSL	3.267	8.211	4.850
Time	22:17	22:29	22:05

Table 4.7: Losses and required training time of SAGE48 on Voronoi2500

An exemplary simulation of the model with the smallest MSL (i.e., the one that was trained with a learning rate of 0.0001) compared to the baseline looks like this:

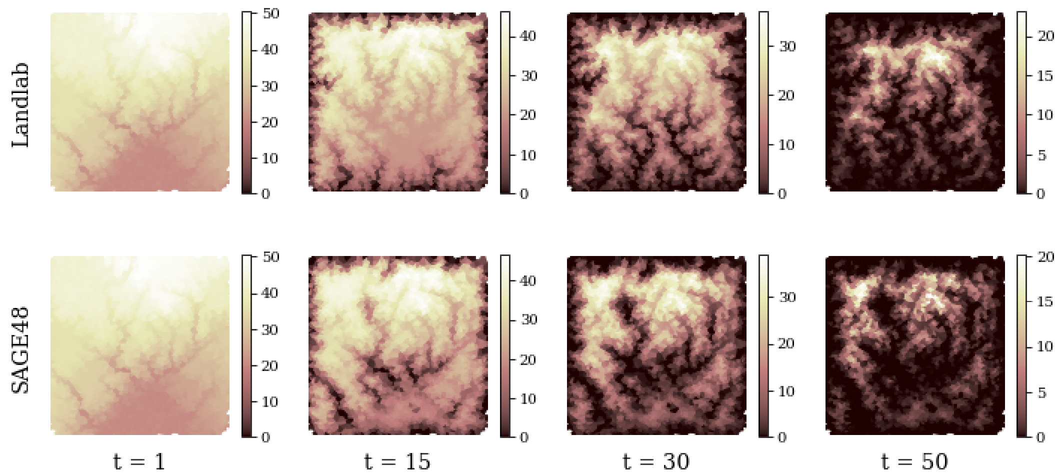

Figure 4.6: SAGE48 on 2500-node voronoi grid

We see, that SAGE48 performs quite similar to SAGE32 regarding the erosion pattern it produces. However, it reproduces the actual decrease of the topographic elevation slightly more accurate.

4.4.3 Evaluation

The above results show, that the `SAGEConv` layer in general is capable of producing meaningful predictions also on bigger grids. However, it seems like the resolution of the produced erosion patterns becomes slightly less accurate as it was the case on 900-node voronoi grids.
Considering the above results, it seems also reasonable to add layers to the

applied GNN as the number of nodes of the processed grid increases. This assumption is substantiated if we compare the MSL of the two applied network architectures.

SAGE32	SAGE48
3.823	3.267

Table 4.8: Simulation losses on 2500-node grids

4.5 6400-Node grids

To further examine, how our approach scales to bigger grids, we finally train a 48-layer and a 80-layer `SAGEConv` network on 6400-node voronoi grids.

4.5.1 SAGE48 on Voronoi6400

After 5000 epochs of training, the smallest VL, the MSL and the required training time for the three training conditions are:

	0.0001	0.00005	0.00001
VL	13.532	1.112	13.533
MSL	7.558	3.181	7.561
Time	57:38	57:27	59:52

Table 4.9: Losses and required training time of SAGE48 on Voronoi6400

An exemplary simulation of the model with the smallest MSL (i.e., the one that was trained with a learning rate of 0.00005) compared to the baseline looks like this:

