POTSDAM INSTITUTE FOR CLIMATE IMPACT RESEARCH

IfU IMA RWTH AACHEN UNIVERSITY

The present work was submitted to the
Institute of Information Management in Mechanical Engineering

Prediction of Power Grid Vulnerabilities using Machine Learning

Master's Thesis
General Mechanical Engineering

presented by
Christian Nauck

External Supervisor
Dr. rer. nat. Frank Hellmann
(Potsdam Institute for Climate Impact Research)

Internal Supervisors
M.Sc. Haoming Zhang
M.Sc. Philipp Ennen

Examiner
Prof. Dr. phil. Ingrid Isenhardt

Aachen, April 2020

Abstract

Power grids play a major role to mitigate the consequences of climate change and to succeed with the energy transition. They need to adapt to new challenges when renewable energies are integrated into the current infrastructure. Due to the decentralized structure of renewable energies as well as their reduced inertia volatility, dynamical effects are of increasing importance. Methods based on Machine Learning (ML) are a promising approach of reducing the computation costs of today's methods analyzing dynamic stability and to facilitate the fast evaluation of future power grids. In this thesis, the feasibility of applying ML-methods to predict dynamic stability is investigated. For that reason, datasets with different complexities are generated. The datasets are based on synthetic power grids and Monte-Carlo simulations computing the single-node basin stability (SNBS) as a measure of dynamic stability. These datasets with SNBS and the full topologies of the graphs are used as inputs to train different ML-models based on sophisticated *Convolutional Neural Networks* (CNNs) and newly developing *Graph Neural Networks* (GNNs). The investigations show that the SNBS of power grids can be predicted using ML, but only with low accuracy. The comparison of the different ML-models shows that the performance of CNNs is better at analyzing small graphs with 10 nodes, even though they are not designed to analyze graphs. However, for grids with 50 nodes, GNNs outperform CNNs and show potential for future applications when GNNs become more sophisticated. The analyses also show that the performance of GNNs significantly changes with the used architecture and the type of convolution while different CNN-architectures are more robust and achieve similar results. A promising observation of GNNs is the feasibility of training a model based on a dataset with small grids and applying this model to grids of larger size to reduce the computational effort, since the performance is comparable to directly train the model on the evaluated dataset.

Contents

Nomenclature

Abbreviations

$2D$	Two-dimensional
χ	Number of runs (samples)
T'	Sub-tree of a graph
$T(X)$	Chebyshev polynomial
Adam	Optimization method, name derived from adaptive moment estimation
ANN	Artificial neural network
Arma	Auto-regressive moving average
AvBS	Averaged single-node basin stability (mean of SNBS)
BS	Batch size
CNN	Convolutional neural network
DCNN	Diffusion-convolutional neural network
DenseNet	Dense Convolutional Network
FastGCN	Fast GCN
GAT	Graph Attention Network
GCN	Graph Convolutional Network
GIN	Graph Isomorphism Network
GNN	Graph Neural Network
GraphSAGE	(SAmple and aggreGatE)
HN	Highway Networks

kV	Kilovolt
LR	Learning rate
ML	Machine learning
MPNN	Message Passing Neural Network
NIC	Number of input channels
NLL	Negative Log - Likelihood
NOC	Number of output channels
ODE	Ordinary Differential Equation
plateauLR	PyTorch scheduler: ReduceLROnPlateau
PReLU	Parametric Rectified Linear Unit
ReLU	Rectified linear unit
SeLU	Scaled exponential linear unit
SGD	Stochastic gradient descent
SGNet	Simple graph convolution network
SNBS	Single-node basin stability
SS	Sample size
TA	Test accuracy
TAG	Topology adaptive graphs
TL	Test loss
V	Volt
w/o	without

Greek characters

β	Learnable parameter used for batch normalization
$\mathbf{\Lambda}$	Diagonal matrix of eigenvalues
$\mathbf{\Theta}$	Learnable diagonal matrix

Δ_k^c	Clustering coefficient
ϵ	Numerical stabilization parameter for batch normalization
γ	Learnable parameter used for batch normalization
κ	Attention coefficient
λ	Eigenvalue
μ_B	Batch mean
ω	coefficient of weight decay
ϕ^*	Fixpoint
σ	Activation function
σ_B^2	Batch variance
θ	Phase angle or vector with parameters for GNN
G_Θ	Filter of GNN

Latin characters

$\boldsymbol{A}$	Adjacency matrix
$\boldsymbol{D}$	Diagonal matrix with the degrees of all nodes
$\boldsymbol{H}$	Signal (result or input) of graph convolutional layer
$\boldsymbol{L}$	Graph Laplacian
$\boldsymbol{R}$	Pseudo-inverse of $\boldsymbol{L}$
$\boldsymbol{s}$	Complex power (vector)
$\boldsymbol{U}$	eigenvector matrix
$\boldsymbol{W}$	Weight matrix of neural network
$\boldsymbol{X}$	Node feature matrix
$\overline{\boldsymbol{A}}$	Normalized $\boldsymbol{A}$
$\overline{d}$	Mean degree
$\mathbf{v}$	Complex voltage

$\tilde{\boldsymbol{L}}$	normalized graph Laplacian which is symmetric positive semidefinite
$\tilde{d}$	Strength
b	Susceptance or bias of neural network
b_k^{flow}	Current-flow betweenness
$b_k^{shortest-path}$	Shortest-path-betweenness
c	Number of channels
$c_k^{central}$	Closeness centrality
d	Degree
e_i	Edge i of edges $\mathcal{E}$
f	Relative frequency
g	Conductance
H_v	Hidden state at node v
I	Identity matrix
i	Current
i_k^{flow}	Through current
$i_k^{s \to t}$	Through current from nodes s to t
k	Filter size of CNN or layer in GNN
k	Layer index of GNN
l	Label or target variables (output) of dataset
l_r	Path length
m	Total number of edges of a graph
M_k	Function with learnable parameters
n	Total number of nodes of a graph
n_f	Number of features
p	Perturbation

P_i	Power injected at node i
p_{ij}	Path from node i to node j
p_{ij}^s	Shortest path from node i to node j
$p_{ij}^{(\varsigma)}$	Number of paths connecting nodes i, j using ς-steps
r	Resistance
S	Number of stacks for Arma-convolutions and of GNN
s	Stack of Arma-convolutions
T	Tree of a graph
t	Time or time step
u	Magnitude of complex voltage
U_k	Function with learnable parameters
v	Voltage
v_i	Vertex i of vertices V
w	Weights of neural network or weights of $\mathcal{G}$
w_i	Weight of vertex v_i
x	Input of neural network
X_{vu}^e	Edge feature vector of edge (v,u)
Y	Nodal output
y	Admittance or output of neural network
Z	Number of hops of TAG

Other coefficients and mathematical symbols

$*_{\mathcal{G}}$	Convolutional operator on graph
$\mathcal{B}$	Batch
$\mathcal{C}$	Result of convolution
$\mathcal{D}$	Dataset

$\mathcal{E}$	Edges
$\mathcal{G}$	Graph
$\mathcal{M}$	Message (used for GNN)
$\mathcal{N}(v)$	Neighbors of vertex v
$\mathcal{P}$	Parameters of ANN
$\mathcal{R}$	Readout function
$\mathcal{S}$	Sample
$\mathcal{V}$	Vertices
$\nabla f_{\mathcal{L},\mathcal{B}}$	Average gradient for one batch
σ	Activation function
$\tilde{\mathcal{P}}$	Number of $\mathcal{P}$
$A \circ B$	Dot product of vector A and B

1. Introduction

The energy transition is one of the key aspects to meet the goals from the *Paris Agreement* [United Nations, 2015]. Renewable energies play a major role by not only replacing conventional power plants, but also generating energy for multiple sectors such as mobility and heating which will be partly electrified in the next decades. To succeed with the energy transition, power grids need to transfer electricity reliably, efficiently and increase its capacity in the future. However, integrating larger shares of renewable energies to the power grids is challenging, since renewable energies cannot simply replace conventional power plants one-by-one, due to different technical properties which are discussed in more detail after considering the current situation of power grids in industrialized countries and focusing on Germany and Europe.

Current situation of power grids in industrialized countries Nowadays power grids have the main task to reliably transport energy from producers to consumers. Large conventional power plants generate electricity which is efficiently transferred at high voltage levels and transformed to lower voltage levels depending on the consumers. The German power grids can be categorized in four voltage levels measured in volt (V) and kilovolt (kV):

- Transmission networks (220 to 380 kV),
- Distribution networks at high voltage levels (60 to 110 kV), e.g. industrial sector,
- Distribution networks at medium voltage levels (6 to 30 kV), e.g. smaller companies,
- Distribution network at low voltage levels (230 to 400 V), e.g. private households.

In Germany, the power grids usually use alternating current and the frequency as the important control variable. To ensure the stable and safe operation of power grids, the amount of produced and consumed energy has to be in an equilibrium. Besides balancing any changes in supply or demand, grid operators are always interested in keeping a static state, including during incidents such as line failures. Any fluctuations of the equilibrium result in frequency changes that must be kept at a minimum. This is controlled at the highest voltage level, where large power plants are connected. Controlling the equilibrium is supported by favorable properties of conventional power plants such as large inertias which damp any fluctuations and the ability of the generators of the power plants to automatically synchronize.

Challenge of integrating renewable energies Renewable energies are often more decentralized and even large solar farms or wind farms produce less energy than conventional power plants. In the following, we use the term renewable energies for solar and wind energy, since other types of renewable energies as water or biomass have different properties but limited capacity. As a result of the lower production of single devices of renewable energies, several challenges must be faced. As the number of producing devices increases, more devices must be controlled. Furthermore, renewable energies are connected at low or medium voltage levels to reduce transformation losses. Most renewable energies also introduce less inertia to the system which weakens the ability to damp any fluctuations. The connection at lower voltage levels as well as the reduced inertia make it more complicated to safely operate grids. Another challenge is volatility in the production since renewable energies depend on uncontrollable parameters such as day or night time and weather conditions. The volatility occurs on different time scales. While certain weather conditions can last for days or even weeks, turbulence is present on the time scale of seconds Milan et al. [2013]. Hence, it is not sufficient to control power grids at the largest voltage level, but all levels of the power grid must be considered. To safely operate power grids in the future, the impact of unavoidable fluctuations has to be limited. Approaches for the future rely on adding *artificial inertia*, e.g. by using intelligent and quick-response battery storage facilities or flywheels.

Especially grid operators on islands already face the challenge of ensuring save grid operation, because islands often run their grids independently. For example Ireland has a limit of energy produced by non-synchronous generation (mostly renewable energies). In 2018 the limit was at 75 % and in 2017, it was even lower at 65 % [Eir, 2017, 2018, O'Sullivan et al., 2014]. Even though the amount of renewable energies is increasing, any limits can still slow down the energy transition. The described challenges are possible in all grids with increasing amounts of renewable energies.

Evaluation of dynamic stability of power grids Analyzing the dynamic stability of power grids is a complex multi-dimensional problem. Formalizing the dynamics of power grids as well as developing systematic methods to investigate the stability are currently open research topics. Analytical investigations have their limitations beyond the linear regime, therefore the analysis of large disturbances requires computationally expensive numerical methods such as Monte-Carlo simulations. However, such methods can only be applied to a limited number of test cases. When planning grid expansions or integrating new kind of producers, different scenarios need to be analyzed to find an optimal solution. Hence, new methods should be less expensive.

Potential of applying machine learning to power grids Machine Learning (ML) is a promising approach to reduce the computational costs of analyzing the stability of power grids. ML can analyze and interpret patterns or structures in data and enable efficient decision making. Properly trained models can quickly evaluate complex problems and could be used to analyze many different

configurations of power grids. Before applying ML, proper methods must be developed and their feasibility has to be shown. The proposed methods must be able to identify vulnerabilities of power grids. If the detection of vulnerabilities works, the decision process of the ML-methods can also be analyzed in detail to identify the decisive criteria. This analysis could extract yet unknown relations of grid properties and the stability. Those criteria could lead to parameters that may then be further used as design parameters for future power grids.

Contribution of this thesis and methodology The idea of this thesis is the identification and comparison of promising methods to predict the dynamic stability of power grids. The focus lies on evaluating different learning methodologies for the sake of future research. The datasets are generated based on known models for creating synthetic power grids and Monte-Carlo simulations to analyze dynamic stability. Those datasets are used for the training of different ML-models. We compare sophisticated *Convolutional Neural networks* (CNNs), known from visual analyses and newly developing *Graph Neural Networks* (GNNs) and evaluate their advantages as well as disadvantages. There is no literature about predicting the dynamic stability of power grids using ML until now, so this thesis is a first evaluation of different methods.

Structure This thesis is organized as follows: We begin in chapter 2 with the theoretical background of power grids and the state-of-the art of approaches of CNNs and GNNs. One focus lies on identifying promising GNN-models, since there is not much experience regarding their application to power grids. In chapter 3 methods of analyzing the dynamic stability using ML are proposed. We also explain the generation of the datasets, which sets the basis for the training of the ML-methods. The results and discussion can be found in chapter 4. In chapter 5 we draw conclusions and suggest topics for future work. Additional material such more background of the fundamentals and the training parameters can be found in the appendix.

2. Theoretical Background

This chapter introduces the necessary background to investigate the dynamic stability of power grids using Machine Learning (ML). Firstly, networks are introduced and secondly, some fundamentals of power grids are presented. Afterwards the concept behind ML is introduced and the state-of-the art of relevant and later-used methodologies is given. Throughout the thesis we attempt to stick to the following convention for mathematical variables: real scalars are represented by simple characters a, complex numbers are bold $\boldsymbol{a}$, vectors by capital characters A and matrices bold capitol characters: $\boldsymbol{A}$. However, we deviate from this convention several times, e.g. when using sums, capital characters often represent the upper limit.

2.1. Networks

To appropriately investigate power grids a basic knowledge of networks is helpful, because the analysis and description of power grids relies on network theory. A detailed introduction to networks is given by Newman [2010]. Networks are models or systems which can be represented by graphs. Graphs consist of a set vertices v and a set edges e:

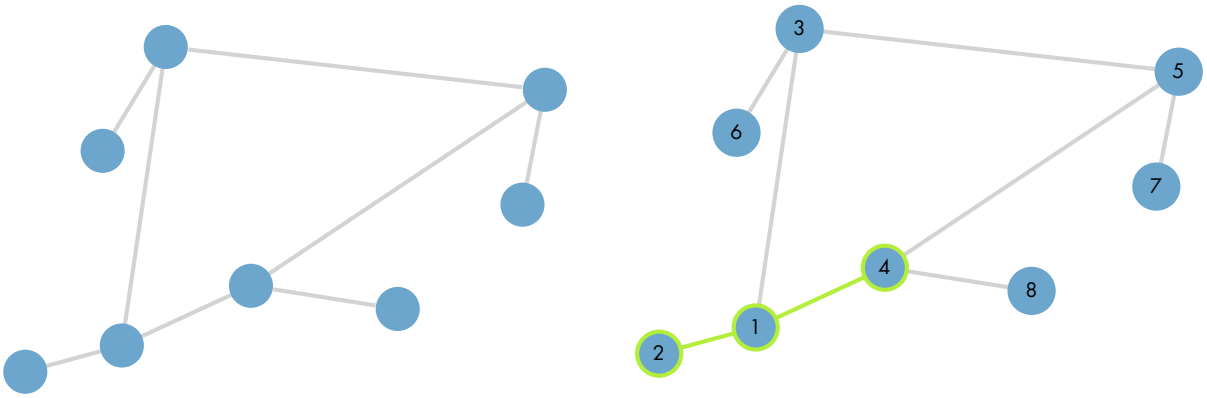
$$v_i \in \{v_1, ..., v_n\} = \mathcal{V}, \tag{2.1}$$

$$e_j \in \{e_1, ..., e_m\} = \mathcal{E}, \tag{2.2}$$

where the set $\mathcal{V}$ represents all vertices and the set $\mathcal{E}$ all edges and each edge connects two vertices. We can also note edges by e_{kl}, meaning that the edge connects nodes k and l. The total number of vertices is denoted by n, and the total number of edges is given by m. This leads to the definition of a graph as the tuple:

$$\mathcal{G} = (\mathcal{V}, \mathcal{E}) \tag{2.3}$$

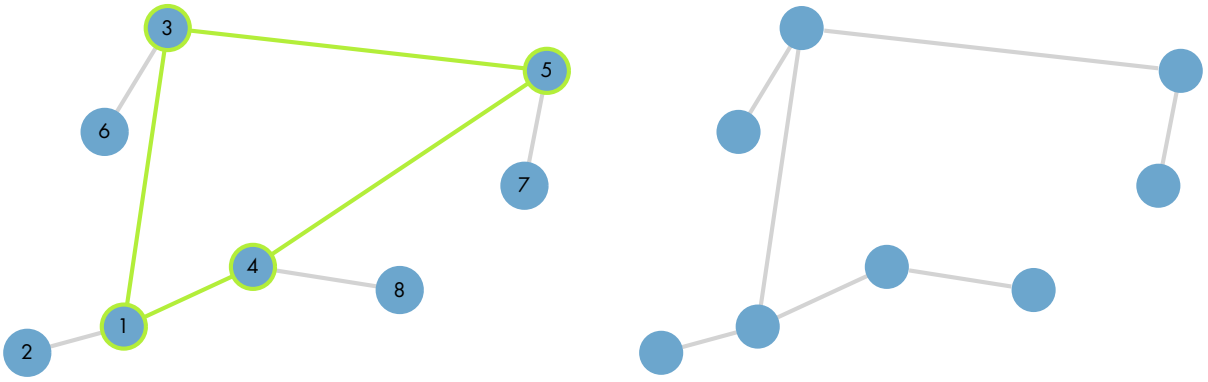
We can simply interpret vertices, which are also often called nodes, as points and edges as lines. One example of a graph is given in fig. 2.1a. When talking about graphs, we often use the term path to describe a connection between two vertices using multiple edges while perhaps passing vertices in between. One path is shown in fig. 2.1b. Depending on the shape and properties of networks we use different terms for categorization. We distinguish between connected and unconnected graphs.



(a) Graph containing eight vertices and eight edges

(b) Path from node 2 to node 4 or vice versa is highlighted

Figure 2.1.: Arbitrary graph based on Synthetic Networks [Schultz et al., 2014a]



(a) One cycle is highlighted

(b) Acyclic graph one edge has been removed

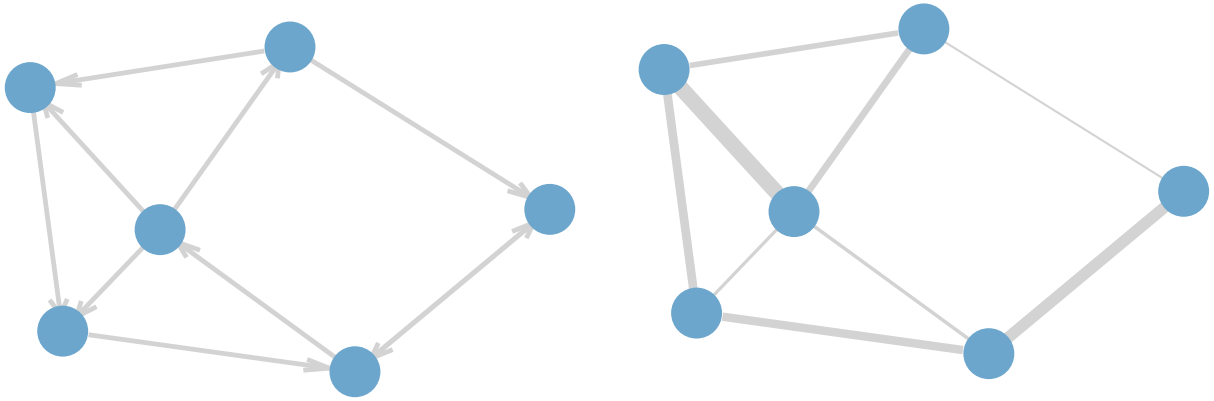
Figure 2.2.: Two graphs showing the difference of a cyclic and an acyclic graph

Connected graphs mean, that there is at least one path for each combination of vertices to go from one vertex v_i to any vertex v_j using one or multiple edges. This connection is also called the path from node i to node j, so we get the path p_{ij}. As soon as there are multiple paths between at least two nodes, there are loops within the network, so the graph contains cycles. One cycle is shown in fig. 2.2a. If there are no cycles, we call the graph **acyclic**, which can be seen in fig. 2.2b. A popular category of connected networks are trees T which are acyclic and the total number of edges m must be $n - 1$. A graph is considered a tree if

- any two vertices of the graph are connected by one path and
- removing any edge would disconnect the network.

An example of a tree is given in fig. 2.2b.

There are further edge properties such as the direction and its weight. Hyperlinks on the internet



(a) Directed graph

(b) Weighted graph, where weights are represented by the thickness of edges

Figure 2.3.: Graphs with certain edge properties

which can be represented by edges for example only work in one direction. So, we call those edges directed. Directed edges may be represented by an arrow giving the direction, which can be found in fig. 2.3a. When talking about the weights of edges, we can interpret them as a capacity. For example, when considering a network of water pipes, pipes with larger diameter can be represented by larger weights, because of the larger potential regarding the water flow. Hence, we can attribute a weight w_i to each edge e_i describing the capacity of this edge.

Another important property are the positions of the nodes of a graph. We call a graph embedded, if all nodes are attributed a position. This is the case when considering a road network for example.

2.1.1. Algebraic representation of networks

Networks can also be described in algebraic form without plotting the graph. Algebraic representations can be stored more efficiently and also enable mathematical analyses and set the basis for graph modeling. We start by considering matrices describing the topology of networks, before introducing other measures describing networks. The notations are based on Newman [2010] with small modifications.

Incidence matrix

The incidence matrix often denoted by $\boldsymbol{B}$ shows the relation between all edges and vertices. Each row represents one vertex and each column one edge. For undirected networks the following convention is applied:

$$B_{ij} = \begin{cases} 1 & \text{if } e_j \text{ is attached to } v_i, \\ 0 & \text{otherwise.} \end{cases} \tag{2.4}$$

This $n \times m$ matrix having n nodes and m edges has the following property: The sum of each column is equal to two, since each edge is connected to two vertices. For directed graphs the following convention is used:

$$B_{ij} = \begin{cases} 1 & \text{if tip of } e_j \text{ is directed to } v_i, \\ -1 & \text{if end of } e_j \text{ is directed to } v_i, \\ 0 & \text{otherwise.} \end{cases} \tag{2.5}$$

In case of directed graphs the sum of each column of $\boldsymbol{B}$ is 0, because each edge has one $+1$ and one -1 entry. This incidence matrix using directed edges is also often used for undirected graphs, because of its mathematical properties including the entries -1 when it is applied to power grids.

Adjacency matrix

The adjacency matrix $\boldsymbol{A}$ for shows the connection between all vertices for an undirected, unweighted graph in the following manner:

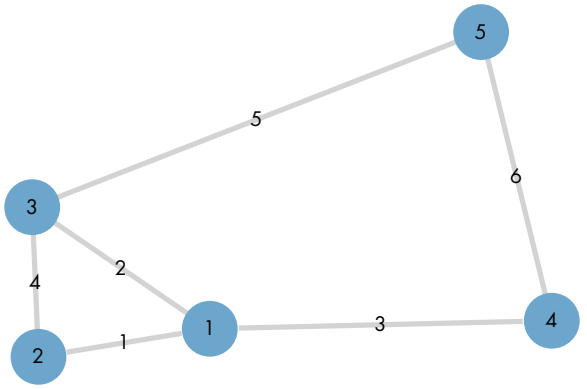
$$A_{ij} = \begin{cases} 1 & \text{if there is an edge between } v_i \text{ and } v_j, \\ 0 & \text{otherwise.} \end{cases} \tag{2.6}$$

In case of n-nodes, $\boldsymbol{A}$ has the dimension $n \times n$. In case of undirected graphs, $\boldsymbol{A}$ is always symmetric, so $\boldsymbol{A} = \boldsymbol{A}^T$. A connection between v_i and v_j means that there is also a connection between v_j and v_i. On the contrary to undirected graphs, the adjacency matrix $\boldsymbol{A}$ of directed graphs does not have to be symmetric. When considering graphs without self-edges, meaning that no vertices are connected to itself with one edge, the diagonal of $\boldsymbol{A}$ is zero. An example for a graph and its adjacency is given in fig. 2.4. In case of self-edges, the corresponding element a_{ii} is equal to 2, because a connecting edge has two ends and both ends are connected to v_i. When the same pair of vertices is connected using multiple edges, the edges can be combined to one edge representing all multiple edges by one edge. Let us say that edges:

$$e_1, e_2, ..., e_K = \mathcal{E}_1 \in \mathcal{E}, \tag{2.7}$$

all connect v_i and v_j of an undirected graph and assume that the weights represent a summable quantity. We combine $e_1, e_2, ..., e_K$ to one edge e_l by computing the weight w_l by:

$$w_l = \sum_{1}^{K} w_i. \tag{2.8}$$



$$\boldsymbol{A} = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} \begin{array}{c} \begin{array}{ccccc} v_1 & v_2 & v_3 & v_4 & v_5 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

$$\boldsymbol{A} * \boldsymbol{A} = \boldsymbol{A}^2 = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} \begin{array}{c} \begin{array}{ccccc} v_1 & v_2 & v_3 & v_4 & v_5 \end{array} \\ \begin{pmatrix} 3 & 1 & 1 & 0 & 2 \\ 1 & 2 & 1 & 1 & 1 \\ 1 & 1 & 3 & 2 & 0 \\ 0 & 1 & 2 & 2 & 0 \\ 2 & 1 & 0 & 0 & 2 \end{pmatrix} \end{array}$$

Figure 2.4.: Graph containing five vertices and 6 edges and its adjacency matrix at the top and the adjacency matrix to the power of two at the bottom

If all edges are equally weighted and have a weight of one, the resulting weight is simply the number of the multiple edges k, so $a_{ij} = K$. In case of multiple edges connecting the same vertices, the adjacency matrix $\boldsymbol{A}$ contains at least one entry other than 0 or 1, because $\boldsymbol{A}$ must contain the weight of this particular edge.