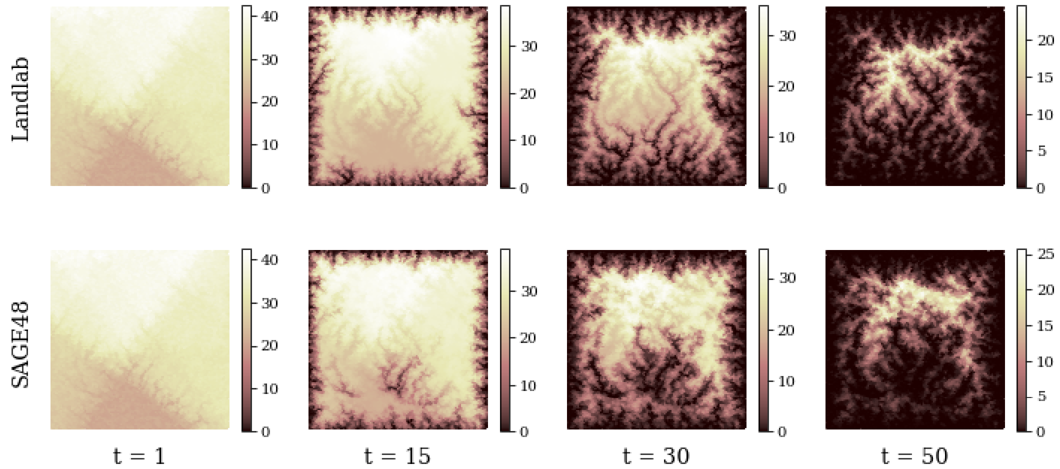

Figure 4.7: SAGE48 on 6400-node voronoi grid

We see, that — again — the model is capable of generating an erosion pattern that resembles the Landlab baseline. But, just as it was when scaling up from 900-node grids to 2500-node grids, the resolution of the produced erosion timeseries decreases: SAGE48 fails in capturing the fine grained channel incisions as they can be seen in the Landlab baseline.

4.5.2 SAGE80 on Voronoi6400

Unfortunately, the training of the models in this experimental condition failed because of the walltime (3 days) of the compute node within the SLURM cluster that hosts the GPU that was used for training.
Analysing the log files that were created during training still lets us assume, that SAGE80 failed to train properly regardless the applied learning rate. The VL in all three conditions converged very fast at around 13. For comparison: The VL of SAGE48 trained with a learning rate of 0.00005 converges at around 1.5.

4.5.3 Evaluation

We have now seen, that the `SAGEConv` layer also on grids with 6400 nodes is capable of generating erosion patterns that at least resemble the patterns that are generated by Landlab. However, the models clearly fail in catching the fine grained channel incisions as they can be seen in the baseline.
We have also seen that in the case of the 6400-node grids, the assumption does not hold that the deeper GNN produces the better results.

4.6 Further Remarks

In all experimental conditions, we are facing GNNs with a comparably high MSL that produce extremely inaccurate simulations. The training and validation losses during training indicate, that these models somehow do not train properly.

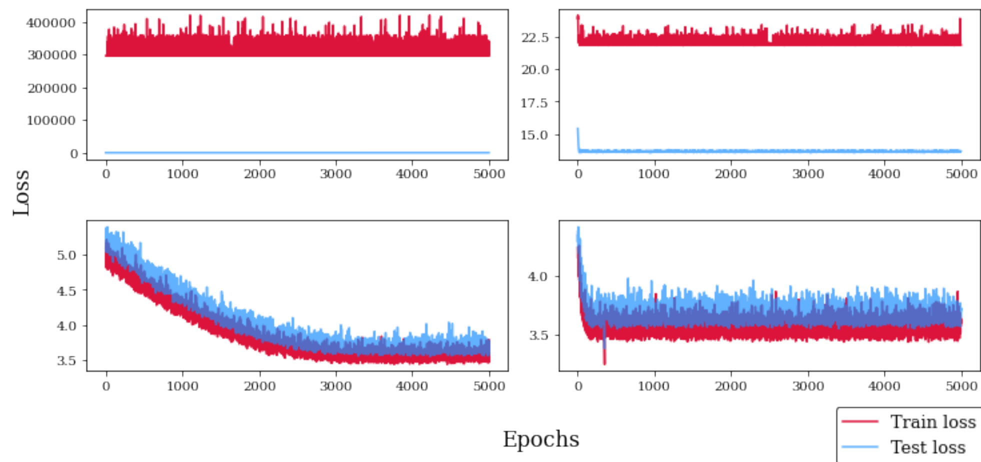

Figure 4.8: Exemplary losses of models that do not train

In some case, this might happen because the optimizer gets trapped in a local minimum of the loss function.
However, a successful training does not seem to depend on the learning rate that is applied. In the SAGE48-Voronoi2500 condition for example, good results are achieved with the learning rates 0.0001 and 0.00001 whereas the medium learning rate of 0.00005 fails. In the SAGE32-Voronoi2500 condition on the other hand, the medium learning rate produces the best simulations.
The real factors that determine whether a model succeeds or fails during training are not known to us.

Chapter 5

Conclusion

5.1 Discussion

In the previous chapters we first introduced a method to generate arbitrary amounts of graph encoded data describing stream power governed erosion processes in mountainous areas. This was achieved with the help of Landlab, a Python library for numerical modeling of earth surface dynamics.
We further introduced two graph convolution operators — the `SAGEConv`- and the `GCNConv`-layer of PyTorch's extension PyTorch Geometric. Alongside these operators, we presented three GNN architectures — one with 32, one with 48 and one with 80 convolution layers.
In a series of experiments we finally tried to learn the erosion dynamics that are inherent to our data in a end-to-end manner in different experimental conditions.