We can also use the adjacency matrix and its higher orders: $\boldsymbol{A}, \boldsymbol{A^2}, ...\boldsymbol{A^n}$ to compute the number of paths between the nodes using a specific number of edges. To give an example: By computing $\boldsymbol{AA} = \boldsymbol{A}^2$, the resulting matrix gives us the total number of paths of length 2 connecting any nodes v_i and v_j. We can expand this procedure to obtain the number of paths for longer distances by multiplying $\boldsymbol{A}$ with its result for the desired amount of times. This approach can be generalized to:

$$p_{ij}^{(\varsigma)} = \boldsymbol{A}_{ij}^{\varsigma}, \tag{2.9}$$

where $p_{ij}^{(\varsigma)}$ shows the number of paths from node i to j using ς-steps. The diagonal elements: $p_{ii}^{(\varsigma)}$ tell us the number of cycles connecting the same vertex for the given path length ς. Cycles are called loops, because starting and end points are the same.

For some applications instead of the adjacency matrix the graph Laplacian is used to describe the topology of the graph. To define the graph Laplacian, we need the term degree that we introduce in the following.

Degree and strength

The **degree** of a vertex is defined by the number of edges connected to this vertex. For directed graphs we distinguish between the in-degree and out-degree and both are equal in case of undirected graphs which we consider in the following. We can compute the degree d by using $\boldsymbol{A}$ such as:

$$d_i = \sum_{j=1}^{n} A_{ij}. \tag{2.10}$$

When building the sum of the degree of all vertices, we obtain the total number of ends of edges. Since each edge has two ends, we get the following relationship:

$$2m = \sum_{i=1}^{n} d_i. \tag{2.11}$$

The mean degree $\overline{d}$ is defined by:

$$\overline{d} = \frac{1}{n} \sum_{i=1}^{n} d_i, \tag{2.12}$$

so we can combine eq. (2.11) and eq. (2.12) to get:

$$\overline{d} = \frac{2m}{n}. \tag{2.13}$$

Besides considering the mean degree it is common to use the degree distribution to compare graphs. The degree distribution is the probability distribution of the degrees. So, we compute the degree for each node and count the occurrences of nodes having a distinct degree. In case of weighted graphs, one can also use the term **strength** which is similarly defined but takes the weights into account.

Graph Laplacian

The graph Laplacian $\boldsymbol{L}$ is a common representation of the topology of a graph and combines the adjacency matrix with the degree and is defined by:

$$\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{A}, \tag{2.14}$$

where $\boldsymbol{D}$ is a diagonal matrix containing the degree of all nodes, such as:

$$\boldsymbol{D} = \begin{pmatrix} k_1 & 0 & 0 & 0 \\ 0 & k_2 & 0 & 0 \\ 0 & 0 & \ddots & \vdots \\ 0 & 0 & \dots & k_n \end{pmatrix}. \tag{2.15}$$

Hence, the graph Laplacian combines information about nodes in terms of the degree and information of edges contained by the adjacency matrix $\boldsymbol{A}$. $\boldsymbol{L}$ is singular, so it cannot be inverted and one uses the pseudo-inverse $\boldsymbol{R}$ instead.

2.1.2. Graph and centrality measures

When looking at graphs, one often faces the challenge of categorizing graphs or identifying relevant and *central* vertices. We introduce some methods to classify and compare nodes and depending on the application different criteria are commonly used.

Shortest-path-betweenness

Shortest-path-betweenness is a criterion by Freeman [1977] to categorize vertices using the number of shortest paths. The shortest path is denoted by p_{kl}^{s} which is the shortest path from node k to node l. The shortest-path-betweenness $b_k^{shortest-path}$ of node k is defined as the number of shortest paths passing through node k and can be expressed by:

$$b_k^{shortest-path} = \sum_{s \to t} \frac{n_k^{s \to t}}{g^{s \to t}}, \tag{2.16}$$

where $n_k^{s \to t}$ is the number of shortest paths from node s to t going through node k and $g^{s \to t}$ is the total number of shortest paths from node s to t. We do not explicitly use this criterion but uses its concept to introduce the current-flow betweenness.

Current-Flow betweenness

Proceeding with identifying central nodes, we do not simply count paths, but consider some type of flow passing through nodes instead. This leads to the current-flow betweenness explained in more detail by Schultz et al. [2014a]. The flow passing through nodes takes the weights of the edges into account. Hence, we compute the through current $i_k^{s \to t}$ for $k \neq s, t$ as:

$$i_k^{s \to t} = \sum_{l=1}^{n} |A_{kl}(R_{ks} + R_{lt} - R_{kt} - R_{ls})|, \tag{2.17}$$

where $\boldsymbol{R}$ is the pseudo-inverse of the graph Laplacian $\boldsymbol{L}$. Then we can build the average over all pairs of s and t and obtain the current-flow betweenness for $k \neq s, t$:

$$b_k^{flow} = \sum_{s=1}^{t} \sum_{l=1}^{n} |A_{kl}(R_{ks} + R_{lt} - R_{kt} - R_{ls})|. \tag{2.18}$$

Furthermore, it is common to normalize the centrality measures, for example by: $\frac{2}{(n-1)(n-2)}$, which we also use in this thesis.

Closeness centrality

The closeness centrality is based on the shortest paths of the network and is defined as the reciprocal of the sum of the shortest paths from the considered node to all other nodes in the network [Bavelas, 1950]. It is also normalized by the number of nodes using $n-1$, so that it can be compared for graphs with different number of nodes. We compute the closeness centrality $c_k^{central}$ of node k by:

$$c_k^{central} = \frac{n-1}{\sum_{l \neq k, l=1}^{n} p_{kl}^s} \tag{2.19}$$

and do not use it directly but use its conceptual idea to compute the current-flow closeness centrality.

Current-flow closeness centrality

Again, the idea of closeness centrality is extended by considering the resistance[1] of lines. We compute the current-flow closeness centrality by the following equation [Brandes and Fleischer, 2005]:

$$c_k^{flow} = \frac{n-1}{\sum_{t \neq s} v_{st}(s) - v_{st}(t)}, \tag{2.20}$$

where $v_{st}(s) - v_{st}(t)$ corresponds to the effective resistance and $v_{st}(s)$ is the potential from node s based on the current supply from node s to t. [2] The current-flow closeness centrality is also used to compare nodes and normalized by $\frac{2}{(n-1)(n-2)}$.

Clustering coefficient

The clustering coefficient is not a centrality measure but shows how connected the neighbors of a considered node are. A high clustering coefficient means that the neighbors of a node are well connected. For an unweighted graph the clustering coefficient Δ_k^c of node k is given by

$$\Delta_k^c = \frac{2 \times \text{number of triangles through node k}}{d_k(d_k-1)}, \tag{2.21}$$

[1] More information about the resistance is given in section 2.2.1

[2] In case of power grids, the potential is the voltage with the following relation: $v = ir$, where i denotes the current and r denotes the resistance. Detailed information can be found in section 2.2.1.

In case of weighted edges, we get the following equation [Onnela et al., 2005]:

$$\Delta_k^c = \frac{1}{d_k(d_k-1)} \sum_{l,m} (\hat{w}_{kl}\hat{w}_{km}\hat{w}_{lm})^{\frac{1}{3}}, \tag{2.22}$$

where the weights are normalized by the largest weight in the network: $\hat{w_{kl}} = \frac{w_{kl}}{max(w_{kl})}$.

2.2. Power grids

Power grids are interconnected networks delivering electricity and consist of producers, consumers, transformers and the connecting power lines. On the contrary to many other networks, one outstanding criterion of power grids are the underlying physical relations of the components. In case of power grids, the interaction of the vertices can be described physically and is not simply a property like the membership in a certain group which is often used to construct social networks. In order to compute those physical relations, fundamentals of electrical engineering are introduced. Afterwards we look at the computation of static and dynamic flows of power grids before considering the stability of power grids.

2.2.1. Power flow equations

The computation of power flows is often based on complex variables, so we introduce the complex voltage as well as complex current, before introducing the relation between them. The complex voltage **v** (potential difference) depends on the magnitude u and the phase angle θ:

$$\mathbf{v} = ue^{j\theta}, \tag{2.23}$$

the complex current $\boldsymbol{i}$ is similarly defined:

$$\mathbf{i} = ie^{j\theta}. \tag{2.24}$$

Both variables are related based on the law of Ohm which states: for an ideal conductor, the voltage is proportional to the current through it.

$$\mathbf{v} = \mathbf{zi}, \tag{2.25}$$

where **z** is the impedance. The impedance consists of the real part r (resistance) and the reactance x: $\mathbf{z} = r + jx$. The reciprocal of the impedance is the admittance:

$$\mathbf{y} = \frac{1}{\mathbf{z}} = g + jb, \tag{2.26}$$

where g is the conductance b the susceptance. For many applications the conductance is of lower magnitude than the susceptance and can be neglected. As with using complex variables, we can also relate the real variables such as: $v = ri$, where v is the real voltage, i the real current. The reciprocal of the resistance r is called the conductance.

Kirchhoff's laws on electrical circuits

In the 19th-century Kirchhoff described laws regarding electrical circuits that are later called the first and second law of Kirchoff. The first law states that the incoming and outgoing flows (the currents) must sum to zero at all vertices v, which can be written as:

$$\sum_j i_j = 0, \tag{2.27}$$

where we sum over the neighboring flows for all vertices. We define: Ingoing currents are positive ($i > 0$) and outgoing currents are negative ($i < 0$). The second law, also called Kirchhoff's voltage law states that for a closed circuit the directed sum of all voltages is zero:

$$\sum_{\text{vertices on a cycle}} v_j = 0. \tag{2.28}$$

2.2.2. Computation of static flow of power grids

Next, we want to compute the flows of a power grid which consists of multiple nodes. In a stationary state the flows do not change, so we call this a synchronized grid. However, such static conditions do not necessarily exist for each grid. To represent the flows of the entire grid, we formulate a matrix multiplication based on the laws by Ohm (eq. (2.25)). This formulation uses the Laplacian matrix $\boldsymbol{L}$, where the weights are given by the admittances of all power lines. The off-diagonal elements of $\boldsymbol{L}$ are given by the absolute value of the admittance and we assume $b >> g$, so $|\boldsymbol{y}| = b$:

$$L_{ij} = -b. \tag{2.29}$$

The diagonal elements are:

$$L_{ii} = \sum_{k=1, k\neq i}^{n} |b_{ik}|. \tag{2.30}$$

Using eqs. (2.25) and (2.26) we get the following system of equations written in matrix form:

$$I = j\boldsymbol{L}V, \tag{2.31}$$

with I, V being vectors with n entries and $\boldsymbol{L}$ is a matrix of dimension $n \times n$. The complex power $\boldsymbol{s}$ (vector) is defined by:

$$\boldsymbol{s} = V \odot I^{\star}, \tag{2.32}$$

where we multiply V and $I^{\star}$ element-wise using the Hadamard product and $I^{\star}$ denotes the complex conjugate of I. Next, we can separate the power in real and imaginary part by using the following relation:

$$\boldsymbol{s} = P + iQ, \tag{2.33}$$

to obtain equations for the real power P and imaginary power Q.

$$P = Re[\boldsymbol{s}] = Re[V \circ (\boldsymbol{L}V)^{\star}], \tag{2.34}$$

$$Q = Re[\boldsymbol{s}] = Im[V \circ (\boldsymbol{L}V^{\star})]. \tag{2.35}$$

We can interpret the powers as sources (producers) and sinks (consumers). Equations (2.34) and (2.35) give us a set of $2n$-equations with $4n$ unknowns. This means that we have two equations per node, but four unknowns. For some applications the powers are known. To give an example, power plants are mostly sources and factories consumers. So, when setting the powers we can reduce the unknowns by $2n$. This leaves $2n$ unknowns (θ and u) from eq. (2.23), so the system is solvable. We can further reduce the complexity by fixing u and then only use the real power (eq. (2.34)) to solve for θ.

2.2.3. Computation of dynamic flow of power grids

Since power grids are rarely in a stationary state due to changes in demand, production or technical failures, we need to investigate the dynamical behavior of power grids. When dynamically modeling power grids, we must characterize all components such as nodes and lines of the grid. In this thesis we focus on a simple approach by using the Kuramoto model to describe all nodes and only use one type of lines, where the admittance simply depends on the length of the line.

Kuramoto model with inertia

The Kuramoto model was originally introduced in 1975 by Kuramoto, republished in Kuramoto [2005] and is a widely used model for coupled oscillators. We apply the Kuramoto model to power grids, by modeling the components of power grids as oscillators. The Kuramoto model describes the frequency variations of the generators and consumers in the network using the instantaneous phases ϕ_i and frequency variations $\dot{\phi}$ and the rate of change $\ddot{\phi}$. We also use α as an numerical dumper and

get the following equation:

$$\ddot{\phi}_i = P_i - \alpha\dot{\phi}_i - \sum_j^n L_{ij} sin(\phi_i - \phi_j). \tag{2.36}$$

We often see the Kuramoto model using a capacity matrix containing the admittances only instead of the graph Laplacian $\boldsymbol{L}$. In this thesis, we only consider the susceptance and neglect the conductance. Using this setup we can do simulations of power grids which is a common approach used for example by Nitzbon et al. [2017], Schultz et al. [2014b]. Next, we introduce a concept to investigate the dynamic stability.

2.2.4. Dynamic stability of power grids

We can compute the dynamic stability of power grids by using basin stability introduced by Menck et al. [2013] which is a general concept for stability analysis of dynamical systems. Leng et al. [2016] enhance this approach for delayed dynamics. The idea of basin stability is to perturb the system from a stationary state ϕ^* and observe its reaction. If the system returns to ϕ^* for $t \to \infty$ it is considered basin-stable. All initial sets from which the system converges are called basin of attraction of ϕ^*. For our applications, we are not restricted to going back to the initial stationary state, but any feasible stationary state meets our demands.

2.3. Machine learning

Machine learning (ML) is one subfield of artificial intelligence. We distinguish machine learning concepts by their learning paradigms: supervised learning, unsupervised learning and reinforcement learning, even though reinforcement may also be considered a subtype of unsupervised learning. In this thesis, we focus on supervised learning, so we need a set of input and output data to use these data to train our model. Furthermore, we only use artificial neural networks (ANNs) which are a universal function approximators of ML. This section is structured as follows. We start by considering a single neuron and then describe the conceptual idea of ANNs. Afterwards we introduce components of ANNs, describe the training of ANNs and consider common architectures of ANNs. The remaining sections are about Graph Neural Networks and their application to power grids.

2.3.1. Setup of artificial neurons

For a simple setup we assume a single-input-neuron, shown in fig. 2.5. The input x is multiplied by the weight w, before adding the bias b and applying the activation function σ to obtain the output

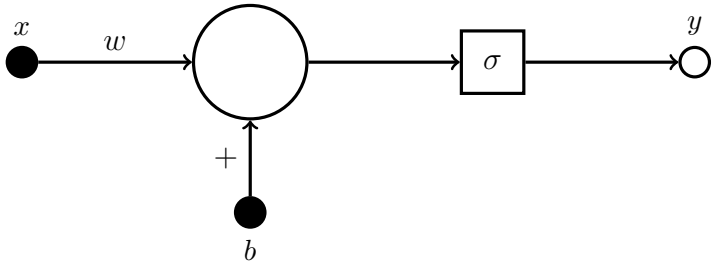


Figure 2.5.: Scheme of one single-neuron layer. The input x is multiplied by the weight w and after adding the bias b the activation function σ is applied to obtain the output y.

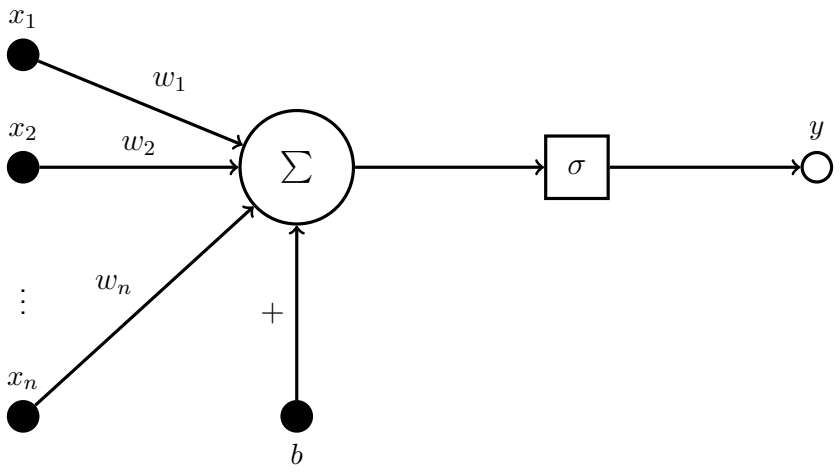


Figure 2.6.: Scheme of one multiple-input neuron with multiple inputs x_i and multiple weights w_i for i=1,...,n, so the sum of the result of the multiplication of the inputs and weights is computed before the bias is added and the activation function applied.

y. This approach can be extended to use a neuron with multiple inputs x_i (fig. 2.6). In this case we have a weight matrix $\boldsymbol{W}$ consisting of multiple weights w_i. Now we can stack many neurons together to form a neural network as shown in fig. 2.7, where we did not explicitly show the bias and activation function, because they take place within each neuron. We also did not label the weights, but the weights are attributed to all arrows. Except for the input layer, this network is fully connected which means that all neurons within one layer are connected to all neurons in the following layer. Later on, we see other ways of structuring ANNs depending on the type of problem.

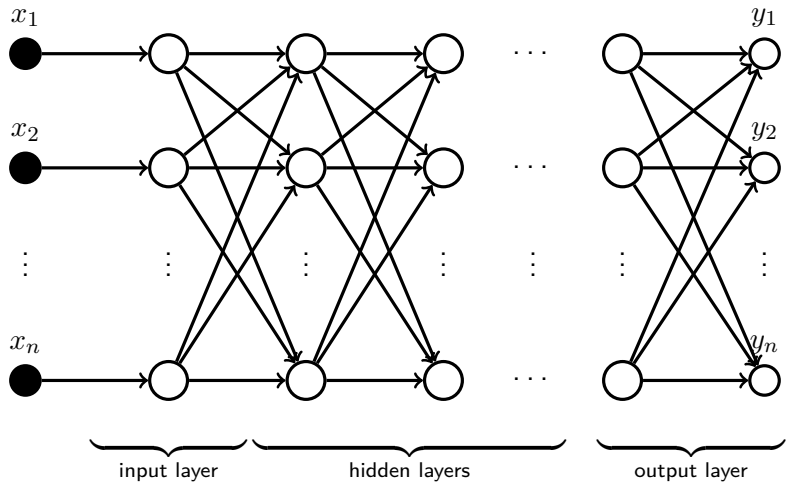


Figure 2.7.: Artificial neural network consisting of multiple layers with multiple in- and outputs.

2.3.2. Conceptual idea of neural networks

The idea of supervised learning is the approximation of latent and non-linear functions that represent the transformation between the input x and output y. So, the model is trained based on a given set of data to estimate the output y for a given input x correctly. The dataset including x and the target variables or labels[3] l must be previously known and needs to provide sufficient information for the training. Each input x is attributed one label l which is the true output that we try to predict. One pair of input is called a sample $\mathcal{S}$ such that:

$$\mathcal{S} = \{x, l\}. \tag{2.37}$$

and our dataset $\mathcal{D}$ consists of multiple samples. To get a good prediction of y based on x we must determine an appropriate set of the network parameters or alternatively called model parameters $\mathcal{P}$:

$$\mathcal{P} = \{\boldsymbol{W}, b\}. \tag{2.38}$$

A perfect model would give the true outputs for all inputs, so it should be applicable over the full domain. Due to the lack of alternatives, we can only investigate the outputs for our datasets, but we assume that the dataset covers the full domain, so we call a model a perfect model if y is always equal to l. Hence, we want to find $\mathcal{P}$ such that $y \approx l$. The computation of $\mathcal{P}$ is explained in section 2.3.4. In this section we focus on some general aspects regarding the training of ANNs.

An entire training process usually consists of multiple epochs, so we improve our model by consecutively updating $\mathcal{P}$ on the same data multiple times. When training ANNs we split our dataset in two (training and test) or three (training, validation and test) sets to train and evaluate our

[3]For the sake of brevity, the term labels often include target variables, so we talk about classification and regression-type problems at once.

model. It is important that there are no general differences in the data used for training or testing. Otherwise training might result in a model that predicts the training set well but is bad at predicting the test set. If a model memorizes the training data, but performs poorly on the validation set, it is called overfitted and such a model is said not to generalize well. On the contrary, if a model is not able to learn the relations given by the training set and performs poorly on the training set it is considered underfitted. Validation sets play an important role during training, which is discussed in section 2.3.4 and testing sets are used to evaluate the performance of trained models.

2.3.3. Components of neural networks

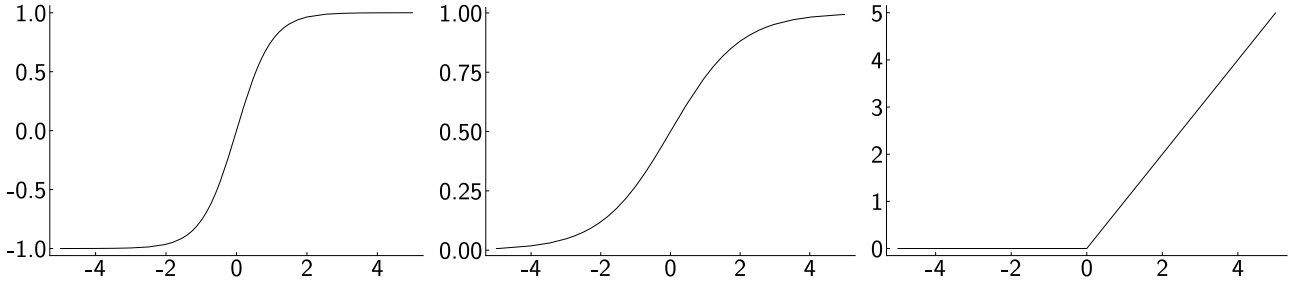
In this section we consider further components and methods that are commonly used in ANNs. We start by introducing activation functions.

Activation function

In general, activation functions are supposed to introduce non-linearities to ANNs, so that the resulting model can map non-linearities. Hence, we will focus on non-linear activation functions and introduce some of the common functions which are used in this thesis. The plots of the activation functions σ are given in fig. 2.8 and more information is given in table 2.1. The first shown activation functions $\tanh$ and sigmoid map curvature properties and are asymptotically bounded. The third activation function is the rectified linear unit (ReLU), which is widely used now. ReLUs were already used by Hahnloser et al. [2000] and Jarrett et al. [2009] investigate ReLUs among other activation functions and show its convincing performance. The general advantages of ReLUs are its cheap computation and fast convergence. According to Krizhevsky et al. [2012], it converges about six times faster in comparison to $\tanh$. Furthermore, the output can be a true zero, which means that a neuron is completely deactivated. Some alternatives as $\tanh$-activation can only approximate this state, see table 2.1. An output of zero however might result in the dying ReLU problem. Once neurons are deactivated due to a negative input, it is likely that those neurons will never be updated again, because their output is meaningless. Ways of avoiding this are described in appendix A.1.1. Further problems are very high output values since the output is not restricted. Restricting the output to a range between 0 and 1 in the output layer is one of the main motivations to apply a Sigmoid layer.

Dropout

To tackle the problem of overfitting dropout is introduced by Hinton et al. [2012]. The basic idea is to prevent complex co-adaptations of neurons, when features detectors are only helpful in combination



(a) hyperbolic tangent (b) sigmoid (c) ReLU

Figure 2.8.: Activation functions that introduce non-linearity to ANNs

name	equation	output range
hyperbolic tangent	$\tanh(x)$	(-1,1)
sigmoid	$\frac{1}{1+e^{-x}}$	(0,1)
ReLU	$\max(0, x)$	$[0, \infty)$

Table 2.1.: Activation functions

with other detectors. The same group pursues the ideas in Nitish Srivastava et al. [2014] and also patent the method with the assignee Google [Hinton et al., 2016]. Dropout means that neurons are randomly deactivated which result in a thinner network during training. At test phase dropout is not applied, so all neurons are active, and the weights are reduced to obtain results in the same range as the thinned network. The probability of presence during training can be freely chosen. [Nitish Srivastava et al., 2014] claim a probability of $.5$ works well for many problems, except for the input layer. For the input units less neurons should be deactivated.

Convolutional networks

Convolution Neural Networks (CNNs) are one of the key technologies to analyze visual images using ANNs. Convolutional layers aggregate spatial information of nodes (pixels) by evaluating the relations of pixels and its neighbors. For image analyses multiple convolutions are applied consecutively. Convolutions aim to extract information by applying learnable filter functions or kernels to the input data. To understand the procedure, we start by looking at the definition of convolutions. The convolution of f and g is given by:

$$[f * g](t) = \int_0^t f(\tau)g(t-\tau)d\tau. \tag{2.39}$$

0	0	0	0	0
0	2	5	4	0
0	1	3	6	0
0	4	1	7	0
0	0	0	0	0

(a) 3x3 input with padding

1	0	1
1	1	0
1	1	1

(b) 3x3 filter

6	17	18
11	22	22
7	12	11

(c) Result

Figure 2.9.: Padding and the result of a convolution of a padded input

In case of investigating pictures, we are usually interested in 2D-cases, so we need a 2D-convolution and discretize it to get the result of the convolution $\mathcal{C}$:

$$\mathcal{C}(x, y) = \sum_{m,n} = g(m, n) * f(x - m, y - n). \tag{2.40}$$

In our case f is the input data and g is our kernel function. So, our convolution can be expressed by the following equation:

$$\mathcal{C}(x, y) = \omega * f(x, y), \tag{2.41}$$

where ω is the kernel function. For CNNs one usually defines the kernel by a $k \times k$ matrix, where k is the filter size and all entries are learnable weights. To apply the kernel, one also chooses the stride which controls the movement of the filter. The number of strides is equal to the number of jumps per step. For example, a stride of 1 means moving the filter one pixel per step. To control the output size, we use padding by adding zeros around the borders. To give an example, we define weights (3×3) in fig. 2.9b and apply it to a 3×3 input given in fig. 2.9a within the borders. The resulting output is a 3×3 matrix given in fig. 2.9c.

It is common to apply several convolutions by using multiple kernels. In most cases kernels uses the same filter size per convolutional layer. The number of applied convolutions per layer is equal to the resulting number of channels c. In case of our example shown in fig. 2.9 applying c convolutions results in an output of size $c \times 3 \times 3$.

Pooling

Pooling is a concept for downsampling information. By applying pooling layers, the spatial dimension is reduced. There are different pooling algorithms like maximum pooling or average pooling. For average pooling we compute the average of the considered input, and maximum pooling simply means to consider the maximum value of the considered input $(k \times k)$. An example of maximum pooling is given in fig. 2.10.

4	2	**6**	1
1	3	2	0
5	**8**	1	3
7	4	**9**	2

(a) Input of pooling layer

4	6
8	9

(b) Output of pooling layer

Figure 2.10.: Example of maximum pooling

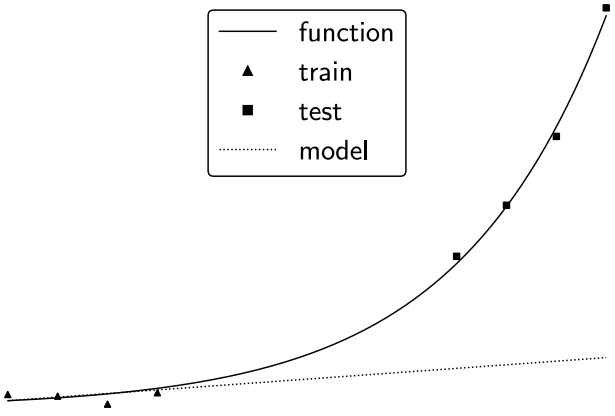


Figure 2.11.: Covariate shift: problem of shifted input distribution for training and testing

Batch normalization

Before explaining batch normalization we must note that a batch is simply a concatenation of samples. So, a batch of size 1 is simply a sample. Batch normalization is a method to increase the performance of ANNs by reducing covariate shift. Covariate shift is the problem of changed input distributions to a learning model. The problem of covariate shift is visually shown in fig. 2.11. For this simple example, we aim to find a model that predicts the function correctly based on the training data. The model is relatively accurate on the training data but is not able to give a good prediction for a shifted distribution used for testing.

In deep networks we constantly have covariate shift to the input of the deep layers due to changes of the weights and the bias in preceeding layers. We can also think of different ranges of features, where some are bounded to $\in [0, 1]$ and others have no bounds at all. To reduce the problem of change of input values we apply batch normalization introduced by Ioffe and Szegedy [2015]. The basic idea is to subtract the batch mean μ_B for each sample and divide this by the batch variance σ_B^2. Two more parameters γ, β are introduced that are learned. Based on the batch input x with

the batch size ($|\mathcal{B}|$), the algorithm can be written as:

$$\begin{aligned}
\mu_B &= \frac{1}{|\mathcal{B}|}\sum_{i=1}^{|\mathcal{B}|} x_n \\
\sigma_B^2 &= \frac{1}{|\mathcal{B}|}\sum_{i=1}^{|\mathcal{B}|}(x_i - \mu_B^2) \\
\hat{x_i} &= \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \\
y_i &= \gamma\hat{x_i} + \beta,
\end{aligned} \tag{2.42}$$

where y_i is the output of the batch layer for sample i and ϵ is purely added for numerical stability.

Normalization reduces the problem of the change of input values, because it decreases the amount of shifting of the output of each layer. Outputs of layers become more stable and deeper layers are less influenced by changes in preceding layers. The coupling of consecutive layers to changes in early layers is reduced, so that each layer can better learn individually. Due to the more stable input values of deep layers, the trained parameters of deep networks get more robust to changes in preceding layers. Batch normalization enables us to use higher learning rates and the model gets better at learning different inputs. Furthermore, there is a small regularization effect, because the normalization is applied to a batch and not the entire training dataset. We scale the batch based on the mean and variance of this batch and not the entire dataset, so some noise is introduced in this process. As with dropout (see section 2.3.3) the added noise reduces overfitting.

Residual blocks and dense nets

As networks get deeper training becomes more challenging. The challenge arises due to vanishing or exploding gradients that make training more difficult [Bengio et al., 1994]. To enable the training of deeper networks, Highway networks are introduced in an extended abstract by Srivastava et al. [2015b] and also in more detail in another paper [Srivastava et al., 2015a]. *Highway Networks* (HN), add highways to ANNs that pass information unimpeded across many layers. The highways are not simply skip connections, because gates control how much information flows across layers. The skip weights of the gates are learned for HN during training.

He et al. [2016] introduce residual blocks and on the contrary to HN, the shortcuts are parameter-free and data-independent. We start by considering the architecture of a residual block shown in fig. 2.12. As we can see the identity of x is added to the stacked layers without adding any parameters. This block is called a residual block, because the layers are trained to fit a residual mapping. The desired mapping is denoted by $\mathcal{H}(x)$ and we know from fig. 2.12: $\mathcal{H}(x) = \mathcal{R}(x) + x$. We can rearrange this to: $\mathcal{R}(x) = \mathcal{H}(x) - x$ and see that $\mathcal{R}(x)$ is now a residue and He et al.

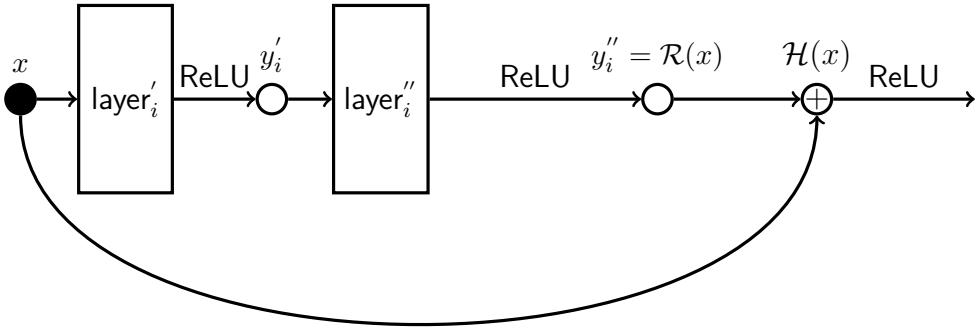


Figure 2.12.: Residual block where $'$ and $''$ denote internal numbering and $\mathcal{H}(x)$ is the desired mapping

[2016] hypothesize that it is easier to optimize the residual than the original mapping. One of the advantages of this definition is that the training algorithm with backpropagation does not have to be modified.

Another approach is called DenseNets introduced by Huang et al. [2017]. DenseNets use an architecture where each layer is connected to all other layers which leads to more connections. For a dense convolutional network (DenseNet) with L layers a DenseNet has $\frac{L(L+1)}{2}$ connections, while a regular network would have only L connections. Since the outputs of all preceding layers are used and the output of each layer is used for all subsequent layers, the problem of vanishing gradients is alleviated. Furthermore, training of DensNets can be faster than training of ResNets, because DenseNets can reduce the number of parameters. On the contrary to residual block, DenseNets do not sum the results of preceding layers, but concatenate them.

2.3.4. Training of neural networks

Training of an ANN is an optimization problem. Before considering the training process in detail, loss functions are introduced.

Loss function

Loss functions are used to compare the output y to the label l and set the basis for the training of $\mathcal{P}$. So, we define a loss function $f_{\mathcal{L}}$ which basically computes the deviation of our model output y and l. Depending on the problem different loss functions can be used. The effects of different loss functions are investigated in Janocha and Czarnecki [2017], but we only focus on its general

name	equation	meaning
L1 loss	$\frac{1}{\|\mathcal{B}\|}\|\|l-y\|\|_1$	mean absolute error
L2 loss	$\frac{1}{\|\mathcal{B}\|}\|\|l-y\|\|_2^2$	mean squared error

Figure 2.13.: Loss functions which are often used for regression problems, where $|\mathcal{B}|$ denotes the size of the batch

human	dog	cat
0.7	0.1	0.2
0.5	**0.4**	0.1
0.9	0.0	0.1

(a) Outputs (probabilities) of an ANN

human	dog	cat
1	0	0
0	1	0
1	0	0

(b) Labels of dataset

LLN
.16
.40
.05

(c) Negative Log Likelihood

Table 2.2.: multiclass classification problem

application in this thesis. Figure 2.13 shows the loss functions computing the mean absolute error and mean squared error. Both are straight-forward and especially useful for regression problems.

For classification problems other loss functions have benefits. We distinguish between multiple and binary classification problems. To construct an example of a binary classification problem we assume to have a dataset with pictures showing either a dog or a human. Now, we want to find a model that predicts whether there is a dog or a human pictured. We could extend this example by adding pictures of cats meaning that our model is supposed to predict three categories. So, we setup an ANN with three outputs y_1, y_2, y_3, where each of them contains a probability that the item belongs to a class. Since only one of the categories can be true for our dataset, we normalize the output such that: $\sum_i y_i = 1$, which can be done by a softmax-layer. To understand the loss function, we generate an arbitrary dataset shown in table 2.2a. In all three cases (table 2.2a), the ANN predicts that a human is in the picture, however we can clearly see that the confidence of the model is different, because the probabilities vary. By computing the negative Log-Likelihood (NLL) we get a measure of the uncertainty:

$$NLL = -l_i * log(y_i). \tag{2.43}$$

Using the model output (table 2.2a) and the labels (table 2.2b), we can compute NLL, which is shown in table 2.2c. Clearly, NLL increases with higher uncertainty, so we can force the model to increase its certainty. This loss function is usually called Cross-Entropy-Loss, and a Binary-Cross-Entropy-Loss can simply be derived from the Cross-Entropy-Loss and is applicable when having only two classes.

Training process

To train ANNs, mathematical optimizations are conducted. Our objective is the loss function which we aim to minimize by optimizing $\mathcal{P}$. The multidimensional problem can be expressed by:

$$\min_{\mathcal{P}} f_{\mathcal{L}}. \tag{2.44}$$

Stochastic gradient descent (SGD) is a common algorithm of determining $\mathcal{P}$. To describe SGD we start by considering gradient descent. Gradient descent is an iterative method finding the local minimum of a function by using its gradient. Since we are interested in the minimum, we need to go in the opposite direction of the gradient: $-\nabla f_{\mathcal{L}}$ We update the parameters using the following scheme:

$$\mathcal{P}_{new} = \mathcal{P}_{old} - \eta \nabla f_{\mathcal{L}}, \tag{2.45}$$

where η is the step size and defines how far we go in the direction given by the gradient. As for many other optimization problems, the step size is crucial for converging. For training ANNs the step size is defined by the learning rate, which itself is a parameter chosen by the user. Choosing the learning rate is discussed later in this section (section 2.3.4). To follow the proposed procedure, we have to compute the gradients and in case of ANNs, backpropagation is widely used to obtain the gradients. Backpropagation was firstly introduced by Werbos [1974] in his PhD thesis (Philosophy) in behavioral sciences and later on rediscovered by Rumelhart et al. [1986]. Backpropagation is applied by using the loss function (section 2.3.4), to compute the contribution of all parameters to the losses by going backwards through the network layer by layer. The chain rule is used to obtain all individual contributions $\frac{\partial f_{\mathcal{L}}}{\partial w_i}$.

The optimization process can be improved by changing the numbers of samples used for each update. To define the number of used samples for each update, we use the batch size. When setting the batch size equal to the number of samples in the training, the entire training set is used for each update. This is called batch-gradient descent or vanilla gradient descent. In this case we compute an averaged loss function such as: $\nabla f_{\mathcal{L},\mathcal{B}}$:

$$\nabla f_{\mathcal{L},\mathcal{B}} = \frac{1}{|\mathcal{B}|} \sum_{i}^{|\mathcal{B}|} \nabla f_{\mathcal{L},i}, \tag{2.46}$$

where $|\mathcal{B}|$ denotes the batch size. However, it is expensive to compute the gradients for all samples before doing an update and memory requirements can also limit the maximum batch size. Hence, the batch size may be reduced. In case of using the smallest possible batch size of 1, $\mathcal{P}$ is updated based on each sample. The batches consist of multiple samples that are arbitrarily chosen. This motivates the term stochastic gradient descent (SGD), introduced by Bottou [1998], who calls SGD *online gradient descent*. In general, we can say that large batch sizes generalize better and stabilize

the optimization process, but are only feasible for small datasets, because we have to go through the entire dataset many times. Small batch sizes generalize less, but the added noise can help to converge to global minima by avoiding to converge to bad local minima. Dekel et al. [2012] show the application of use mini-batches for distributed prediction to efficiently obtain the results using multiple processors. Several modifications are introduced to improve the mini-batch based approach to increase its performance, memory cost or both for example by Li et al. [2014] and Wang et al. [2017].

Learning rate

Choosing an appropriate learning rate is essential, so we look at the basic ideas of setting the learning rate. The learning rate should be chosen to get a substantial reduction of the objective. If the learning rate is too low, we might not achieve significant improvements or we could get stuck at a local minimum. If the learning rate is too large, the loss might increase, because we overshoot and do not converge. A lot of research is done to develop methods of defining the learning rate. Several approaches rely on varying learning rates, so the learning rate is adapted throughout the training. The initial learning rate is often called the most important hyperparameter of training ANNs [Bengio, 2012]. Intuition suggest that decaying the learning rate helps to: a) escaping potential local minima at the beginning and b) converging to minimum without oscillations at the end. You et al. [2019] assume that this common knowledge might be insufficient and explain that large learning rates avoid memorizing noisy data at the beginning and complex patterns are learned while decaying the learning rate. Decaying the learning rate is set by schedulers during training. So when using a simple setup the initial learning rate is simply decayed with increasing step size, e.g. exponentially. There are also more advanced approaches that include validation information for choosing the learning rate. For example, the learning rate is only decreased when the model stops improving on the validation set, because it reaches a plateau. Another widely used approach is Adam [Kingma and Ba, 2015] which has individual learning rats for each parameter. The term Adam is derived from adaptive moment estimation. More concepts of choosing the learning rate are given in appendix A.1.2.

Weight decay

Weight decay is a method to prevent overfitting and used for regularization purposes, because large weights are penalized. The concept is explained in more detail by Krogh and Hertz [1992]. The idea is reducing unneeded complexity and punishing large weights by adding the sum of the squares of all the weights of the model to the loss function and obtaining a new loss:

$$f_{\mathcal{L},new} = f_{\mathcal{L}} + \omega \frac{1}{2} \sum_{i}^{\tilde{\mathcal{P}}} \mathcal{P}_i^2, \tag{2.47}$$

where ω denotes a coefficient for weight decay and $\tilde{\mathcal{P}}$ denotes the number of parameters. When applying this to SGD, we obtain the following update scheme:

$$\mathcal{P}_{new} = \mathcal{P}_{old} - \eta \nabla f_{\mathcal{L}} - \eta \omega \mathcal{P}_{old}. \tag{2.48}$$

Nowadays concepts of adaptive weight decay are introduced, e.g. by Nakamura and Hong [2019].

Momentum

Another relevant hyperparameter is the momentum, explained by Qian [1999]. Momentum is used to avoid converging towards local minima and stabilizing the optimization process. When not using momentum and updating using as shown in eq. (2.45), we get the following scheme for update step t:

$$\Delta \mathcal{P}_t = -\eta \nabla f_{\mathcal{L}}. \tag{2.49}$$

When using momentum, we add the moving average of our gradients and obtain:

$$\Delta \mathcal{P}_t = -\eta \nabla f_{\mathcal{L}} + p \Delta \mathcal{P}_{t-1}, \tag{2.50}$$

where p is the momentum parameter. So the update depends on the gradient as well as the change of the previous update. Using this update, the oscillation around a narrow valley on the optimization path can be decreased[Qian, 1999]. For some applications a different implementation is used, where the learning rate is multiplied by the previous parameter change as well:

$$\Delta \mathcal{P}_t = \eta(-\nabla f_{\mathcal{L}} + p \Delta \mathcal{P}_{t-1}). \tag{2.51}$$

More concepts are given in appendix A.1.3.

Convergence of training

When training ANNs, one hopes to find a global minimum or at least a very good local minimum and is afraid of ending up at a saddle point or a bad local minimum. Investigations by Goodfellow et al. [2014] show that many ANNs never encounter any significant obstacles during training, but Liu et al. [2019] show that bad global minima exist. They use the term bad global minima to describe cases, when the train accuracy is very high, but the test accuracy low due to missing generality. By manipulating the initialization parameters, ANNs can converge towards bad local minima during training. Furthermore, Kawaguchi and Kaelbling [2019] suggest that all suboptimal local minima can be eliminated, by adding one neuron per output unit to any deep ANNs. So increasing the complexity of ANNs helps converging to a global minimum and apparently global optima or at least sophisticated local minima are reached often.

2.3.5. Architectures of convolutional neural networks

CNNs are advanced ANNs that combine many of the previously introduced components and methods to tackle complex problems. CNNs are widely used for image or voice recognition. In this section we introduce some of the mostly used architectures.

Alexnet

Krizhevsky et al. [2012] introduce a CNN-architecture which is commonly referred to by AlexNet. In 2012 AlexNet was one of the largest convolutional networks, which contains eight learned layers with five convolutional and three fully connected layers. At that time $\tanh$ is commonly used as activation functions, but AlexNet uses ReLU instead. To prevent overfitting, dropout is used. In 2012 AlexNet won the classification task ImageNet Large Scale Visual Recognition Challenge (ILSVRC) by more than 10 % in comparison to its competitors and drew a lot of attention to deep ANNs[4] [ILS, 2012].

ResNet

ResNets rely on residual nets and their architecture enable the usage of larger number of layers as described in section 2.3.3 [He et al., 2016]. In 2015 ResNet won the first place on ILSVRC (object detection) with 152 layers [ILS, 2015]. With ResNets, it is also possible to train networks with more than 1000 layers.

2.3.6. Graph neural networks

Graph neural networks (GNNs) are a special type of ANNs, because they enable to directly provide graph structure represented by vertices and edges (see eq. (2.3)) to ANNs without making any simplifications. This can be crucial for investigating graph-structured data. GNN solve different tasks such as graph-level classification, node classification, edge classification or any of the mentioned classification problems as regression problems. GNN are used for a variety of applications including social or citation networks, traffic prediction and chemistry identifying molecular structures. The key aspect of GNNs is the aggregation of spatial information from the graph-structure using convolutions. This section is structured as follows. We start by a small overview of the development of GNNs, before introducing two general types of convolutions and at the end the application of GNNs to power grids is considered.

[4] Deep neural networks are ANNs with multiple hidden layers

Development of Graph Neural Networks

There are different structures and types of GNNs that are covered by multiple review papers in more detail [Wu et al., 2019b, Zhang et al., 2019, Zhou et al., 2018, Bronstein et al., 2017]. For this thesis we often use the notation given by the first. The development of GNNs is based on different approaches. Gori et al. [2005] extend recurrent neural networks to apply ANNs directly on graph structures. The information is passed from neighbor to neighbor until a fixpoint is reached. Li et al. [2016] introduce Gated Graph Neural Networks with gated recurrent units and set a fixed number of steps instead of waiting until a fixpoint is reached. The two approaches from recurrent networks and convolutional networks are combined and written in a generalized form by Gilmer et al. [2017]. For our application however, we focus on convolutions, since they set the basis for our investigations. The convolutions are categorized by spectral-based and spatial-based methods and we begin by considering the first.