Through our experiments, several things became clear: (*i*) GNNs in general are capable of learning erosion dynamics and can produce good landscape evolution simulations on small grids with 900 nodes (*ii*) Operating on voronoi grids (versus operating on raster grids) results in more accurate simulations from a visual point of view (*iii*) The `SAGEConv` layer clearly outperforms the `GCNConv` layer both in accuracy and in required training time (*iv*) The proposed method does somehow scale to bigger grids but as the number of nodes increases, the resolution of the model predictions slightly decreases (*v*) Deeper networks do not necessarily go along with better results (*vi*) Some networks do not train and we do not know why.

The question still to be answered is, if the approach taken in this thesis is of any practical relevance. The short answer is: No. This becomes clear if we have a look at the baseline: Numerical methods for landscape evolution modeling as they are implemented in Landlab can operate on grids with up

10^8 nodes on a normal laptop and, potentially, up to 10^{10} nodes on large multi-processor computers in $\mathcal{O}(n)$ where n is the number of nodes (Braun and Willett 2013).
In this thesis we achieve satisfying results on grids with about 10^3 nodes but already lack resolution when operating on grids with more than $2.5 \cdot 10^3$ nodes.
Further, we have seen that the time to train a 80-layer model that processes grids with 6400 nodes already exceed 3 days. So training models that process grids with 10^8 to 10^10 nodes most probably is highly infeasible.

5.2 Outlook

In order to mitigate the limitation described in the previous section, we suggest two directions.

The first one takes into account two observations: (*i*) Deeper networks do not necessarily produce better results but need much longer to train (*ii*) There are still reasons that keep some models from training that we do not know of.
As a consequence, one way to achieve better results in less training time could be, to operate with shallow networks also on big grids but to refine and debug both the network architecture and the training process.
However, training time will stay an issue like this if we want to process really big grids in one piece.

Therefor we suggest a second direction. It might be even more promising but also much harder to implement.
As described in chapter 2.2.5, the `SAGEConv` operator has the capability to generalize to unseen nodes of a graph. This makes it possible to only use a subset of a graph during training and to still make meaningful predictions on other parts of the graph. Exploiting this fact, we could use a small but well trained GNN and slide it over a potentially huge grid to predict the landscape transformation patch by patch and not for the whole grid at once. This approach could (*i*) drastically reduce the training time even for very big grids and (*ii*) keep the resolution of the model predictions high.
However, a correct and consistent handling of the flux (i.e., soil) crossing the boundaries between those patches is crucial and potentially hard and time consuming to implement.

Bibliography

Barnhart, K. R. et al. (2020). “Short communication: Landlab v2.0: A software package for Earth surface dynamics”. In: *Earth Surface Dynamics* 8.2, pp. 379–397. DOI: 10.5194/esurf-8-379-2020. URL: https://esurf.copernicus.org/articles/8/379/2020/.

Braun, Jean and Sean D. Willett (2013). “A very efficient O(n), implicit and parallel method to solve the stream power equation governing fluvial incision and landscape evolution”. In: *Geomorphology* 180-181, pp. 170–179. ISSN: 0169-555X. DOI: https://doi.org/10.1016/j.geomorph.2012.10.008. URL: https://www.sciencedirect.com/science/article/pii/S0169555X12004618.

Bronstein, Michael M. et al. (July 2017). “Geometric Deep Learning: Going beyond Euclidean data”. In: *IEEE Signal Processing Magazine* 34.4, pp. 18–42. DOI: 10.1109/msp.2017.2693418. URL: https://doi.org/10.1109%5C%2Fmsp.2017.2693418.

Davy, Philippe and Dimitri Lague (2009). “Fluvial erosion/transport equation of landscape evolution models revisited”. In: *Journal of Geophysical Research: Earth Surface* 114.F3. DOI: https://doi.org/10.1029/2008JF001146. eprint: https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/2008JF001146. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2008JF001146.

Fey, Matthias and Jan Eric Lenssen (2019). *Fast Graph Representation Learning with PyTorch Geometric*. DOI: 10.48550/ARXIV.1903.02428. URL: https://arxiv.org/abs/1903.02428.