2.3.7. Spectral-based convolutions

Bruna et al. [2014] introduce a spectral-based approach for convolutions and Defferrard et al. [2016] formulate convolutions on graphs that we will use to explain the basic idea of spectral-based GNN. The term *spectral-based* is used, because this type of GNN uses the spectrum of the graph Laplacian $\boldsymbol{L}$ (see section 2.1.1). We can normalize the graph Laplacian so that the normalized graph Laplacian $\tilde{\boldsymbol{L}}$ is symmetric positive semidefinite:

$$\tilde{\boldsymbol{L}} = \boldsymbol{I} - \boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{A}\boldsymbol{D}^{-\frac{1}{2}} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^T, \tag{2.52}$$

where $\boldsymbol{I}$ is an identity matrix, $\boldsymbol{U}$ is an eigenvector matrix and $\boldsymbol{\Lambda}$ a diagonal matrix of eigenvalues. Since spectral convolutions use tools of graph signal processing such as the filters explained in Shuman et al. [2013], we introduce the term signal X which are scalar node features, so $X \in \mathbb{R}^n$ is a vector. The convolutions are applied in the Fourier domain, so we look at the graph Fourier transform of a signal X, which is given by $\hat{X} = \mathscr{F}(X) = \boldsymbol{U}^T X$ and the inverse is defined as: $X = \boldsymbol{U}\hat{X}$. The convolutional operator on graph $*_{\mathcal{G}}$ is defined in the Fourier domain with a filter $G \in \mathbb{R}^n$ such that:

$$X *_{\mathcal{G}} G = \mathscr{F}^{-1}(\mathscr{F}(X) \odot \mathscr{F}(G)) = \boldsymbol{U}(\boldsymbol{U}^T X \odot \boldsymbol{U}^T G)), \tag{2.53}$$

where $\odot$ is the Hadamard product. Hence spectral convolutions are basically the multiplication of a filter and the signal in the Fourier space. When specifying the filter G to $G_\Theta = diag(\boldsymbol{U}^T X)$ we obtain:

$$X *_{\mathcal{G}} G_\Theta = \boldsymbol{U} G_\Theta \boldsymbol{U}^T X, \tag{2.54}$$

which is the general setup of spectral-based convolutions. Different filters G_Θ are used for spectral graph convolutions. For $G_\Theta = \Theta_{i,j}^k$ the spectral graph convolutional layer for multiple channels is given by [Wu et al., 2019b]:

$$\boldsymbol{H}_{:,j}^{(k)} = \sigma(\sum_{i=1}^{(c_{k-1})} \boldsymbol{U}\boldsymbol{\Theta}_{i,j}^{(k-1)}\boldsymbol{U}^T\boldsymbol{H}_{:,i}^{(k-1)}) \quad (j = 1, 2, \ldots, c_k), \tag{2.55}$$

where k is the layer index, $\boldsymbol{\Theta}_{i,j}^{(k-1)}$ is a diagonal matrix with learnable parameters, $H^{(k-1)}$ the input signal, c_{k-1} the number of input channels and c_k the number of output channels to get the result $\boldsymbol{H}_{:,j}^{(k)}$. Defferrard et al. [2016] introduce the ChebNets, which use the Chebyshev polynomials for the filter function G_Θ and avoid explicitly using the Graph Fourier basis. Kipf and Welling [2016] use a first-order approximation of ChebNet to reduce computational costs and further simplify the setup. The filter operation by Defferrard et al. [2016] is given by:

$$X * G_{\theta'} \approx \sum_{k=0}^{k_K} \theta'_k T_k(\tilde{\boldsymbol{\Lambda}}), \tag{2.56}$$

where θ is a vector of parameters which contains Chebyshev parameters in this case and $\tilde{\boldsymbol{\Lambda}} = \frac{2}{\lambda_{max}} - \boldsymbol{I}$, where λ are the eigenvalues of $\tilde{\boldsymbol{L}}$ and $T_k(X) = 2XT_{k-1}(X) - T_{k-2}(X)$, with $T_0 = 0$ and $T_1(X) = X$ and k_K denotes the order of approximation. Furthermore, they propose to set $\lambda_{max} \approx 2$. Additionally, Kipf and Welling [2016] propose $k_K = 1$ and we obtain:

$$X * G_{\theta'} \approx \theta'_0 X + \theta'_1(\boldsymbol{L} - \boldsymbol{I})X = \theta'_0 X - \theta'_1 \boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{A}\boldsymbol{D}^{-\frac{1}{2}}X. \tag{2.57}$$

To avoid overfitting the number of parameters is reduced: $\theta = \theta'_0 = -\theta'_1$, so we have θ only and get:

$$X * G_\Theta \approx \theta(\boldsymbol{I} + \boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{A}\boldsymbol{D}^{-\frac{1}{2}})X \tag{2.58}$$

To increase the numerical stability the following renormalization is applied: $\tilde{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}$ and $\tilde{D_{ii}} = \sum_j \tilde{A_{ij}}$, such that: $\boldsymbol{I} + \boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{A}\boldsymbol{D}^{-\frac{1}{2}} \rightarrow \overline{\boldsymbol{A}} = \tilde{\boldsymbol{D}}^{-\frac{1}{2}}\tilde{\boldsymbol{A}}\tilde{\boldsymbol{D}}^{-\frac{1}{2}}$. When using multiple channels this can be extended to:

$$\boldsymbol{H} = \sigma(\overline{\boldsymbol{A}}\boldsymbol{X}\boldsymbol{\Theta}) \tag{2.59}$$

where $\boldsymbol{\Theta}$ contains multiple θ. This convolutional layer is referred to by Graph Convolutional Network (GCN) [Kipf and Welling, 2016], which might be misleading, because it is only one type of filters that is used for GNN. One big disadvantage of spectral-based graph convolutions are its dependency on the eigenbasis of the graph, so changing the topology changes the eigenbasis and hence the learned parameters have to be trained again, so spectral-based methods are limited to constant domains.

2.3.8. Spatial-based convolutions

Spatial-based convolutions use the spatial relations of nodes and its neighbors and are on the contrary to spectral-based methods not limited to one topology, so they attract more research attention recently. In 2009 Micheli [2009] already proposes to aggregate information of a node by summing up the neighboring information. Atwood and Towsley [2016] present diffusion-convolutional neural networks (DCNNs) which are somehow similar to recurrent neural networks. DCNNs rely on the idea that information is diffused through the network and each node simultaneously receives information from the past and its neighbors using convolutions. The diffusion process is independent of the node indexing and the parameters of DCNNs are determined with respect to search depth and not their position in the grid. The computation of the diffusion layer is given by:

$$\boldsymbol{H}^{(k)} = \sigma(\boldsymbol{W}^k \odot \boldsymbol{P}^k \boldsymbol{X}), \tag{2.60}$$

where $\boldsymbol{P}$ is a degree-normalized transition tensor: $\boldsymbol{P} = \boldsymbol{D}^{-1}\boldsymbol{A}$, and $\boldsymbol{X} \in \mathbb{R}^{n \times n_f}$ is a matrix containing the features of the nodes. We concatenate multiple $\boldsymbol{H}^{(0)}$, $\boldsymbol{H}^{(1)}$, $\boldsymbol{H}^{(...)}$, $\boldsymbol{H}^{(k)}$ and use a dense layer to connect the result to the node outputs Y. Instead of concatenating, Li et al. [2018] sum over the resulting $\boldsymbol{H}^{(k)}$. One of the challenges is that local information can be represented quite accurately, but global information is poorly represented. So Zhuang and Ma [2018] invent dual graph convolutional network to consider local and global properties at the same time.

Before introducing multiple new filters, we consider GCN [Kipf and Welling, 2016] again, because GCN can also be categorized as a spatial-based approach and many of the following introduced methods found on GCN. GCN can be interpreted spatially, because the information is aggregated from the direct neighborhood of the node and the computation of the spectrum of $\boldsymbol{L}$ is not required. Since GCN can also be derived and motivated using a spectral-based, GCN combines spectral-based and spatial-based approaches. Several methods rely on GCN and improve it, as we see when considering some of the recently introduced filter operations. In this section we focus on those types of convolution that are also used later on, while additional filter operations are explained in appendix A.2.

Simple Graph Convolution

Wu et al. [2019a] simplify GCN by reducing the number of nonlinearities of GNN by using a final softmax layer as the only nonlinear activation function. They hypothesize that the success of GNN relies on the local averaging and that the nonlinearity between GCN layers is not critical. Equally to using multiple layers, we multiply $\overline{\boldsymbol{A}}$ with itself for the desired number of times and combine all

weight matrices in to one weight matrix leading to the Simple Graph convolution:

$$\boldsymbol{H} = \text{softmax}\left(\overline{\boldsymbol{A}}^{i}\boldsymbol{X}\boldsymbol{\Theta}\right). \tag{2.61}$$

where we can interpret i as the number of convolutional layers using GCN, but in case of simple graph convolutions we simply combine this in one layer by using the i-th power of the renormalized adjacency matrix. Their results are comparable to other GNN which justifies their approach. Models based on simple graph convolutions are called SGNets in this thesis.

Topology Adaptive Graph-Convolution

The name *Topology adaptive Graph-convolution* (TAG) is motivated by the varying topologies of graphs on the contrary to approaches for fixed graphs, but this is the case for all spatial-based approaches [Du et al., 2017]. The basic idea of TAG is to consider larger surroundings of a node by using different exponentials of $\overline{\boldsymbol{A}}$ and multiple fixed-size learnable filters per layer. By using $\overline{\boldsymbol{A}}^2$ for example, the second nearest neighbors are also considered within one layer. So, TAG is able to leverage information at a farther distance than GCN for example. The output of the convolutional layer is given by:

$$\boldsymbol{H}^{(k)} = \sum_{z=0}^{Z} \boldsymbol{D}^{-\frac{1}{2}}\overline{\boldsymbol{A}}^{z}\boldsymbol{D}^{\frac{1}{2}}\boldsymbol{H}^{(k-1)}\boldsymbol{\Theta}_{z}, \tag{2.62}$$

where Z is the number of hops and the authors suggest using $Z = 2$.

Convolution using Arma-filters

Auto-regressive moving average (ARMA)-filters combine several aspects in one convolutional layer. Each Arma-layer consists of multiple internal layers as well as several stacks. By using multiple internal layers information can be aggregated across further distances. To apply convolutions using Arma-filters [Bianchi et al., 2019], we use a renormalized $\overline{\boldsymbol{L}}$ based on $\tilde{\boldsymbol{L}}$ such that:

$$\overline{\boldsymbol{L}} = \frac{2\tilde{\boldsymbol{L}}}{\lambda_{max}} - \boldsymbol{I}. \tag{2.63}$$

The convolution is given by:

$$\boldsymbol{X}_{s}^{(k)} = \sigma\left(\overline{\boldsymbol{L}}\boldsymbol{X}^{(k-1)}\boldsymbol{W}_{s}^{(k-1)} + \boldsymbol{X}^{(0)}\boldsymbol{V}_{s}^{(k-1)}\right), \tag{2.64}$$

where $\boldsymbol{V}$ are trainable parameters, that are always applied to the initial input $\boldsymbol{X}$, so we can consider this as one approach of using skip connections and s denotes the number of the stack and k the internal layer. For each stack and starting from the second layer the weights $\boldsymbol{W}$ and $\boldsymbol{V}$ are shared

which reduces the complexity. So, we can state that $\boldsymbol{W}_s^{(k)} = \boldsymbol{W}_s^{(k+1)}\ \forall k > 1$ and the same for $\boldsymbol{V} : \boldsymbol{V}_s^{(k)} = \boldsymbol{V}_s^{(k+1)}\ \forall k > 1$. For $k > 1$ we also use dropout, and this is the reason, why the weights are not shared with the first layer, because we do not want to use dropout in the initial layer. At the end, we combine S stacks to obtain the output:

$$\boldsymbol{H}^{(k)} = \boldsymbol{X}^{(k)} = \frac{1}{S}\sum_{s=1}^{S} \boldsymbol{X}_s^{(k-1)}. \tag{2.65}$$

There are no exponentials of $\overline{\boldsymbol{L}}$, so $\overline{\boldsymbol{L}}$ remains sparse which saves computational effort. Since skip connections are used within one Arma-convolution, it is also possible to stack multiple internal layers together without ending up with too much noise in the last internal layers. The convolution operation is also permutation invariant to renumbering of nodes.

Concluding remarks

It is also worthwhile to mention that many methods are developed simultaneously or at least multiple overlapping occurs during development time. Hence it is possible that some methods are only special cases of one another without being mentioned in this thesis. For example, Monti et al. [2017] set up a framework which can cover GCN by Kipf and Welling [2016], DCNNs by Atwood and Towsley [2016] and Graph Attention Network (GAT)[5] by Veličković et al. [2018] and perhaps other methods as well.

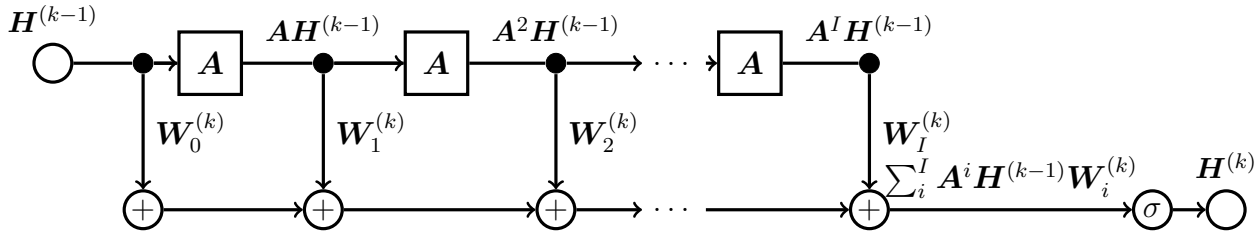
2.3.9. Application of graph neural networks to power grids

Recently, GNN are also applied to power grids. In 2019 three publications by different groups are about investigations of power grids using GNN [Owerko et al., 2019, Kim et al., 2019, Donon et al., 2019]. They all focus on static problems such as predicting optimal flow distributions under different circumstances and we briefly introduce their work.

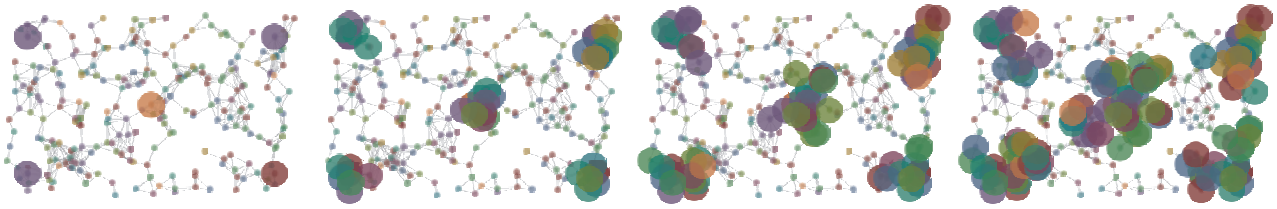
Kim et al. [2019] predict the optimal load-shedding that prevents transmission lines from being overloaded under line contingency. They compare GNN to a fully connected ANN. Their GNN consists of two convolutional layers based on GCN (see eq. (2.59)) and a third fully connected layer. The GCN outperforms the GNN by orders of magnitude. For a grid with 118 busses, the root-mean-squared error (RMSE) for the ANN is ≈ 6.58 and for GCN ≈ 0.029.

Owerko et al. [2019] use GNN to find the optimal power that generators within the must to produce to satisfy a given demand. They also use a spatial based approach and they do not only consider direct neighbors within each convolutional layer, but also neighbors of higher orders. The order

[5] GAT are described in appendix A.2



(a) Scheme of convolutional layer used by Owerko et al. [2019], picture inspired by Owerko et al. [2019]



(b) Picture by Owerko et al. [2019] showing the considered nodes for different number of hops, from zero hops on the left to three hops on the right

Figure 2.14.: Pictures showing the setup the GNN by Owerko et al. [2019] to analyze the power flow

describes the number of hops needed to go there by using powers of $\boldsymbol{A}$, such as $\boldsymbol{A}^i$, where i denotes the number of hops. By applying multiple hops I, the output of the hidden layer is given by:

$$\boldsymbol{H}^{(k)} = \sigma\left(\sum_{i=0}^{I} \boldsymbol{A}^i \boldsymbol{H}^{(k-1)} \boldsymbol{W}_i\right). \tag{2.66}$$

The full scheme is shown at the top in fig. 2.14a. Depending on the dimensions of $\boldsymbol{W}$, the number of node features can be chosen. For the first layer, we take the input node features $\boldsymbol{X}$, so $\boldsymbol{H}^{(0)} = \boldsymbol{X}$. Owerko et al. [2019] also publish a picture, where we can see how the different hops act and how more information is spread through the network by each hop, as we see in fig. 2.14b.

Donon et al. [2019] interpret power grids as graphs differently. Instead of representing power lines as edges, the power lines are vertices and power nodes as edges. The line impedances are all equal and the goal is to predict the flow through the lines. The lines are represented by bipolar lines and to distinguish its direction, each line has an origin and an extremity, and this information is encoded by two adjacency matrices to represent the network topology. A third matrix, the injections adjacency matrix, encodes the way injections (productions and consumptions) are connected to lines. The computation can be split in three steps: 1) embedding, 2) propagation and 3) decoding. The first step embeds the initial information and sends this information to the transition lines. In the propagation step, the states of all lines are updates based on the information from its neighbors. The third step simply decodes the information from the embedded space to the output space using the same function for all nodes and no information is exchanged between any nodes. All steps are conducted by matrix multiplications. Their *graph neural solver* is about twice as fast as proprietary

load-flow solver, which shows one of the advantages of using GNNs.

3. Methodology

In this chapter the methodology to reproduce the obtained results is given. The chapter is split in three main sections. Firstly, we look at the methods to generate the datasets. Secondly, the properties of the different datasets are presented. Thirdly, we consider the application of ANNs to the generated data.

3.1. Data generation: dynamic stability of power grids

This section covers four main topics. At first we look at the generation of power grids. Afterwards we investigate the ability to synchronize, before looking at the computation of dynamic stability. Information about the implementation and software is presented at the end of this section.

3.1.1. Generation of power grids

Power grids have distinct properties regarding the network topology and parameters of its components. We start by generating representative topologies, before we parametrize the lines and nodes. For our application we assume that all vertices have a fixed position in a two-dimensional (2D) space, so we use the package EmbeddedGraphs [Emb] to deal with graphs.

Grid generation using Synthetic Networks

To obtain the topology of our power grids, we use the tool *Synthetic Networks* available on GitHub [Syn] which is published by Schultz et al. [2014a]. The parameters from table 3.1 are used to generate the grids for our investigations and Euclidean distance is used as spatial distance function.

Ordering of the nodes

To increase the comparability of different grids, the nodes of the grids are ordered. By sorting the nodes with decreasing current-flow betweenness (see section 2.1.2), the adjacency matrix is independent of different ways of labeling the same grid. The ordering of the nodes is also a method

parameter	value	description
n0	1	initial number of nodes
p	.1	probability of constructing an additional redundancy line attached at each node
q	.32	probability of constructing a further redundancy line between existing nodes in each growth step
r	1.0	exponent for the cost-vs-redundancy trade-off
s	0.0	probability for splitting an existing line in each growth step

Table 3.1.: Parameters for grid generation using the tool *Synthetic Networks* Syn

of generalizing adjacency matrices, since similar networks are more likely to have similar adjacency matrices. In case of equal current-flow-betweenness nodes are ordered based on the grid generation process of the package Synthetic Networks. So, the previously added nodes during the grid generation go first.

Line properties

For our investigations we assume that all lines are equally made and that the admittance purely depends on the length of the line. So, the weights of the edges are computed based on the distance of two connected nodes, whose locations are drawn uniformly at random from the unit square. We use those weights and scale them with a mean value of 6.0 which corresponds to a transmission line length of roughly 200 km in the parametrization of eq. (2.36). Hence the weighted Laplacian matrix is generated by using the following scheme for the admittances $\boldsymbol{y}$ based on the adjacency matrix $\boldsymbol{A}$, for which the weights only depend on the distances of the nodes:

$$\begin{aligned} \tilde{\boldsymbol{y}}_{ij} &= \begin{cases} \frac{1}{a_{ij}} & \text{if } a_{ij} \neq 0 \\ 0 & \text{if } a_{ij} = 0 \end{cases} \\ \boldsymbol{y}_{ij} &= \frac{6}{\overline{\tilde{\boldsymbol{y}}_{ij}}} \times \tilde{\boldsymbol{y}}_{ij} \end{aligned} \tag{3.1}$$

where $\overline{\tilde{\boldsymbol{y}}_{ij}}$ is the average of all non-zero elements of $\boldsymbol{L}$. Again we assume that the conductance is much smaller than the susceptance: $g << b$, so $\boldsymbol{y}$=b and compute $\boldsymbol{L}$ according to eqs. (2.29) and (2.30).

Static node properties

We only consider two types of nodes: producers and consumers or sources and sinks, but mathematically both can be described by one type of node. Producers inject positive and consumers negative amount of power P_i. Two different methods of attributing P_i are used. Both have in common that

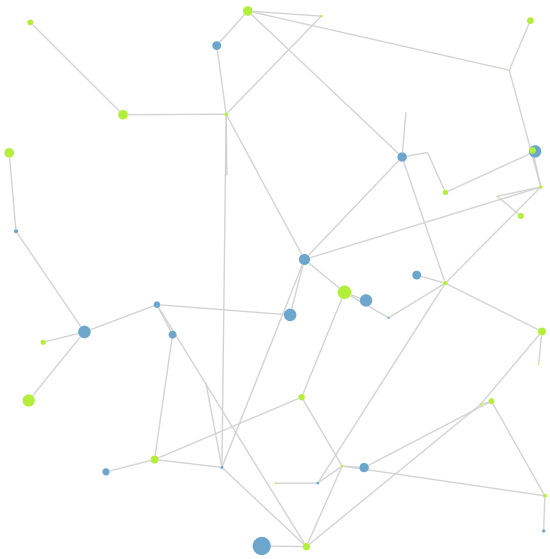


Figure 3.1.: Power grid with producers and consumers and the size of the nodes represent the magnitude of P_i allocated using method 2

the total amount of the produced energy is zero, so the consumers and producers are in balance. The two approaches have the following properties:

1. Half of the nodes are producers and have $P_i = +1$ and the other half are consumers with $P_i = -1$.
2. We randomly distribute producers and consumers using a regular Gaussian distribution with mean 0 and standard deviation 1, and subtract the average of all of them. So those grids do not necessarily have the same number of sources and sinks.

An example of a power grid is given in fig. 3.1.

3.1.2. Computation of static flow

Computing the static flows of a grid is a first condition for analyzing dynamic stability. Unless a stable state can be found which fulfills the boundary conditions as well as the physical laws, e.g. the flow through a line must not exceed its capacity, the grid cannot be dynamically stable. A grid fulfilling the static condition (eqs. (2.34) and (2.35)) is able to synchronize. If a synchronized state is possible depends on the topology and the attributed powers of the nodes. We set for all nodes the voltage $u = 1$ and focus on the solution of real power balance P (eq. (2.34)) and assume that each node can provide the necessary amount of reactive power Q. In order to reduce the computational effort, a synchronization criterion can be computed to state whether a synchronous state exists. We

usually refrain from computing the static flow condition in case the synchronization criterion is not met. The synchronization criterion is introduced by Dörfler et al. [2013] and defined by:

$$||\boldsymbol{R}P||_{\mathcal{E},\infty} < 1, \tag{3.2}$$

where we apply the ∞-norm over the edges and the norm is computed by: $||x||_{\mathcal{E},\infty} = \max_{\{i,j\}\in\mathcal{E},\infty}|x_i - x_j|$. So, the biggest dissimilarity of x is restricted. This criterion is only sufficient if more restrictions are considered.

3.1.3. Dynamical analysis using Kuramoto model

For the dynamical analysis we use the Kuramoto model introduced in section 2.2.3. The powers are distributed according to section 3.1.1 using both methods and we use $\alpha = 0.1$ for all nodes. In the following we consider the application of perturbations and the resulting setups of different test cases.

Definition of perturbations for dynamical analysis

Based on the static solution, we perturb a node by changing ϕ and $\dot{\phi}$ by adding random perturbations. The random perturbations are uniformly distributed between zero and a maximum value defined by the perturbation factor which we vary for different datasets. So, we use different magnitudes of the average perturbation. The magnitude is oriented by the investigations by Nitzbon et al. [2017] and in general we define the perturbation factor of the perturbation of the order of the ratio of the average of $|P|$ and α such as: $\frac{\overline{|P|}}{\alpha}$. Hence, the perturbation is defined by the probability and the perturbation factor:

$$\text{perturbation} = \text{perturbation factor} \times \text{probability}, \tag{3.3}$$

where the probability is uniformly chosen $\in [0, 1]$.

Test case setup for single-node basin stability

To compute SNBS, we consecutively investigate the stability of all nodes separately by using Monte-Carlo simulations. For each node we run multiple simulations using different perturbations to obtain a probability of how the grid reacts to perturbations of this particular node. Increasing the number of runs per node increases the accuracy of the prediction. For an accuracy of 10 % we have to run 100 perturbations per node, because the error scales with $\frac{1}{\sqrt{\chi}}$, where χ denotes the number of runs, respectively samples.

This problem can be set up as a regression or classification problem. In case of a regression-type problem, the SNBS is predicted as a value. In case of a classification problem, categories are

introduced firstly. In this thesis we only use binary classification, so all nodes below the median of the SNBS are considered instable and vice versa.

3.1.4. Implementation

We use the programming language Julia [Jul] for all simulations. To compute the synchronized state, we set up a root-finding problem and solve it by using the command *nlsolve* from the *NLsolve* package with default settings. The static condition sets the initial condition for the dynamical analysis. To compute the dynamic stability we use the Julia package *DifferentialEquations* by Rackauckas and Nie [2017] and set up a Second-Order-ODE-Problem based on the static flow condition independently of applying perturbations and solve the system using Tsitouras-5/4-Runge-Kutta algorithm and the default settings.

3.2. Properties of power grids and different datasets

In this section we look at properties of power grids, especially certain node properties and also introduce six generated datasets with different parameters. Using multiple datasets with significant differences helps to ensure the generality of the methods and we can also see if less complex datasets help the training process. Before evaluating the datasets, we start by looking at a conceptual categorization of nodes. Those properties can be used to analyze graphs and we will also use them as additional input features to check if this helps the training process.

Categorization of different nodes using the graph topology Nitzbon et al. [2017] introduce different node categories and show that they correlate with SNBS. Their concept of categorizing the nodes relies on the idea of splitting the graph $\mathcal{G}$ in in subparts which are tree-shaped. The program to categorize the nodes [Nod] is based on the PyTorch framework NetworkX [Net]. A subpart T' is called tree-shaped, if this subpart has exactly one node (*root*) that has at least one neighbor in the remaining graph. The root has a degree d that is at least 3. *Leaves* are part of T' and have $d = 1$ and inner tree nodes have $d > 1$. If leaves are directly connected to roots, they are called *sprouts* and leaves of larger trees are called *proper leaves*. Sprouts are split in two categories: Dense sprouts are connected to high-degree roots ($d > 5$) and sparse sprouts are connected to roots with $d < 6$. We summarize and attribute the following properties to each node:

- *Bulks* are nodes that are not part of trees.
- Each T' has one *root* that has $d \geq 3$.
- *Proper leaves* have $d = 1$ and are connected to inner leaves.
- *Inner tree nodes* connect sprouts, roots and other inner tree nodes.

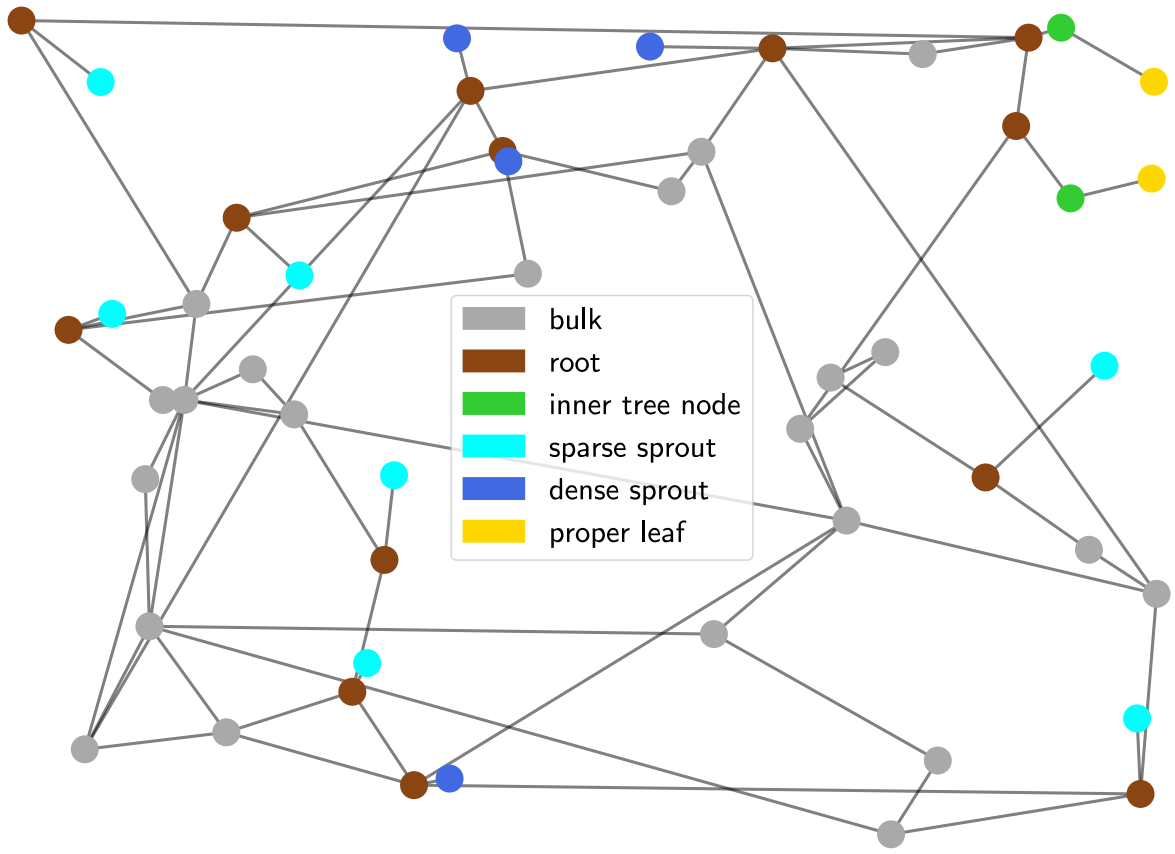


Figure 3.2.: Node categories by Nitzbon et al. [2017] for a graph of dataset fGN50 (see section 3.2.1)

- *Sparse sprouts* are connected to roots which have $d > 5$
- *Dense sprouts* are connected to roots which have $d < 6$.

In fig. 3.2 we show one example of a graph with its different node categories.

Further node properties may be used as additional input features The following measures can also be used to support the training:

- current-flow betweenness (see section 2.1.2)
- current-flow closeness centrality (see section 2.1.2)
- strength (see section 2.1.1)
- clustering coefficient (see section 2.1.2)

In the following the properties of the datasets are introduced. We use three different methods to generate datasets and for each method we use two different grid sizes (10 and 50 nodes). This leads to six datasets in total.

3.2.1. Dynamic datasets based on one fixed graph

The first two datasets rely on fixed graphs. So, the graph is the same for all samples within one dataset. We only change the nodal attributions P_i, so depending on the dataset, nodes with the same coordinates can be sources or sinks with different magnitudes. The first graph has 50 nodes and the second one only 10.

Dataset based on one fixed graph with 50 nodes (fGN50)

The dataset has the following properties:

- One fixed graph, so the topology is the same for all samples and the graph consists of 50 nodes.
- P_i are distributed randomly, using the second approach introduced in section 3.1.1 for 1000 times, leading to 1000 samples.
- We run 100 perturbations per node.
- The perturbation factor is set to 30.

To evaluate the distribution of SNBS of all nodes of a dataset, we plot the histogram shown in fig. 3.3. The first dataset has 1000 samples and the graph has 50 nodes, so in total we have 50000 cases with attributed SNBS. Looking at the histogram showing the normalized distribution with the relative frequency f on the y-axis, we can conclude that there are many nodes with SNBS=1, so those nodes are always stable. Besides the global maximum for SNBS=1, there is a local maximum for relatively small SNBS. This peak shows that there are several nodes that cause stability problems. The first peak shows nodes that weaken the stability and the second peaks shows nodes that strengthen the stability. By only applying positive perturbations (see section 3.1.3), the histogram differs from Nitzbon et al. [2017], where they have three peaks in the distribution, because they also apply negative perturbations.

Figure 3.4a shows SNBS and the averaged SNBS (AvBS) depending on the degree d. The degree is not a clear indicator for SNBS, as we can see, but there is a noticeable relation between d and AvBS. As with the degree, we can see an impact of the node categories introduced in section 3.2 on the AvBS, while there is no obvious relation to SNBS. In conclusion, we notice some relations between AvBS and the properties of the topology of the graph, even though having only one graph in the entire dataset.

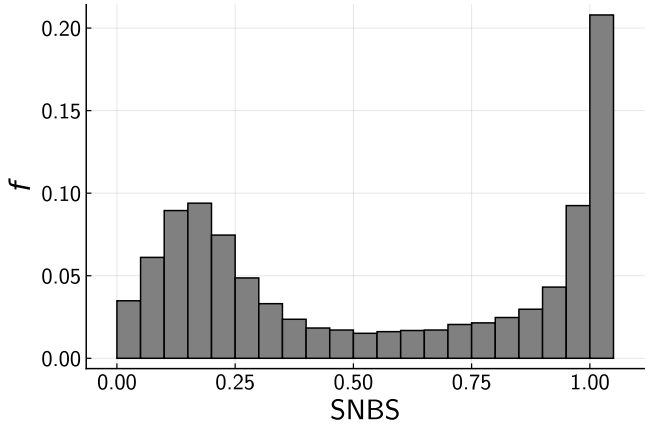
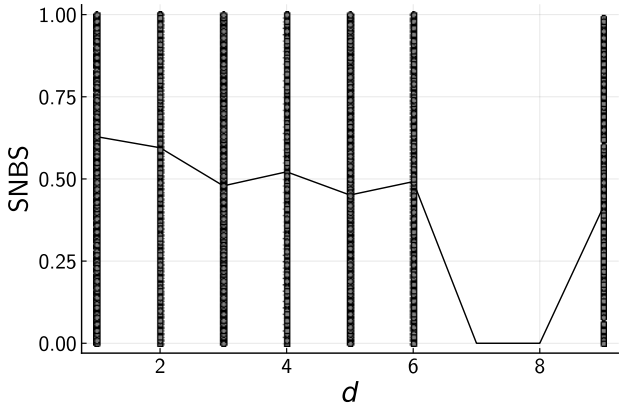
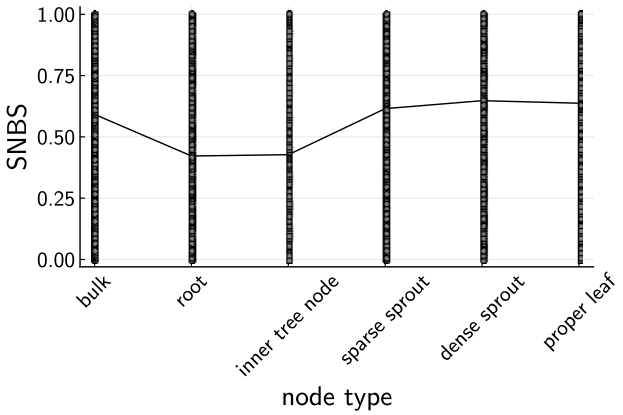


Figure 3.3.: Histogram showing the normalized distribution of SNBS for all 50000 cases (50 nodes per sample and 1000 samples) of fGN50 (fixed graph with 50 nodes). The distribution is normalized so that bin heights sum to 1 and f denotes the relative frequency.



(a) SNBS depending on the degree d and the solid line represents AvBS, which is set to 0 for $d = 7$ and 8, because there are no nodes with such a degree



(b) SNBS for different node types and the solid line shows AvBS

Figure 3.4.: SNBS of fGN50 (fixed graph with 50 nodes)

Dataset based on one fixed graph with 10 nodes (fGN10)

This dataset has only 10 nodes, so we can efficiently increase the sample size and we use the following properties:

- One fixed graph, so the topology is the same for all samples and the graph consists of 10 nodes.
- P is distributed randomly, using the second approach introduced in section 3.1.1 for 10000 times, leading to 10000 samples.
- We run 100 perturbations per node.
- The perturbation factor is set to 20.

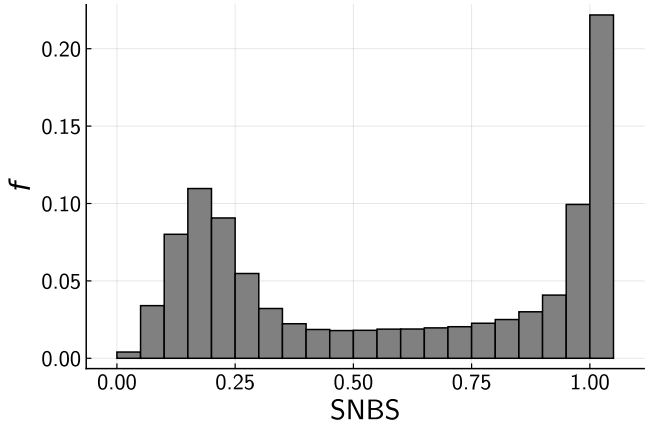


Figure 3.5.: Histogram showing the distribution of SNBS for all 100,000 cases (10 nodes per sample and 10000 samples) of fGN10 (fixed graph with 10 nodes). The distribution is normalized so that bin heights sum to 1 and f denotes the relative frequency.

Figure 3.5 shows the distribution of the SNBS over all samples and nodes and we can see that it is comparable to fGN50 for a perturbation factor of 20 instead of 30. Detailed information about this dataset is given in appendix B.1.

3.2.2. Dynamic datasets with absolute values of sources/sinks fixed to 1

For all following datasets we focus on some properties of the datasets and additional information is given in appendix B.1. The following two datasets have different topologies. To reduce the complexity, the absolute value of all sources/sinks is fixed to 1. The producers and consumers cancel each other out, because we have equally many of them. Again, we have two datasets, where one of them is based on graphs with 50 nodes and the other graph is of size 10.

Dataset with fixed absolute values of sources/sinks and 50 nodes (P1N50)

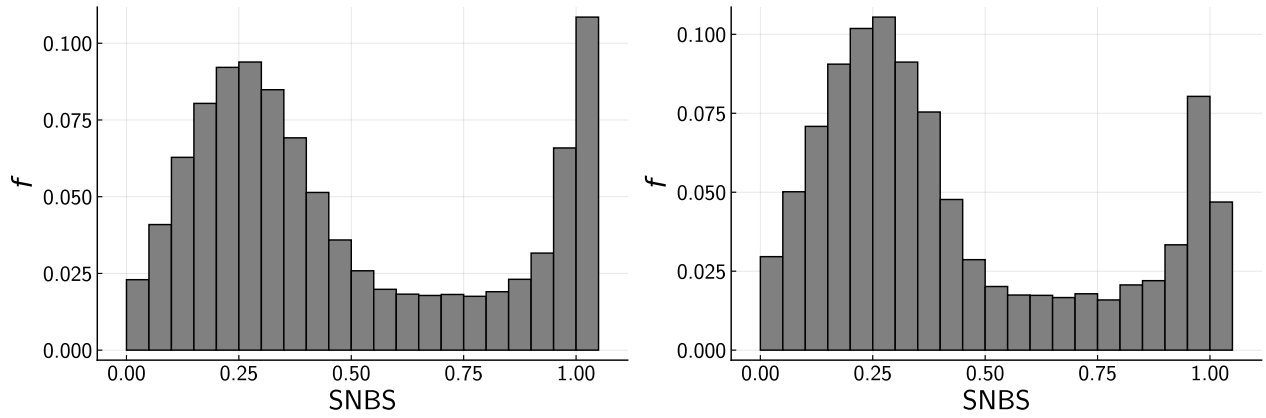
- The dataset consists of 1000 generated grids.
- Each grid has 50 nodes.
- All sources and sinks have a magnitude of 1, so the first approach is used, which is introduced in section 3.1.1.
- We run 100 perturbations per node.
- The perturbation factor is set to 20.

The histogram is given in fig. 3.6a.

Dataset with fixed absolute values of sources/sinks and 10 nodes (P1N10)

- The dataset consists of 2000 generated grids.
- Each grid has 10 nodes.
- All sources and sinks have a magnitude of 1, so the first approach is used, which is introduced in section 3.1.1.
- We run 500 perturbations per node.
- The perturbation factor is set to 20.

We can see in the histogram (fig. 3.6b) that there are less nodes that are stable in all cases in comparison to the test case with 50 nodes (fig. 3.6a)).



(a) P1N50 (50000 cases)

(b) P1N10 (20000 cases)

Figure 3.6.: Histogram showing the distribution of SNBS for datasets with multiple topologies and absolute values of power/sources fixed to one. The distributions are normalized so that bin heights sum to 1 and f denotes the relative frequency.

3.2.3. Dynamic datasets with random values for sources/sinks

The next datasets are also based on different topologies and we additionally use random values for P_i instead of fixed ones. So those datasets are more complex. Again, we have two datasets with different sizes of the grids using 50 and 10 nodes.

Dataset with random absolute values of sources/sinks and 50 nodes (PrandN50)

- The dataset consists of 1400 generated grids.
- Each grid has 50 nodes.

- The powers P_i are distributed randomly, using the second approach, which introduced in section 3.1.1.
- We run 100 perturbations per node.
- The perturbation factor is set to 25.

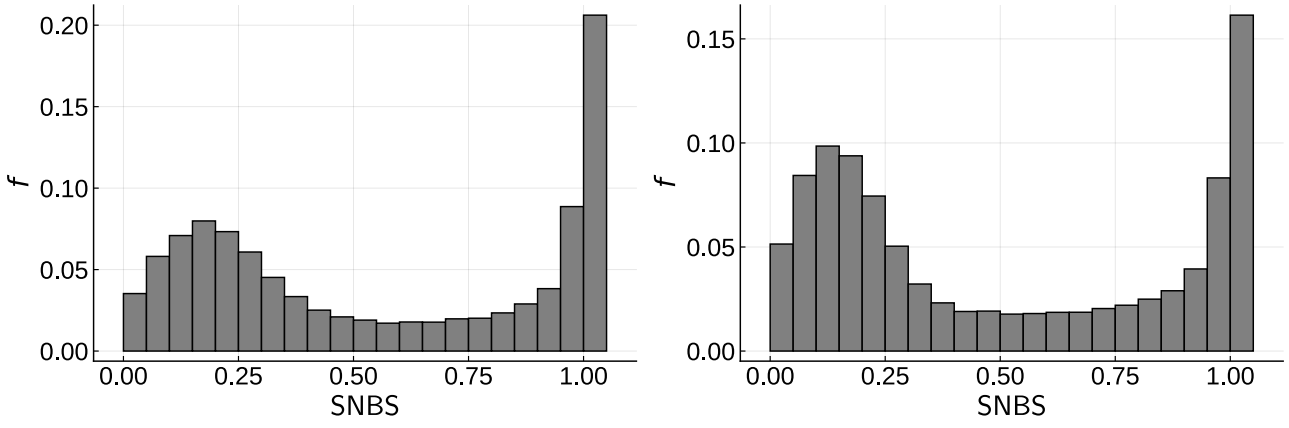
The histogram is given in fig. 3.7a and we clearly see many nodes that strengthen stability in all cases.

Dataset with random absolute values of sources/sinks and 10 nodes (PrandN10)

The dataset has the following properties:

- The dataset consists of 10000 generated grids.
- Each grid has 10 nodes.
- The powers P_i are distributed randomly, using the second approach, which is introduced in section 3.1.1.
- We run 100 perturbations per node.
- The perturbation factor is set to 30.

The histogram for using ten nodes (fig. 3.7b) is comparable to histogram of the grids with 50 nodes. On the contrary to the test cases, where the absolute values of P is fixed to 1, we see in the histogram (fig. 3.7b) more nodes with SNBS=1 in case of the grids with 10 nodes.



(a) PrandN50 (70000 cases)

(b) PrandN10 (100,000 cases)

Figure 3.7.: Histogram showing the distribution of SNBS for datasets with multiple topologies and random values for sources/sinks. The distributions are normalized so that bin heights sum to 1 and f denotes the relative frequency.

3.2.4. Methods to evaluate training process and overview of datasets

To evaluate different models, we use the deviation and accuracy. When predicting the SNBS of the nodes as a regression-type problem, we compute the average of the deviation of the predicted and true SNBS depending on the labels l and model outputs y by:

$$\text{deviation} = \frac{1}{n} \sum_{n} |l_i - y_i|, \tag{3.4}$$

where we sum over all nodes n of the grid. Secondly, the accuracy founds on counting the results of meeting a criterion. Let the criterion be:

$$|y_i - l_i| < \text{tolerance} = 0.3. \tag{3.5}$$

Whenever the criterion is met, we count this sample as accurate. In case of using classes instead of probabilities, we count all correctly predicted nodes to obtain the accuracy. For all investigations in this thesis we choose to set the tolerance to 0.3.

To quantify the success of the trained model, we introduce a simple model called 0.5-model which simply predicts SNBS of each node to 0.5. This is a reasonable output for an untrained ANN with outputs between 0 and 1. Depending on the generated datasets we have different deviations and accuracies using this simple model. An overview of all datasets including the deviation and accuracy using the 0.5-model is shown in table 3.2.

reference name	number of samples	number of nodes	key property	deviation of 0.5-model in %	accuracy in %
fGN50	1000	50	fixed graph	35.4	30.5
fGN10	10000	10	fixed graph	34.6	33.1
P1N50	1000	50	$P_i \in \{-1, 1\}$	28.4	53.1
P1N10	2000	10	$P_i \in \{-1, 1\}$	28.2	54.5
PrandN50	1400	50	$P_i \in [-1, 1]$	34.1	35.3
PrandN10	10000	10	$P_i \in [-1, 1]$	34.6	31.6

Table 3.2.: Overview of dynamic datasets, the deviation is computed for a 0.5-model

3.3. Application of artificial neural networks

In this section the methodology of applying multiple ANNs to predict the SNBS of our generated datasets is described and the used methods are compared. The goal of this section is to identify promising methods and to find reasons for differences in the performance. To apply ANNs, we use

PyTorch [Paszke et al., 2019], which is an open source toolbox for machine learning algorithms. For GNNs, we use an additional software package called *PyTorch Geometric* [Fey and Lenssen, 2019] which relies on *PyTorch*. All applications follow the basic scheme shown in fig. 3.8. We provide the graph $\mathcal{G}$ and the node features $\boldsymbol{X}$ as inputs to obtain a vector of size n (number of nodes) containing the SNBS. We apply two general types of ANNs: CNNs and GNNs. We investigate CNNs, because they are well advanced and can extract spatial relations of pictures, so perhaps CNNs can also be applied to power grids. GNNs on the other hand are still relatively new, but they are specifically designed to analyze graph-structured data, so they are a promising approach for analyzing power grids.

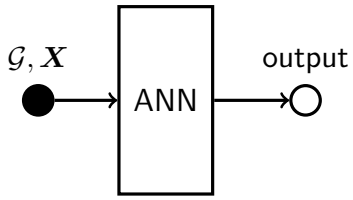


Figure 3.8.: Application of GNN with the aim to obtain the output (graph-level or node-level) for the inputs $\mathcal{G}$ and $\boldsymbol{X}$

3.3.1. Using convolutional neural networks to predict the ability to synchronize of power grids

Instead of using SNBS to compare different CNN-methods we start by comparing different CNN-models on the task to predict the ability to synchronize. This prediction is a graph-classification problem and there are only two possible conditions, whether the grid is able to synchronize or not. Hence, this task is set up as a binary graph-classification.

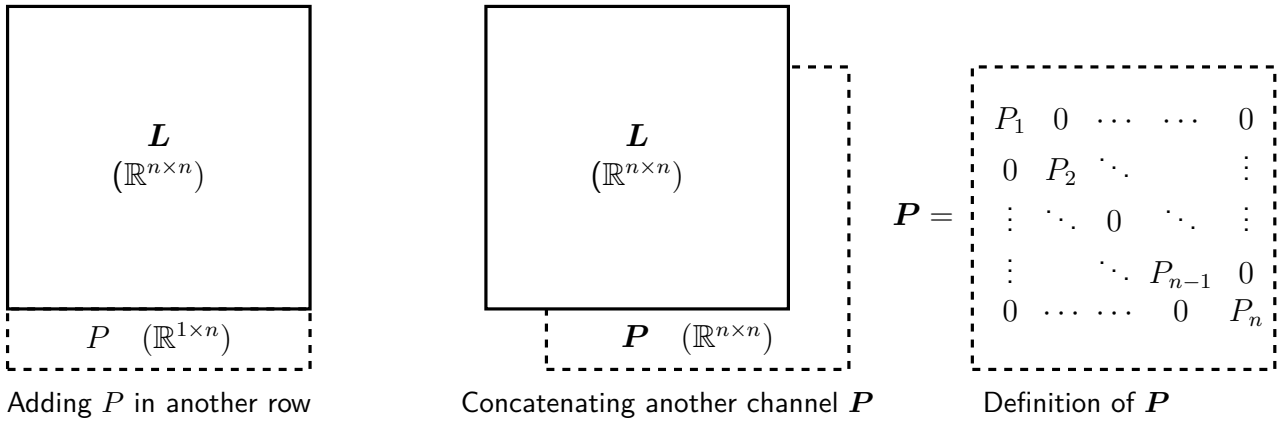
The datasets are generated according to section 3.1.2 and the number of samples is equal in both categories. The datasets include three limitations resulting in an unrealistically high number of grids that are not able to synchronize, but the dataset still fits the purpose of investigating CNN-models. The first problem concerns the topology, because the generated grids do not represent realistic models of power grids due to wrongly connected nodes that lead to many intersections, because some nodes are connected across long distances. Secondly, the computation of the ability to synchronize is wrongly implemented. Thirdly, the ordering of the nodes is computed without considering the weights of the edges. However, we can still use this dataset to compare different ANNs, since we do not draw any conclusions from the data and only consider the trainability. For this setup we only have the injected power P_i as node features, so the node features are a vector $\mathbb{R}^{n\times 1}$.

Setup of CNNs

When using CNNs we provide $\mathcal{G}$ by using the graph Laplacian $\boldsymbol{L}$. Using $\boldsymbol{A}$ or even multiple powers of $\boldsymbol{A}$ did not show significant improvements, so we refrain from showing those results. However, we experiment with providing P to the CNNs in two different ways:

1. Firstly, we add an additional row to our matrix, leading to an input of $1 \times (n+1) \times n$ per sample.[1]
2. Secondly, we add another channel, where the information of P is contained in the second channel which is a matrix of $\mathbb{R}^{n \times n}$, but all non-diagonal elements are zero and the diagonal consists of P_i. So the input has the following dimension: $(2 \times n \times n)$.

The first approach has the advantage that the input is of much lower dimension (almost half of the input values). Approach number 2 however puts $\boldsymbol{L}$ in a spatial context to P, since the information is at the same relative position, which is in analogy to applying CNNs to colored pictures, where each red-green-blue (RGB) color would be one channel. The setups are shown in fig. 3.9.