Grover, Aditya and Jure Leskovec (2016). “Node2vec: Scalable Feature Learning for Networks”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’16. San Francisco, California, USA: Association for Computing Machinery, pp. 855–864. ISBN: 9781450342322. DOI: 10.1145/2939672.2939754. URL: https://doi.org/10.1145/2939672.2939754.

Hamilton, William L., Rex Ying, and Jure Leskovec (2017). *Inductive Representation Learning on Large Graphs*. DOI: 10.48550/ARXIV.1706.02216. URL: https://arxiv.org/abs/1706.02216.

Hobley, D. E. J. et al. (2017). "Creative computing with Landlab: an open-source toolkit for building, coupling, and exploring two-dimensional numerical models of Earth-surface dynamics". In: *Earth Surface Dynamics* 5.1, pp. 21–46. DOI: 10.5194/esurf-5-21-2017.

Howard, Alan D. and Gordon Kerby (June 1983). "Channel changes in badlands". In: *GSA Bulletin* 94.6, pp. 739–752. ISSN: 0016-7606. DOI: 10.1130/0016-7606(1983)94<739:CCIB>2.0.CO;2. eprint: https://pubs.geoscienceworld.org/gsa/gsabulletin/article-pdf/94/6/739/3434551/i0016-7606-94-6-739.pdf. URL: https://doi.org/10.1130/0016-7606(1983)94%3C739:CCIB%3E2.0.CO;2.

Kipf, Thomas N. and Max Welling (2016). *Semi-Supervised Classification with Graph Convolutional Networks*. DOI: 10.48550/ARXIV.1609.02907. URL: https://arxiv.org/abs/1609.02907.

Lague, Dimitri (2014). "The stream power river incision model: evidence, theory and beyond". In: *Earth Surface Processes and Landforms* 39.1, pp. 38–61. DOI: https://doi.org/10.1002/esp.3462. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/esp.3462. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/esp.3462.

Paszke, Adam et al. (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Perozzi, Bryan, Rami Al-Rfou, and Steven Skiena (Aug. 2014). "DeepWalk". In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. DOI: 10.1145/2623330.2623732. URL: https://doi.org/10.1145%5C%2F2623330.2623732.

Scarselli, Franco et al. (2009). "The Graph Neural Network Model". In: *IEEE Transactions on Neural Networks* 20.1, pp. 61–80. DOI: 10.1109/TNN.2008.2005605.

Tucker, Gregory (Dec. 2015). "Landscape Evolution". In: pp. 593–. ISBN: 9780444538031. DOI: 10.1016/B978-0-444-53802-4.00124-X.

Whipple, Kelin (2004). "Bedrock Rivers and the Geomorphology of active Orogens". In: *Annual Review of Earth and Planetary Sciences* 32.1, pp. 151–185. DOI: 10.1146/annurev.earth.32.101802.120356. eprint: https://doi.org/10.1146/annurev.earth.32.101802.120356. URL: https://doi.org/10.1146/annurev.earth.32.101802.120356.

Whipple, Kelin and Gregory E Tucker (1999). "Dynamics of the stream-power river incision model: Implications for height limits of mountain ranges, landscape response timescales, and research needs". In: *Journal of Geophysical Research: Solid Earth* 104.B8, pp. 17661–17674.

Yavari, Shahla, Saman Maroufpoor, and Jalal Shiri (Aug. 2017). "Modeling soil erosion by data-driven methods using limited input variables". In: *Hydrol-*

ogy Research 49.5, pp. 1349–1362. ISSN: 0029-1277. DOI: 10.2166/nh.2017.041. eprint: https://iwaponline.com/hr/article-pdf/49/5/1349/483241/nh0491349.pdf. URL: https://doi.org/10.2166/nh.2017.041.

Yuan, X. P. et al. (2019). "A New Efficient Method to Solve the Stream Power Law Model Taking Into Account Sediment Deposition". In: *Journal of Geophysical Research: Earth Surface* 124.6, pp. 1346–1365. DOI: https://doi.org/10.1029/2018JF004867. eprint: https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/2018JF004867. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2018JF004867.

Zhou, Fan, Rongfan Li, and Goce Trajcevski andKunpeng Zhang (2021). "Land Deformation Prediction via Slope-Aware Graph Neural Networks". In: *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, pp. 15033–15040. URL: https://ojs.aaai.org/index.php/AAAI/article/view/17764.

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum Unterschrift