(a) Input of size: $1 \times n + 1 \times n$ (b) Input of size: $2 \times n \times n$

Figure 3.9.: Two ways of providing nodal input to CNNs

We investigate CNNs based on ResNet (section 2.3.5) and AlexNet (section 2.3.5) with a slightly modified version of AlexNet introduced by Krizhevsky [2014]. Furthermore, we have to make small adjustments to the CNNs such as changing output layers to one and adding one sigmoid layer at the end or changing the input dimensions to match our setup. To avoid completely different $\boldsymbol{L}$ for similar graphs, we order all our nodes using the current-flow betweenness (section 2.1.2). Besides comparing AlexNet- and ResNet-architectures, we vary several parameters in order to get a feeling for an appropriate setup. We use SGD for training with a momentum of 0.9 if not mentioned differently.

[1] First factor shows the number of input channels

Static Testcase 1

The first datasets consists of 24252 samples. To find an appropriate setup we investigate the following parameters:

- input: $1 \times 51 \times 50$ or $2 \times 50 \times 50$,
- architecture: AlexNet, ResNet18, ResNet34, ResNet50.
- sample size (SS): 24252, 1000, 10000,
- batch size (BS): 10, 50, 100, 200, 500, 1000,
- scheduler: none, stepLR, ReduceLROnPlateau (plateauLR),
- initial learning rate: 0.01 - 0.0001.

The scheduler stepLR decays the learning rate of each parameter every step size by a given constant (for details see appendix B.4.1). ReduceLROnPlateau reduces the learning rate when the valid loss has stopped improving. The results of training this dataset are presented in table 3.3, where TL denotes test loss and TA test accuracy out of several epochs, but not necessarily the same epochs. There are many cases, when the best TA and TL are not met within the same epoch. The accuracy is the share of correctly predicted samples.

For most tested configurations the results are comparable, which means that training is relatively stable and not highly influenced by the investigated sets of parameters, but the overall accuracy is relatively low with a maximum at 65 %. Interestingly the type of inputs ($1 \times 51 \times 50$ or $2 \times 50 \times 50$) has no relevant impact on the results. Different types of CNNs do also not impact the result very much. A sample size of 1000 seems to be sufficient. More samples do not really improve the performance. The BS, LR and scheduler also have minor impact. In one case, the LR is set very low which results in a very low accuracy of 52 % meaning that the network cannot really be trained. In the next step, the input dimension is reduced to see if a less complex problem can be trained more easily.

Static Testcase 2

To reduce the complexity, we investigate grids with 10 instead of 50 nodes. Using less nodes enables us to also use larger datasets (SS = 62822). We focus on the two-channel setup, so our input is of size: $2 \times 10 \times 10$. We investigate the following parameters:

- CNN-architecture: ResNet18, ResNet34, AlexNet,
- BS: 500, 1000.

input	CNN	SS	BS	scheduler	LR	TL	TA in %
$1 \times 51 \times 50$	AlexNet	1000	10	none	.001	.672	60
$2 \times 50 \times 50$	AlexNet	1000	10	none	.001	.666	64
$1 \times 51 \times 50$	AlexNet	10000	10	none	.001	.629	64
$2 \times 50 \times 50$	AlexNet	10000	10	none	.001	.639	64
$1 \times 51 \times 50$	AlexNet	24252	10	none	.001	.628	63
$2 \times 50 \times 50$	AlexNet	24252	10	none	.001	.628	63
$1 \times 51 \times 50$	AlexNet	10000	100	none	.001	.619	65
$2 \times 50 \times 50$	ResNet18	10000	10	none	.001	.668	63
$1 \times 51 \times 50$	AlexNet	10000	200	none	.001	.622	64
$2 \times 50 \times 50$	ResNet34	10000	10	none	.001	.647	63
$1 \times 51 \times 50$	AlexNet	10000	500	none	.001	.633	62
$2 \times 50 \times 50$	ResNet50	10000	10	none	.001	.653	62
$1 \times 51 \times 50$	AlexNet	10000	200	none	.01	.698	52
$2 \times 50 \times 50$	ResNet50	10000	200	none	.001	.646	63
$1 \times 51 \times 50$	AlexNet	10000	100	none	.001	.629	63
$2 \times 50 \times 50$	ResNet50	10000	200	none	.005	.652	63
$1 \times 51 \times 50$	AlexNet	10000	100	none	.005	.625	64
$1 \times 51 \times 50$	AlexNet	10000	100	none	.0005	.624	64
$1 \times 51 \times 50$	AlexNet	10000	100	none	.0001	.626	63
$2 \times 50 \times 50$	ResNet50	10000	200	none	.0001	.653	62
$2 \times 50 \times 50$	ResNet18	10000	100	none	.0001	.681	59
$2 \times 50 \times 50$	ResNet18	10000	100	plateauLR	.001	.662	61
$1 \times 51 \times 50$	AlexNet	10000	200	plateauLR	.001	.626	64
$2 \times 50 \times 50$	ResNet18	10000	100	stepLR	.001	.662	63
$1 \times 51 \times 50$	AlexNet	10000	200	stepLR	.001	.628	63

Table 3.3.: Results of static test case 1, SS: sample size, BS: batch size, LR: learning rate, TL: test loss, TA: test accuracy

Additional information on the setup is given in appendix B.4.1. The results of the training are shown in table 3.4. The accuracy is much higher in comparison to the first static test case (table 3.3), so reducing the grid size helps to improve the training process. Again, we see that the investigated variations of the parameters lead to comparable results. Perhaps a setup with a smaller number of nodes also works better, because the same sized convolution filters are applied to larger portions of the network. When applying a filter of a sufficiently large size, it can capture most of the neighborhood of a node. Since we did not significantly change the filter sizes of the CNNs, large parts of the network can only be captured by one filter, if $\boldsymbol{L}$ is roughly of the same dimension.

Conclusions of using CNNs with graphs represented by graph Laplacian

To conclude, we can say that it is possible to train CNNs by providing the grid in a matrix shape and separately providing nodal information, however the accuracy is relatively low. For our applications,

CNN	BS	TL	TA in %
ResNet18	500	.472	78
AlexNet	500	.452	78
AlexNet	1000	.452	78
ResNet34	1000	.474	78
ResNet34	5000	.48	77

Table 3.4.: Results of static test case 2, graphs with 10 nodes, BS: batch size, TL: test loss, TA: test accuracy

CNNs face several limitations:

- The investigated setups are fixed to one number of nodes.
- All setups provide the information in a sparse and relatively inefficient way.
- Node properties cannot be clearly specified as node properties, because there is no distinction between providing the topology and the features of nodes.
- Prediction of smaller grids is much better, so the methods do not seem to be applicable to large networks.

Regarding the decreasing performance of larger grids, one can guess that spatial relations between nodes with large differences in the node numbering cannot be captured appropriately, because they are far away from another in $\boldsymbol{L}$, even though they might be close in terms of the shortest path length.

Using images to predict ability to synchronize

One way of using ML to different types of problems is to convert the problems to images, since ML-tools work very well at visual analyze. For example, Fourier transformations are applied to sound signals using spectrograms as inputs for CNNs [Hershey et al., 2017]. Graphs can be visualized easily, so we apply CNNs on plots of the graph. We use different colors and sizes of the nodes to represent different properties of nodes. Since all power lines have the same property, we do not have to vary the type of the edges. An example of the input with the used and original resolution is given in fig. 3.10. We clearly see unrealistic edge connections across long distances, especially when comparing the grid to fig. 3.1. The resulting intersections of many edges can be one of the reasons, why the visual analysis does not seem to work well. Hence, this approach is not followed in this thesis, even though this approach might work better, when using more realistic models for power grids. Details regarding the training are given in appendix B.4.1.

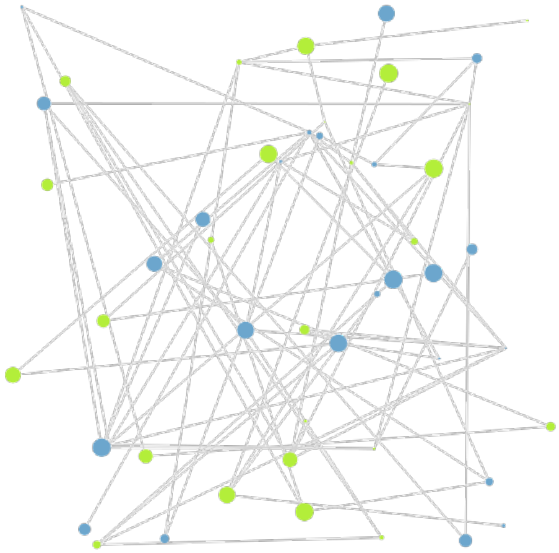


Figure 3.10.: Power grid with producers and consumers and the size of the nodes represent the magnitude of P_i in the original resolution that is used as an input for CNNs

3.3.2. Predicting dynamic stability using graph neural networks

In this section we introduce the methodology to investigate the dynamic stability as SNBS by using GNNs. On the contrary to investigating the ability to synchronize, the dynamic stability is set up as a node-regression or node-classification problem. We only consider GNNs in this section, because the application of CNNs is straightforward when combining this section and section 3.3.1.

Setup of GNN

All investigations are based on providing two general types of input:

1. Graph consisting of vertices and edges including the weight of the edges,
2. Node features, for example the powers P_i.

An exemplary setup of a GNN using three convolutions is shown in fig. 3.11. There are ReLU-activations after the first two convolutional layers. Any type of the filters introduced in section 2.3.6 can be used as convolutional layers. Another parameter we must choose are the number of channels of each convolutional layer. The number of input channels of the first channel is equal to the number of the provided node features. The number of the output channel of the last layer is one, because we are only interested in determining one feature per node. The remaining number of channels can be chosen freely if the number of the input channels matches the number of the output channels of the preceding layer.

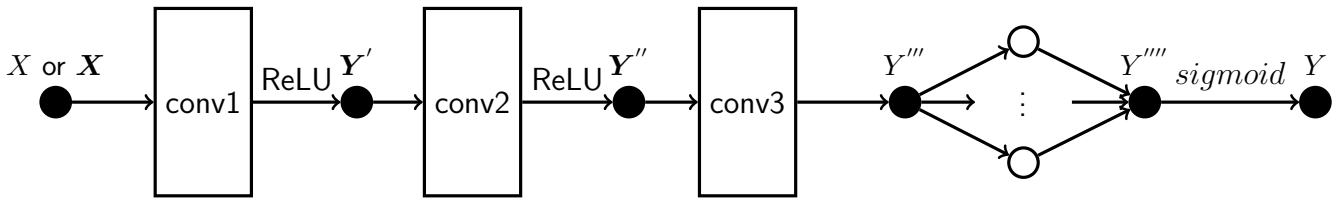


Figure 3.11.: Architecture of a GNN using three convolutions and ReLU-activation after the first two convolutions and a fully connected layer at the end

GNN models

We investigate different models and vary the convolutional type, the number of convolutions and channels, as well as applying activation functions, batch normalization and dropout. The default dropout settings of PyTorch are used if not mentioned differently. The investigated models are introduced in the following and categorized depending on the type of used convolution.

Models based on GCN [Kipf and Welling, 2016]

Three GCN-models are studied in detail, so we introduce their details.

GCNNet1 has the following structure:

1. GCN-convolution, with the number of input channels (NIC) equal to the number of node features and the number of output channels (NOC) equal to 4,
2. ReLU and dropout,
3. GCN-convolution (NIC = 4, NOC = 1),
4. ReLU,
5. fully connected layer and sigmoid output layer.

GCNNet2 has the following structure:

1. GCN-convolution (NIC = number of node features, NOC =16),
2. ReLU and dropout,
3. GCN-convolution (NIC = 16, NOC = 4),
4. ReLU,
5. GCN-convolution (NIC = 4, NOC = 1),
6. fully connected layer and sigmoid output layer.

GCNNet3 has the following structure:

1. GCN-Convolution (NIC = number of node features, NOC =16),
2. batch normalization, ReLU and dropout,
3. GCN-Convolution (NIC = 16, NOC = 4),
4. batch normalization, ReLU,
5. GCN-Convolution (NIC = 4, NOC = 1),
6. batch normalization, fully connected layer and sigmoid output layer.

Models based on Arma-convolutions [Bianchi et al., 2019]

All Arma-convolutions use three stacks and four layers.

ArmaNet1 has the following structure:

1. Arma-Convolution (NIC = number of node features, NOC =1, ReLU activation),
2. ReLU and dropout,
3. fully connected layer and sigmoid output layer.

ArmaNet2 has the following structure:

1. Arma-Convolution (NIC = number of node features, NOC =16, dropout = 0.25, ReLU activation),
2. Batch normalization and dropout,
3. Arma-Convolution (NIC = 16, NOC = 1, dropout = 0.25),
4. batch normalization, ReLU, fully connected layer and sigmoid output layer.

Models based on Simple Graph Convolution [Wu et al., 2019b]

SGNet1 consists of only one convolutional layer

1. SG-Convolution (NIC = number of node features, NOC =1, number of hops=2),
2. ReLU, fully connected layer and sigmoid output layer.

SGNet1WOReLU consists of only one convolutional layer and no activation function:

1. SG-Convolution (NIC = number of node features, NOC =1, number of hops=2),
2. fully connected layer and sigmoid output layer.

SGNet2 consists of the following layers:

1. SG-Convolution (NIC = number of node features, NOC =4, number of hops=2),
2. ReLU, dropout,
3. SG-Convolution (NIC = 4, NOC =1, number of hops=2),
4. ReLU, fully connected layer and sigmoid output layer.

SGNet3 consists of the following layers:

1. SG-Convolution (NIC = number of node features, NOC =16, number of hops=2),
2. ReLU, dropout,
3. SG-Convolution (NIC = 16, NOC =4, number of hops=2),
4. ReLU,
5. SG-Convolution (NIC = 4, NOC =1, number of hops=2),
6. fully connected layer and sigmoid output layer.

One Model based on Topology Adaptive Graph-Convolutions [Du et al., 2017]

TAGNet1 has two convolutional layers and the following setup:

1. TAG-Convolution (NIC = number of node features, NOC =4),
2. ReLU, dropout,
3. TAG-Convolution (NIC = 4, NOC =1),
4. fully connected layer and sigmoid output layer.

Overview of GNNs models

Table 3.5 shows the introduced models with the number of convolutions and the number of learnable parameters.

Providing additional node features

From the paper by Nitzbon et al. [2017], which is explained in section 3.2 we know that certain types of nodes influence the SNBS. We want to investigate if explicitly providing this information as additional input helps to improve the training process. Perhaps, simple GNNs are not capable of aggregating this information properly, but if this information is provided additionally the performance might be improved. The results of this analysis are given in chapter 4.

name	number of convolutions	number of learnable parameters
ArmaNet1	2	38
ArmaNet2	3	1047
GCNNet1	2	15
GCNNet2	2	107
GCNNet3	3	149
SGNet1	1	4
SGNet1WOReLU	1	4
SGNet2	2	15
SGNet3	3	107
TAGNet1	2	39

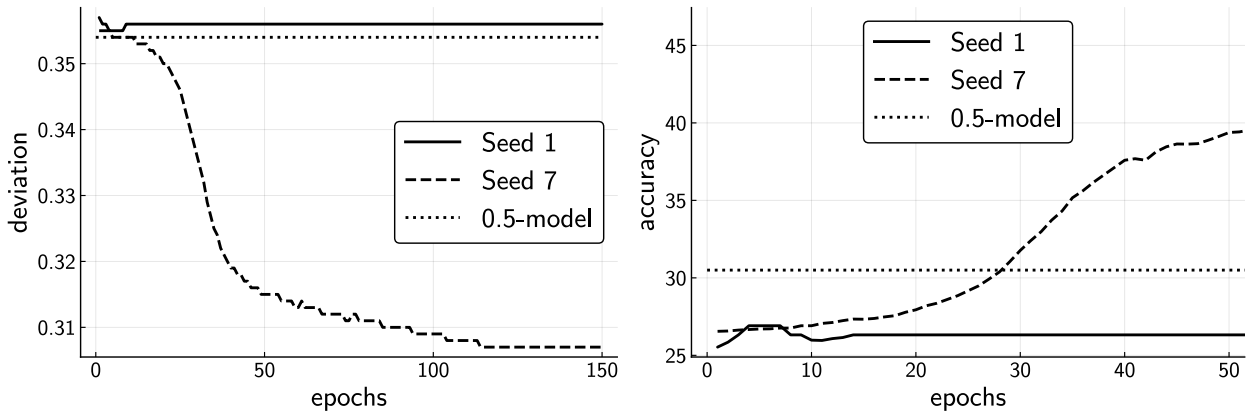
Table 3.5.: GNN-models and number of convolutional layers and learnable parameters

3.3.3. Influence of random initialization during training

Training neural networks includes some randomness which is also desired in some parts. In our case the randomness is defined by the seed in PyTorch and has two main effects for our investigations:

- The initial weights and biases and
- the order of training examples depends on the seed.

There are cases when just changing the seed prevents any training progress. For example, in fig. 3.12, we see the training for two equal setups except for different seeds. To ensure the generality of the methods, one should vary the seed multiple times and investigate its influence. However, this limits the possibility of investigating multiple models and we rather investigate different architectures. By not investigating the influence of the seed in detail, all statements have limited validity.



(a) deviation (b) accuracy

Figure 3.12.: Training progress for the same setup and different seeds

3.3.4. Computational effort of training models

In the following the computational effort is described. However, there was no focus on reducing the training time by parallelization or other improvements in the code. Hence, we only give a brief overview of the training time. Throughout the thesis different hardware is used, but we only use one cluster with GPU (nVidia Kepler K40 accelerator) for benchmarking. The following computation times are given for training of fGN50 with 1000 samples for 50 epochs and one node feature:

- ResNet18: 256 s
- ResNet34: 260 s
- ResNet50: 267 s
- GCNNet1: 607 s
- ArmaNet1: 622 s
- ArmaNet2: 639 s

When using 10000 samples for grids of size 10 (fGN10), we have the following computation times for the training using 25 epochs:

- GCNNet2 1590 s
- ResNet34: 825 s

We can see that the training time is relatively low of all sets and takes between 4 and 30 minutes. The sampling size has a large impact on the training time. However, one has to keep in mind that more epochs may be necessary for smaller sample sizes. Training CNN-models takes about half the time of training GNN-models, but more efficient implementations of GNNs could close this gap. When considering the number of learnable parameters of each model[2] we notice that the number of parameters has only small effects on the training time. For example, ResNet50 has about twice as many parameters as ResNet18, but the training time is only slightly increased and all GNN-models have much less learnable parameters than CNN, but training takes more time.

[2]Number of learnable parameters is given for GNNs in table 3.5 and for CNNs in appendix B.3

4. Results and Discussion

In this chapter the results of the simulation are shown and discussed. The chapter is split in four sections. The first two sections cover the prediction of SNBS as a probability using CNNs at first and later GNNs. Afterwards we consider the prediction of SNBS using two classes only: stable and instable nodes. This reduces the complexity in comparison to predicting the probability and we compare the performance of CNNs and GNNs again. Throughout the sections we combine the presentation and discussion of the results. In addition, we combine and interpret aspects of all investigations at the end.

4.1. Predicting single-node basin stability using convolutional neural networks

The section is ordered by the datasets. We compare different architectures and properties for all datasets. The training protocols are given in appendix B.4.2.

4.1.1. Dataset of fixed graph with 50 nodes

This dataset fGN50[1] consists of one fixed topology with a graph of 50 nodes. We compare ResNet 18 and ResNet 34 and also evaluate the influence of adding further input node features. The results of the training showing deviation and accuracy are given in fig. 4.1. When models exceed the deviation of $35.4\,\%$ computed using the 0.5-model[2], the models are not trained successfully. However, we see that all shown models can be trained successfully. The final deviation is little above 30 %, so it is reduced by only 4 %. The accuracy can be improved by roughly 20 %. Apparently including multiple node features does not help the training. The overall performance is not increased when using multiple input features and the training process seems less robust as we see in the peaks and the increase of the deviation after some epochs shown by the dashed lines.

[1] This dataset is introduced in section 3.2.1
[2] This model for the evaluation of the performance is introduced in section 3.2.4

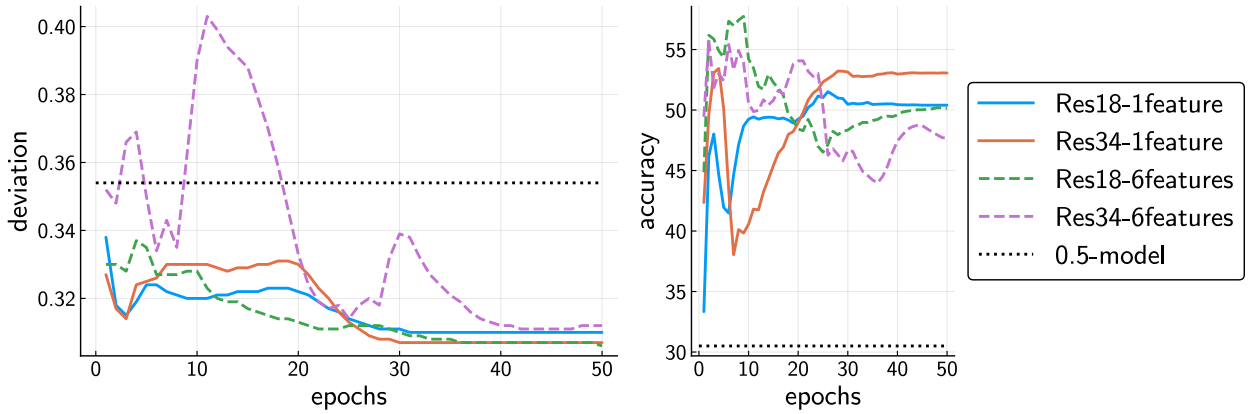


Figure 4.1.: Training progress using CNNs and the dataset fGN50

4.1.2. Dataset of fixed graph with 10 nodes

When investigating the dataset with a smaller graph fGN10[3], but more samples, less epochs are necessary to reach the final state (fig. 4.2). There is no progress after 5 epochs, so the train and test loss remain constant (fig. B.8). One reason could be, that there is enough redundant information within the dataset so that the number of epochs leading to an improvement is limited. In comparison to the dataset with a graph of 50 nodes, the performance is much higher. The deviation is lower than 0.2 and the accuracy is at almost $80\,\%$. Hence, reducing the input dimension helps CNNs to predict SNBS. Since both ResNet-architectures have a comparable performance, the additional effort of training ResNet34 instead of ResNet18 appears to be unnecessary.

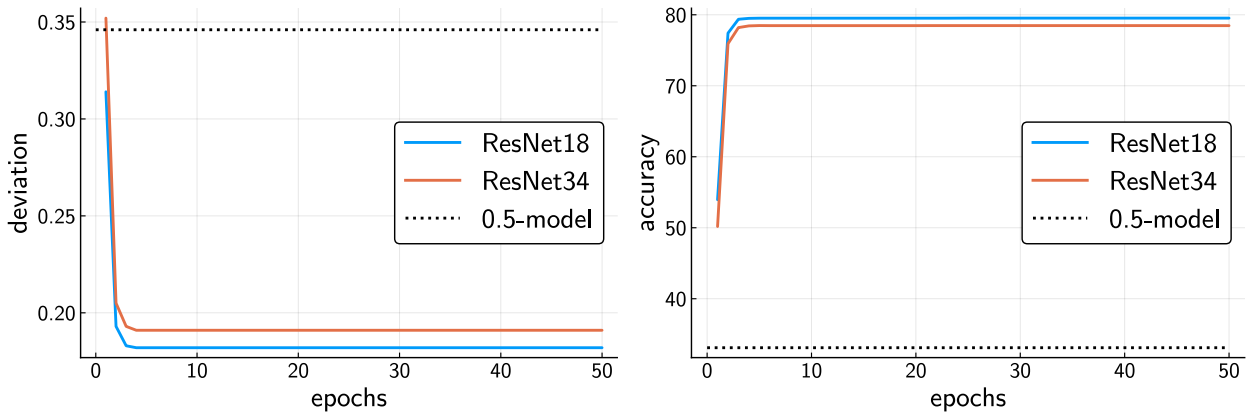


Figure 4.2.: Training progress using CNNs and the dataset fGN10

[3]This dataset is introduced in section 3.2.1

4.1.3. Dataset of graphs with fixed magnitudes of sources/sinks and 50 nodes

On the contrary to the previous investigated datasets, this dataset P1N50[4] includes different topologies. Furthermore, the magnitude of all sources and sinks is set to 1. There are several fluctuations of the deviation and the accuracy during the first epochs, until a stable level is reached (see fig. 4.3). When looking at the training loss, we see that it continuously decreases, but the valid loss and test loss sometimes increase presented in fig. B.9. The training process takes about thirty epochs until it levels out. Again, both ResNet-architectures are comparable. However, the overall improvement in comparison to the 0.5-model, is relatively low. The accuracy is only increased by 5-6 %.

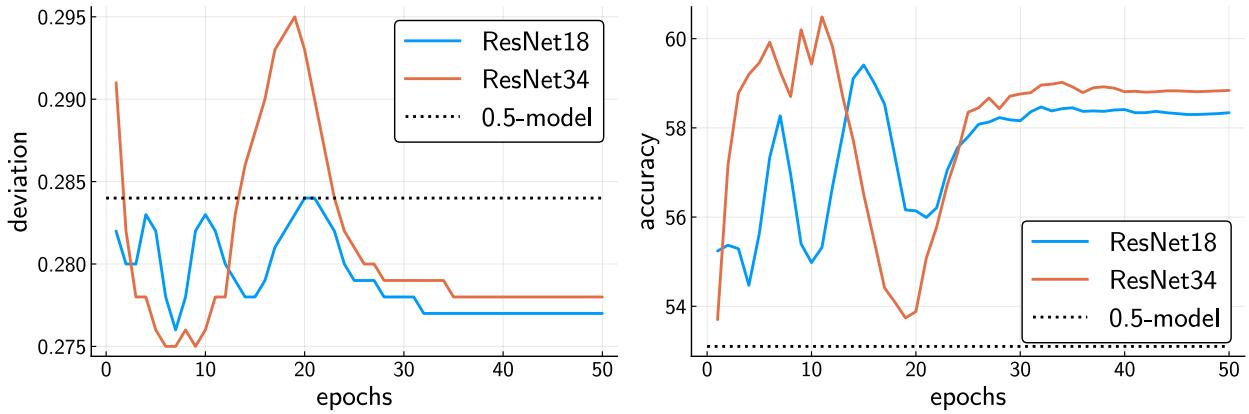


Figure 4.3.: Training progress using CNNs and the dataset P1N50

4.1.4. Dataset of graphs with fixed magnitudes of sources/sinks and 10 nodes

The dataset P1N10[5] consists of graphs of size 10 with fixed magnitudes of sources and sinks. Reducing the input dimension of the graphs again helps to significantly increase the performance presented in fig. 4.4, when comparing those results to the setup with 50 nodes (fig. 4.3). The deviation is decreased by roughly 7 % for the smaller grids instead of less than 1 % when considering graphs with 50 nodes. On the contrary to fGN10, this dataset consists of only 2000 samples, which is one fifth. We clearly see that more epochs are necessary for P1N10 to reach a final state. Training takes about half as many epochs in comparison to PrandN50, which makes sense since this dataset has half as many samples.

[4]This dataset is introduced in section 3.2.2
[5]This dataset is introduced in section 3.2.2

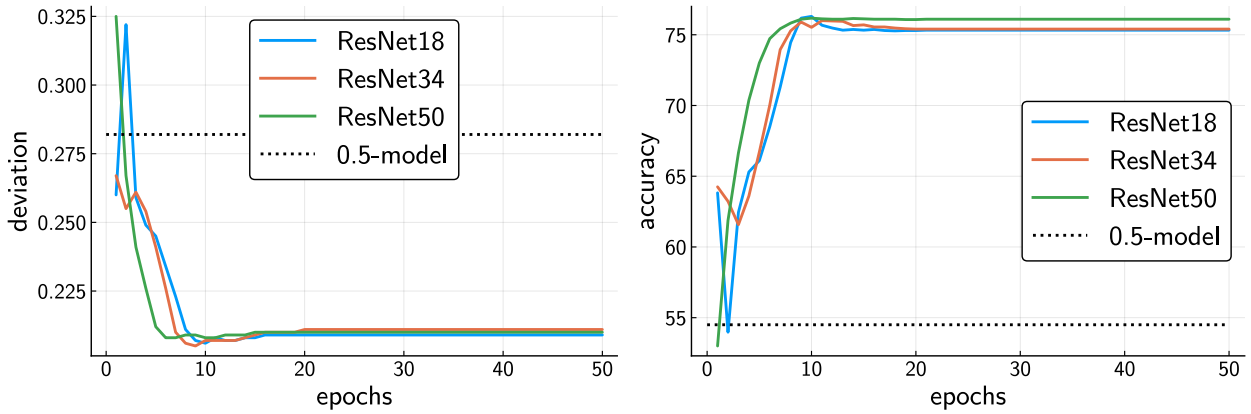


Figure 4.4.: Training progress using CNNs and the dataset P1N10

4.1.5. Dataset of graphs with random sources/sinks and 50 nodes

This dataset PrandN50[6] is the most complicated dataset, because it consists of different topologies and the magnitude of the sources and sinks also varies. The training process is not very successful (fig. 4.5). The deviation is reduced by $0.2\,\%$. The accuracy is increased by up to 6 % in case of ResNet18. Using a more complex ResNet-architecture does not improve the training of this complicated dataset. We also see that it takes up to 30 epochs to get this minor improvement.

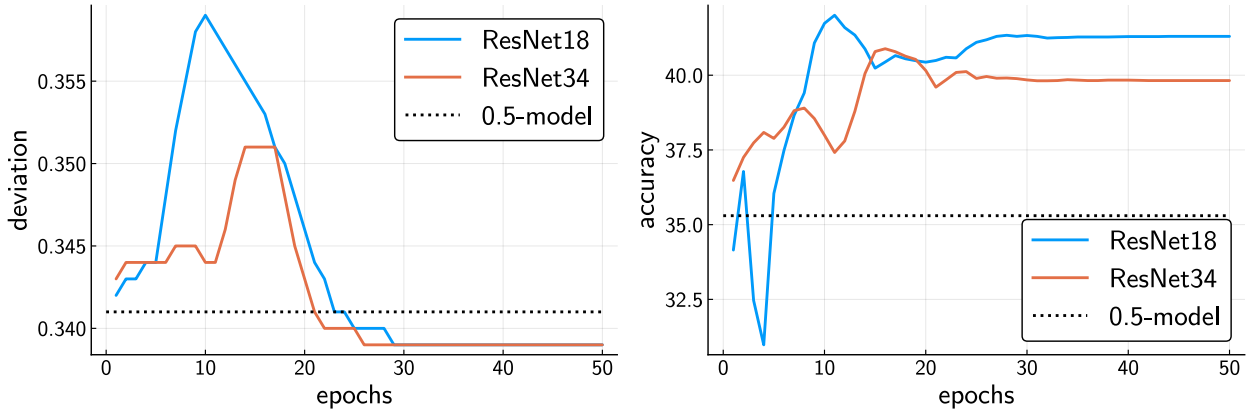


Figure 4.5.: Training progress using CNNs and the dataset PrandN50

4.1.6. Dataset of graphs with random sources/sinks and 10 nodes

The graphs of his dataset PrandN10[7] have 10 nodes. On the contrary to PrandN50, training performance is much better for PrandN10 due to the reduced input dimension (fig. 4.6). The reduction of deviation is about 5 % and the accuracy can be improved by about 20 %. We also observe

[6]This dataset is introduced in section 3.2.3
[7]This dataset is introduced in section 3.2.3

that less epochs are necessary due to the large dataset. When comparing the ResNet-architectures, we notice that ResNet18 performs a bit worse than the others, but the overall difference is small between the models.

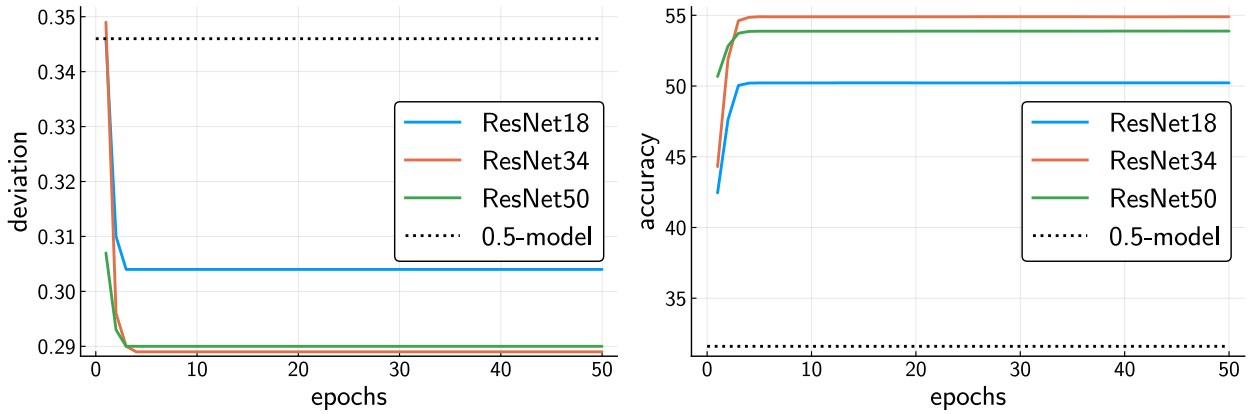


Figure 4.6.: Training progress using CNNs and the dataset PrandN10

4.1.7. Overview of improvements due to training

To get an overview of the trainability of different architectures and the performance of different models, the final deviation and accuracy as well as the increase of performance is shown in table 4.1. We clearly see that training gets more difficult with increased complexity. The best results are achieved in case of fGN10, for which the deviation is reduced by more than 15 %. For PrandN50 the training results in only small improvements of less than 1 %. There are also no big differences depending on the architecture or the number of input features, so training is relatively robust in different circumstances.

dataset/model	final deviation	final accuracy	improvement of deviation	improvement of accuracy
fGN50				
0.5-model	35.4	30.5		
Res18-1F	31.0	50.4	4.4	19.9
Res34-1F	30.7	**53.1**	4.7	2.6
Res18-6F	**30.6**	50.1	4.8	19.6
Res34-6F	31.2	47.7	4.2	17.2
fGN10				
0.5-model	34.6	33.1		
Res18	**18.2**	**79.5**	16.4	46.4

Continued on next page

dataset/model	final deviation	final accuracy	improvement of deviation	improvement of accuracy
Res34	19.1	78.5	15.5	45.4
P1N50				
.5-model	28.4	53.1		
Res18	**7.7**	58.3	0.7	5.2
Res34	27.8	**58.8**	0.6	5.7
P1N10				
.5-model	28.2	54.5		
Res18	**20.9**	75.3	7.3	20.8
Res34	21.1	75.4	7.1	20.9
Res50	21.0	**76.1**	7.2	21.6
PrandN50				
.5-model	34.1	35.3		
Res18	33.9	**41.3**	0.2	6.0
Res34	33.9	39.8	0.2	4.5
PrandN10				
.5-model	34.6	31.6		
Res18	30.4	50.2	4.2	18.6
Res34	**28.9**	**54.9**	5.7	23.3
Res50	29.0	53.9	5.6	22.3

Table 4.1.: Overview of training process and improvement of dynamic stability using CNNs. All measures are in %. 1F means one input feature per node and 6F means that six node features are used as input. If no information is provided, then one input feature per node is used. The best deviation and best accuracy are shown by bold characters.

4.2. Predicting single-node basin stability using graph neural networks

As in the last section we compare different models for each dataset, but using GNNs instead of CNNs. An overview of the results is given at the end of this section (section 4.2.7). The training protocols are given in appendix B.4.3.

4.2.1. Dataset of fixed graph with 50 nodes

We start again by investigating the dataset fGN50[8] with one fixed topology of 50 nodes. Multiple architectures are compared in figs. 4.7 and 4.8, where we see large differences in the performance. When considering the models using SGNets, we see that training of SGNet2 is not successful. Furthermore, the good results of SGNet1 without activation function shows that the aggregation of information of neighbors is crucial and apparently more important than introducing non-linearities, at least in this case. TAGNet1 achieves the best performance of all configurations. GCNNet3 being the most complex model based on GCN-convolutions performs worse than the other GCN-models.

When providing additional node features as input, training appears to be more difficult. Training examples are given in appendix B.4.3. However, we must keep in mind that the additional node features are equal in all cases, since we only one topology for this dataset.

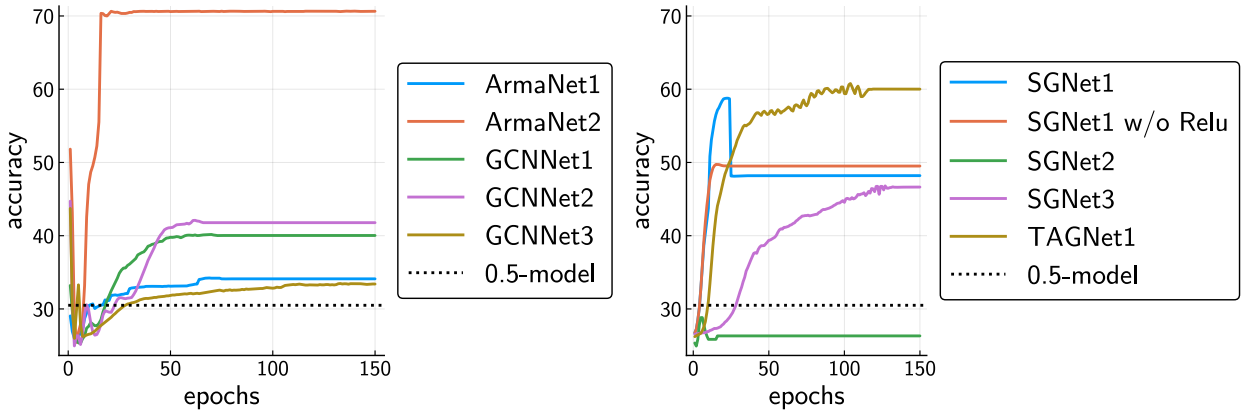


Figure 4.7.: Training progress showing accuracy of multiple GNN-models for fGN50, w/o means without

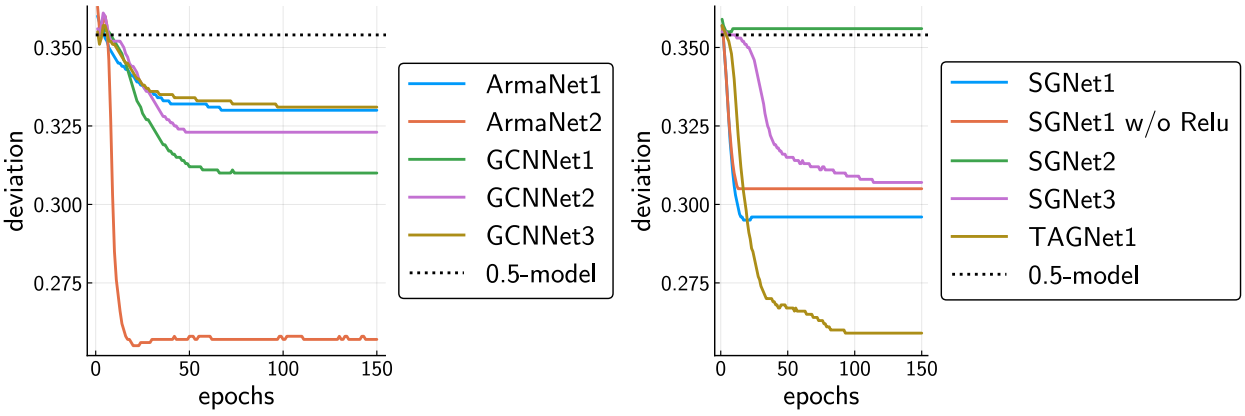


Figure 4.8.: Training progress showing deviation of multiple GNN-models for fGN50

[8]This dataset is introduced in section 3.2.1

4.2.2. Dataset of fixed graph with 10 nodes

When using a smaller graph with only ten nodes fGN10[9], training is expected to be easier. However, the training process represented by the deviation and the accuracy (fig. 4.9) is less successful in comparison to fGN50 (see section 4.2.1). The overall improvement in accuracy and the reduction in deviation is lower for fGN10. In case of SGNet3 and TAGNet1, the deviation is only slightly lower than the deviation of the 0.5-model. Both ArmaNet-configurations perform well. The large datasets result in a training, where only the first epochs are relevant. SGNet2 is not trained at all which shows the sensitivity and missing robustness of this and perhaps all investigated methods. We can already conclude that simply reducing the grid sizes does not increase the training performance.

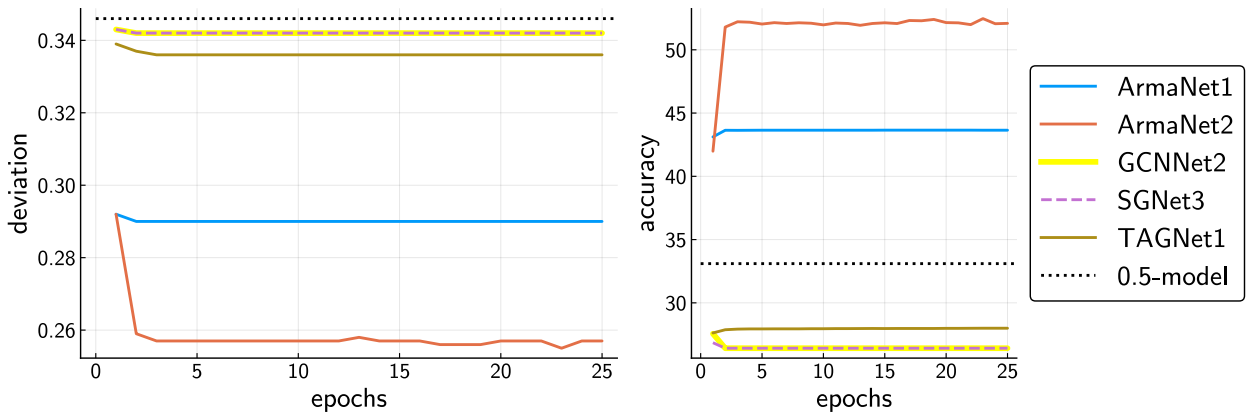


Figure 4.9.: Evaluation of models during training of fGN10

4.2.3. Dataset of graphs with fixed magnitudes of sources/sinks and 50 nodes

The training of the dataset with fixed magnitudes P1N50[10] is successful for all used models and ArmaNets as well as TAGNet1 perform well (fig. 4.10). The SGNets only decrease the deviation a little bit and in case of SGNet2 training also takes many epochs. Most models reach its final state after roughly 20 epochs.

[9] This dataset is introduced in section 3.2.1
[10] This dataset is introduced in section 3.2.2

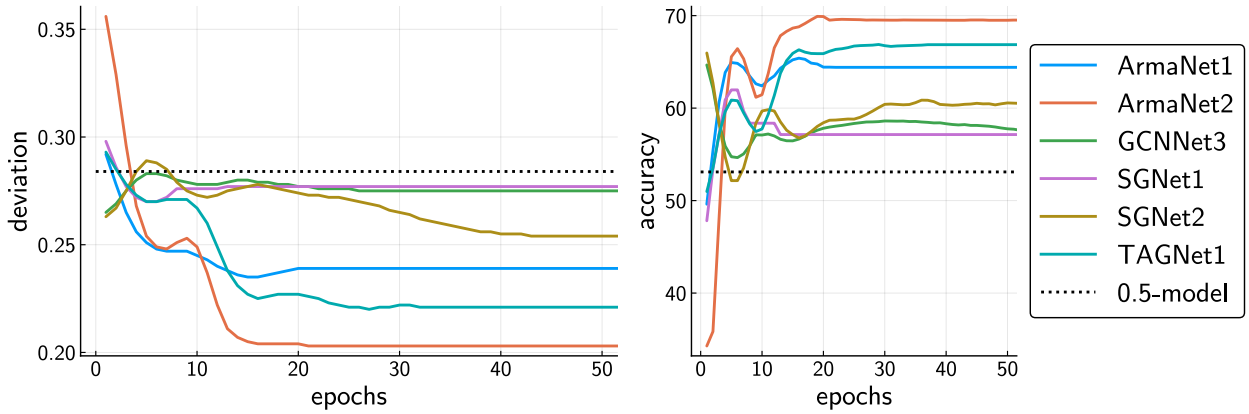


Figure 4.10.: Evaluation of models during training of P1N50

4.2.4. Dataset of graphs with fixed magnitudes of sources/sinks and 10 nodes

This dataset P1N10[11], has only 10 nodes per graph. When comparing the results of P1N10 in fig. 4.11 to the results of P1N50, we do not see big differences in the performance. Both, the deviation and accuracy reach similar levels. Apparently, the reduced complexity of considering grids with smaller size is not easier to train. Hence, we can emphasize that training does not depend very much on the investigated grid sizes.

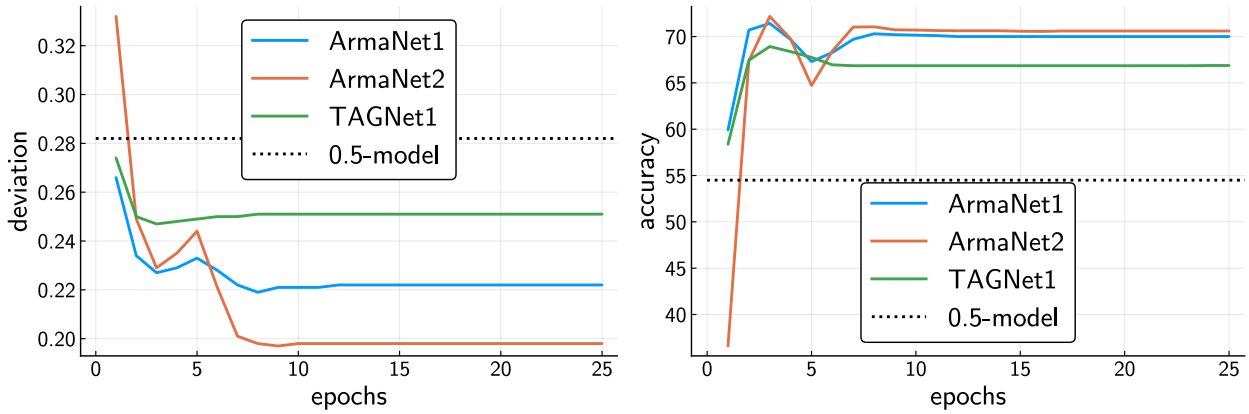


Figure 4.11.: Evaluation of models during training of P1N10

4.2.5. Dataset of graphs with random sources/sinks and 50 nodes

When training the dataset PrandN50[12] with one node feature as input, we see in fig. 4.12 that only the ArmaNet-configurations are trained successfully. Adding additional node features prevents

[11]This dataset is introduced in section 3.2.2
[12]This dataset is introduced in section 3.2.3

training progress in case of ArmaNet2. In case of TAGNet1 and when using two node features, training is improved as we see in fig. 4.13. The increased performance of TAGNet1 when using multiple node features as input does not necessarily mean that the added information is crucial. One reason could also be a different initialization since more channels are added. In section 3.3.3 the influence of the seed is already mentioned and perhaps adding more channels also significantly changes the initialization. It is also interesting to mention that more epochs are needed to converge in case of using two node features as input for TAGNet1.

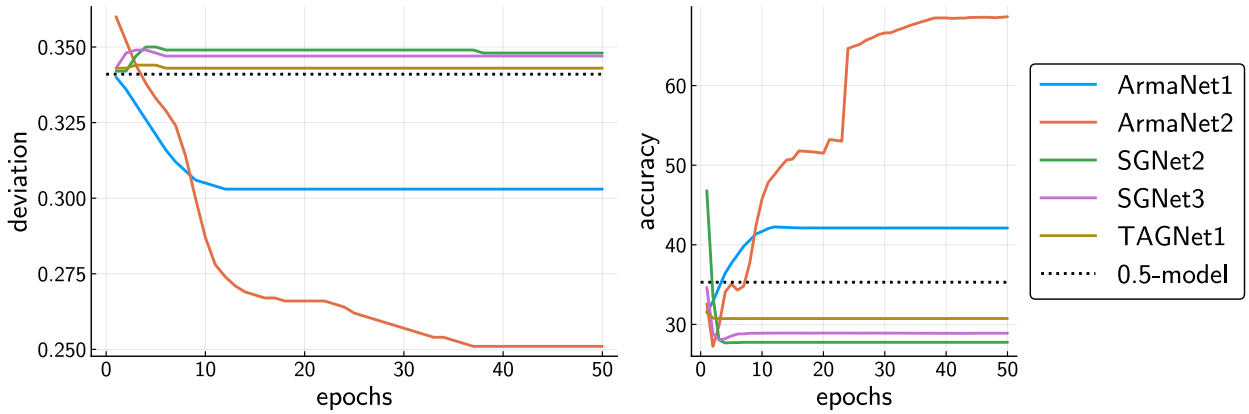
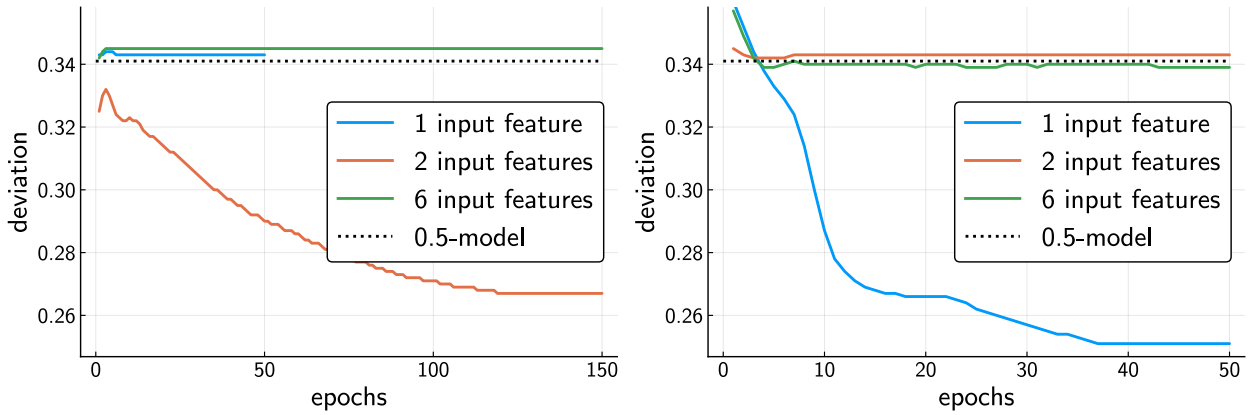


Figure 4.12.: Evaluation of models during training of PrandN50



(a) TAGNet1 (b) ArmaNet2

Figure 4.13.: Using multiple input node features during training of PrandN50

4.2.6. Dataset of graphs with random sources/sinks and 10 nodes

This dataset PrandN10[13] consists of graphs of size ten and the sources/sinks are distributed randomly. As with the previous analyses, the reduced grid dimension also does not result in a better

[13]This dataset is introduced in section 3.2.3

training performance as we see in fig. 4.14 when using random magnitudes for P_i. Instead of an improvement, the training performance is decreased in comparison to the best performance of investigating PrandN50. On the contrary to PrandN50, ArmaNets are the best again and in case of PrandN10, TAGNet1 cannot be trained successfully. This shows that the entire training process is not very robust and can be highly influenced.

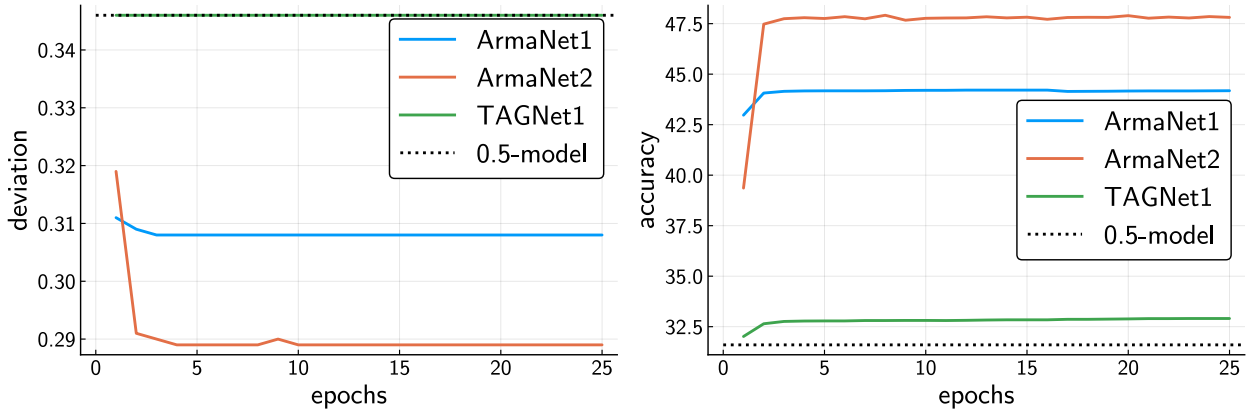


Figure 4.14.: Evaluation of models during training of PrandN10

4.2.7. Overview of improvements due to training

Table 4.2 shows the improvements of all datasets and different models. On the contrary to CNNs, there are many configurations that do not result in an improved deviation or accuracy when using GNNs. In general, ArmaNet2 and sometimes TAGNet1 perform the best. The best overall performance occurs for fGN50 and ArmaNet2 with a reduced deviation of almost 10 % and an increased accuracy by more than 40 %. For the most complex dataset PrandN50, ArmaNet2 achieves an improvement of 9 % in terms of the deviation and increases the accuracy of more than 33 %. For less difficult datasets, ArmaNet2 also performs worse in some cases. For example, ArmaNet2 improves the deviation only by 5.7 % in case of PrandN10. For P1N10 and P1N50, ArmaNet2 reduces the deviation by roughly 8 %. There is no big difference in the performance for datasets P1N10 and P1N50 using ArmaNet2, so the input dimension seems to have a minor impact on the results. A reduced dependency on the grid size is one of the observations from using GNNs, which is different from the observations using CNNs.

dataset/model	final deviation	final accuracy	improvement of deviation	improvement of accuracy
fGN50-1F				
0.5-model	35.4	30.5		

Continued on next page

dataset/model	final deviation	final accuracy	improvement of deviation	improvement of accuracy
ArmaNet1	33.0	34.1	2.4	3.6
ArmaNet2	**25.7**	**70.6**	9.7	40.1
GCNNet1	31.0	40.0	4.4	9.5
GCNNet2	32.3	41.8	3.1	11.3
GCNNet3	33.1	33.4	2.3	2.9
SGNet1	29.6	48.2	5.8	17.7
SGNet1WORelu	30.5	49.5	4.9	19.0
SGNet2	35.6	26.3	-0.2	-4.2
SGNet3	30.7	46.6	4.7	16.1
TAGNet1	25.9	60.0	9.5	29.5
fGN50-6F				
0.5-model	35.4	**30.5**		
ArmaNet2	**35.2**	27.6	0.2	-2.9
GCNNet2	35.6	25.9	-0.2	-4.7
fGN10				
0.5-model	34.6	33.1		
ArmaNet1	29.0	43.6	5.6	10.5
ArmaNet2	**25.7**	**52.1**	8.9	19.0
GCNNet2	34.2	26.4	0.4	-6.7
SGNet3	34.2	26.4	0.4	-6.7
TAGNet1	33.6	28.0	1.0	-5.1
P1N50				
0.5-model	28.4	53.1		
ArmaNet1	23.9	64.4	4.5	11.3
ArmaNet2	**20.3**	**69.5**	8.1	16.4
GCNNet3	27.1	57.2	1.3	4.1
SGNet1	27.7	57.1	0.7	4.0
SGNet2	25.2	60.7	3.2	7.6
TAGNet1	22.1	66.9	6.3	13.8
P1N10				
0.5-model	28.2	54.5		
ArmaNet1	22.2	70.0	6.0	15.5
ArmaNet2	**19.8**	**70.6**	8.4	16.1
TAGNet1	25.1	66.9	3.1	12.4

Continued on next page

dataset/model	final deviation	final accuracy	improvement of deviation	improvement of accuracy
PrandN50				
0.5-model	34.1	35.3		
ArmaNet1	30.3	42.1	3.8	6.8
ArmaNet2	**25.1**	**68.7**	9.0	33.4
SGNet2	34.8	27.8	-0.7	-7.6
SGNet3	34.7	28.9	-0.6	-6.4
TAGNet1	34.3	30.7	-0.2	-4.6
PrandN50-2F				
0.5-model	34.1	35.3		
ArmaNet1	34.9	27.6	-0.8	-7.7
ArmaNet2	34.3	32.5	-0.2	-2.8
TAGNet1	**26.7**	**51.8**	7.4	16.5
PrandN50-6F				
0.5-model	34.1	35.3		
ArmaNet1	34.5	29.3	4.8	-6.0
ArmaNet2	**33.9**	**31.4**	0.2	-3.9
TAGNet1	34.5	29.3	-0.4	-6.0
PrandN10				
0.5-model	34.6	31.6		
ArmaNet1	30.8	44.2	3.8	12.6
ArmaNet2	**28.9**	**47.8**	5.7	16.2
TAGNet1	34.6	32.9	0.0	1.3

Table 4.2.: Overview of training process and improvement of dynamic stability using GNNs. All measures are in %. 1F, 2F, 6F describe the number of input features per node. The number of input features does not change within two horizontal lines. If no information is provided, then 1 input feature per node is used. The best deviation and best accuracy are shown by bold characters.

4.2.8. Application of one trained model on two datasets

The application of using one trained model on different dataset is evaluated in this subsection. One could for example train models on small grids and then apply the trained models on large grids to save the additional training effort. We investigate the performance of a model, which is trained on grids with 10 nodes and apply it to grids of size 50. We use the ArmaNet02-model for the two training-datasets P1N10 and P1N50 and evaluate both on P1N50. The performance is presented in

fig. 4.15 and we see that both models achieve comparable results. Hence, predicting the stability of larger grids based on models that are trained for smaller grids is possible. So, we can conclude that GNNs are capable of extracting relevant information independently of the trained grid size and can generalize this information. In conclusion, GNN are one approach of reducing the complexity of the analysis of the dynamic stability of power grids. One can refrain from conducting simulations of the full-sized problem to train models, but only considers parts of the real grid for training and may still apply the trained model on the full domain of the original problem.

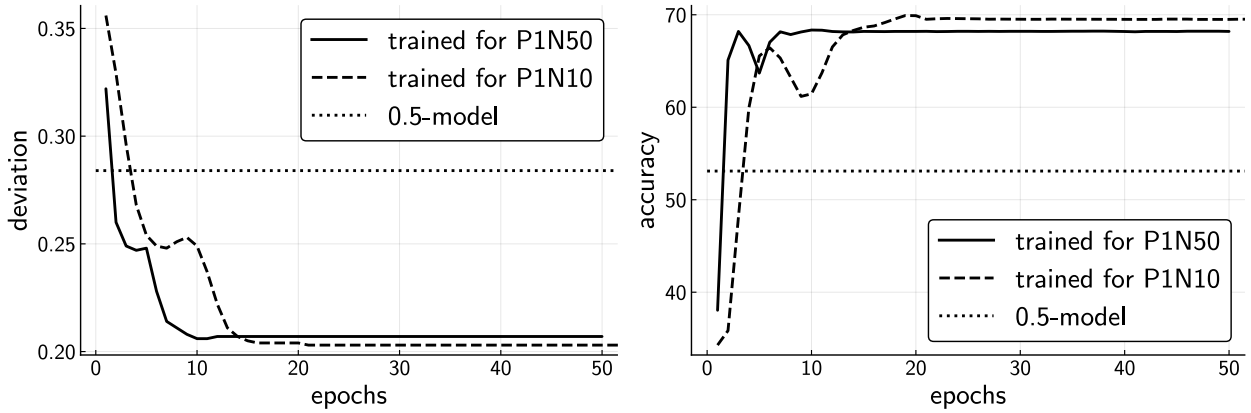
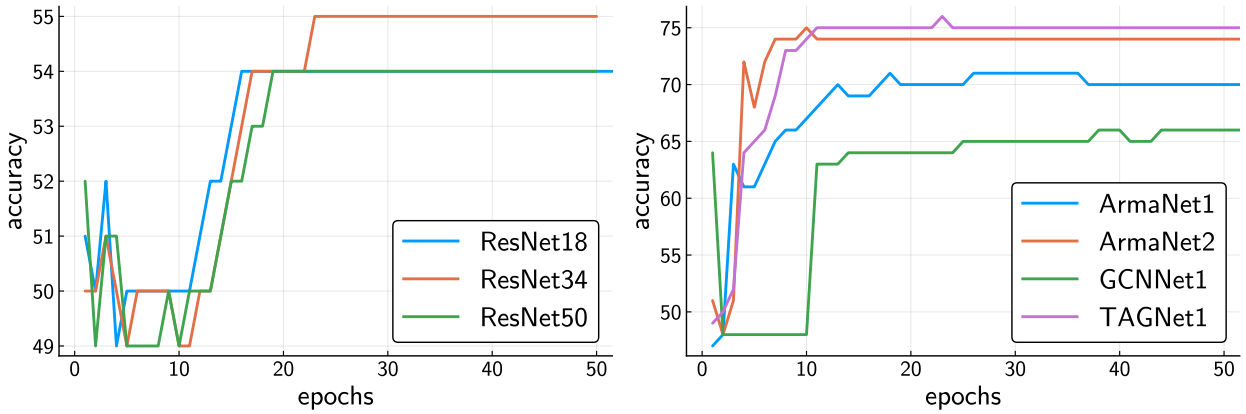


Figure 4.15.: Deviation (left) and accuracy (right) of evaluating two differently trained ArmaNet02-models which are evaluated on P1N50. The model showed by the solid line is trained using P1N50, while the dashed line represents a model which is trained using P1N10.

4.3. Predicting single-node basin stability when considering two classes

On the contrary to the previous sections, we simplify the setup by only considering two classes of SNBS: stable or instable instead of predicting values. Hence, this is a node-classification problem. The training process for PrandN50 using CNNs and GNNs is shown in fig. 4.16. Apparently GNNs are much better at predicting the class accurately. All GNN-configurations are better than the investigated ResNets. The difference between different ResNets is negligible. In this setup TAGNet1 achieves the best results and predicts the correct class of nodes with an accuracy of 75 %. ArmaNet2 is almost as good as TAGNet1, while other GNN-configurations perform worse, but still significantly better than CNNs.



(a) CNNs (b) GNNs

Figure 4.16.: Accuracy of binary classification of SNBS of nodes using the median of the dataset as threshold

4.4. Interpretation of the observed results of all datasets

In this section we focus mainly on the interpretation of GNN methods, since the application of CNN appears to be limited to small grids. The previously discussed results show that GNNs can generalize different graphs and one can also train models on small grid sizes and apply them on larger grids. We achieved comparable results when training one model on graphs with 10 and 50 nodes and applying those models on a graph of 50 nodes. Hence in the latter case, the model is trained and evaluated on the same dataset. Hence, learnable parameters (weights) can be shared across different grid sizes. We also observed that the performance highly depends on the type of convolution, so we discuss some of the reasons in the following.

TAGNet, as well as the ArmaNet-configurations outperform all other models and the main reason could be the consideration of more neighbors. Both methods have in common that higher orders of neighboring nodes are used due to the quasi-usage of multiple exponents of $\boldsymbol{A}$ per layer. In case of TAGNet, one directly computes different powers of normalized $\boldsymbol{A}$ and in case of ArmaNets one refrains from computing different powers of $\boldsymbol{A}$ directly but uses $\boldsymbol{A}$ multiple times as a factor within one layer. ArmaNets also uses activation functions between the multiplication of $\boldsymbol{A}$, but the result of the convolutional layer is still influenced by higher orders of $\boldsymbol{A}$. By using such higher orders of $\boldsymbol{A}$ the considered region is increased and higher order neighbors are taken into account. Hence, more spatial information is aggregated and considered per layer. Other configurations such as GCN can only consider higher order of neighbors by increasing the number of layers. However, adding more layers does not necessarily increase the performance for GCN, as we see when comparing GCNNet1, GCNNet2 and GCNNet3. In case of SGNets, one includes higher orders of $\boldsymbol{A}$ by setting the number of hops. For all SG-models we set the number of hops per layer to 2 and this could be the reason why SGNets perform better than GCN in many cases.

When considering the number of learnable parameters from table 3.5 we can see that the most complex model ArmaNet2 achieves the best performance, but other models with many parameters do not necessarily perform well. ArmaNet1 and TAGNet1 for example achieve very good results, but they have less than 40 learnable parameters in comparison to 149 by GCNet3. Dehmamy et al. [2019] investigate the training success based on the complexity of different models and come to the conclusion that the depth of GNN is more important than their width, so one should rather invest in adding more layers. When combining their observations that more layers are needed with our observations that the performance of the convolution-type differ, the development of GNN-architectures enabling more layers as well as more complex types of convolutions seems promising.

5. Conclusion and Outlook

In this thesis we investigate ML-methods to predict the dynamic stability of power grids using SNBS. To compute SNBS we run Monte-Carlo simulations and apply perturbations to synthetic power grid models. As ML-methods we use well known CNNs and recently developing GNNs and compare their performance. There is no literature about predicting the dynamic stability using GNNs or CNNs yet. Other attempts of investigating power grids using GNNs are only applied to less complex problems. The results of this thesis show, that the dynamic stability can be predicted using state-of-the-art ML-methods, however the accuracy is still low. Nevertheless, one can already make several conclusions from the obtained results and we start with observations from preliminary conducted investigations as part of chapter 3.

Before applying CNNs on analyzing dynamic stability, we investigate different CNN-models predicting the ability to synchronize. The ability to synchronize is a graph classification problem and less computationally expensive. It is also not really an interesting task itself, because it can be computed analytically, but we use it to generate simple datasets. In general, the training is relatively stable, meaning that varying parameters has only slight effects on the results. We investigate the influence of AlexNet and multiple ResNets, as well as changing the sample size, batch size, learning rate and applied different types of schedulers. There are limits for varying the parameters, but when staying within certain range, one gets comparable results and the performance differs only slightly. The overall performance is relatively low with its peak at 78 % to predict the ability to synchronize.

The key observation from the static test cases investigating the ability to synchronize (section 3.3.1) is a large dependency of CNN-results on the grid size. Reducing the grid dimension from 50 to 10 nodes increases the performance from roughly 60 % to 78 % when predicting the ability to synchronize. This observation is confirmed from analyzing the dynamics. When evaluating SNBS, the accuracy is improved by more than 45 % when considering grids of size 10, instead of up to 23 % for grids of size 50. One reason for the dependency on the grid size is the design of CNNs and especially its filter size. CNNs are designed to analyze pictures, where each pixel has a fixed number of neighbors and obvious spatial relations. When using $\boldsymbol{L}$ to describe the topology of power grids, we have a setup where nodes could be connected by short path lengths, but could still be far away from one another in $\boldsymbol{L}$. However, filters of CNNs are applied to regions and capture significant parts of an area around a node. When using small $\boldsymbol{L}$, the filters of CNNs might capture enough of the neighborhood of a node, but the filters fail when they are applied to large power grids.

GNNs appear to be less depending on the grid size, but there are large differences in the performance for the used GNN-models. In this thesis, we use GNN-models with 1 up to 3 convolutional layers, which is much less in comparison to 50 layers of ResNet50. However, the number of convolutions appears to be less important than the type of convolution. GNNs using Arma- and TAG-convolutions achieve the best performance (table 4.2). Adding more convolutions does not necessarily increase the performance. In most cases ArmaNet2 with three convolutional layers achieves the best results and usually outperforms ArmaNet1 which has only two convolutional layers. While adding some additional layers seems promising in case of ArmaNets, one might also be already close to the finite limit of convolutional layers after which the performance cannot be increased by simply adding more layers. Adding more layers increases the noise level, so other methods such as skip layers should be applied as well. From investigating GCNNets and SGNets we already know that more layers do not necessarily increase the performance.

When comparing the overall results, we observe that CNNs are better for grids consisting of 10 nodes and GNNs are better for grids with 50 nodes (compare tables 4.1 and 4.2). The well advanced CNNs can compensate its design limitations in case of small grids. Once GNNs close the gap to CNNs regarding the experience with good training settings and more difficult architectures of GNNs become trainable, GNNs should be able to achieve much better results. GNNs perform better than CNNs for grids with 50 nodes independently of predicting the probability or simply the class of SNBS which is done in section 4.3. Furthermore, GNNs require much less learnable parameters. While the investigated CNNs use millions of parameters, the most complex GNN that we investigate only requires about 1000 parameters. Hence, GNN provide the information in such a meaningful way that less parameters are needed for the prediction. In conclusion, GNN-methods show potential but have to be improved and further developments are necessary in order to use them for analyzing the dynamic stability.

For the future, the accuracy of GNN-methods shall be improved. One approach is based on using the knowledge from CNNs and applying strategies to enable deeper GNNs by using skip connections for example. Another important aspect is the type of convolution. As we see from our investigations different types of convolutions have a significant impact on the results. New ways of aggregating spatial information could dramatically boost the performance. Perhaps there will be different types of convolutions for different problems. Specific convolutions for the application of power grids could for example be able to take physical conditions into account. One promising approach are the integration of differential equations into GNNs, since the interaction of the components of a power grid can be described by differential equations.

Another promising aspect is the possibility of applying trained GNNs to different domains. We show the feasibility of training one model based on a grid with 10 nodes and then apply this model to a grid of 50 nodes and achieve comparable results (section 4.2.8). This idea must be tested more deeply, but it could mean that models can be trained on very small datasets to reduce the training

effort and can still be applied to real-sized grids. However, one needs to emphasize that the synthetic grids are very homogeneous, so the application to other data will be more challenging.

Once the performance of GNNs-models is enough to accurately predict the dynamic stability, those methods would open the field to a variety of applications. ML-methods can be used to quickly evaluate different types of problems such as possible configurations for future power grid topologies. One can also analyze the decision-making of the GNNs to identify parameters which are relevant for stability. Perhaps one finds previously unknown parameters which can for example be used as design parameters for future power grids.

Bibliography

FHell/EmbeddedGraphs.jl: Embedded Graphs for Julia. URL https://github.com/FHell/EmbeddedGraphs.jl.

julia-License. URL https://github.com/JuliaLang/julia/blob/master/LICENSE.md.

networkx/networkx: Official NetworkX source code repository. URL https://github.com/networkx/networkx.

luap-pik-tree-node-classification. URL https://github.com/luap-pik/tree-node-classification.

luap-pik/SyntheticNetworks. URL https://github.com/luap-pik/SyntheticNetworks.

ImageNet Large Scale Visual Recognition Competition 2012 (ILSVRC2012), 2012. URL http://image-net.org/challenges/LSVRC/2012/results.

ILSVRC2015 Results, 2015. URL http://image-net.org/challenges/LSVRC/2015/results.

Annual Renewable Energy Constraint and Curtailment Report 2017. Technical report, Eirgrid Group, 2017. URL http://www.eirgridgroup.com/Annual-Renewable-Constraint-and-Curtailment-Report-2017-V1.pdf.

Annual Renewable Energy Constraint and Curtailment Report 2018. Technical report, Eirgrid Group, 2018. URL http://www.eirgridgroup.com/site-files/library/EirGrid/Annual-Renewable-Constraint-and-Curtailment-Report-2018-V1.0.pdf.

J. Atwood and D. Towsley. Diffusion-convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2016.

A. Bavelas. Communication Patterns in Task-Oriented Groups. *Journal of the Acoustical Society of America*, 1950. ISSN NA. doi: 10.1121/1.1906679.

Y. Bengio. Practical recommendations for gradient-based training of deep architectures. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2012. ISSN 03029743. doi: 10.1007/978-3-642-35289-8-26.

Y. Bengio, P. Simard, and P. Frasconi. Learning Long-Term Dependencies with Gradient Descent is Difficult. *IEEE Transactions on Neural Networks*, 1994. ISSN 19410093. doi: 10.1109/72.279181.

F. M. Bianchi, D. Grattarola, C. Alippi, and L. Livi. Graph Neural Networks with convolutional ARMA filters. jan 2019. URL http://arxiv.org/abs/1901.01343.

L. Bottou. Online Learning and Stochastic Approximations. 1998. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.52.4918.

U. Brandes and D. Fleischer. Centrality measures based on current flow. In *Lecture Notes in Computer Science*, 2005. doi: 10.1007/978-3-540-31856-9_44.

M. M. Bronstein, J. Bruna, Y. Lecun, A. Szlam, and P. Vandergheynst. Geometric Deep Learning: Going beyond Euclidean data, 2017. ISSN 10535888.

J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. Spectral networks and deep locally connected networks on graphs. In *2nd International Conference on Learning Representations, ICLR 2014 - Conference Track Proceedings*, 2014.

J. Chen, T. Ma, and C. Xiao. FastGCN: Fast learning with graph convolu-tional networks via importance sampling. In *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, 2018.

M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, 2016.

N. Dehmamy, A.-L. Barabási, and R. Yu. Understanding the Representation Power of Graph Neural Networks in Learning Graph Topology. jul 2019. URL http://arxiv.org/abs/1907.05008.

O. Dekel, R. Gilad-Bachrach, O. Shamir, and L. Xiao. Optimal distributed online prediction using mini-batches. *Journal of Machine Learning Research*, 2012. ISSN 15324435.

B. Donon, B. Donnot, I. Guyon, and A. Marot. Graph Neural Solver for Power Systems. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2019-July. Institute of Electrical and Electronics Engineers Inc., jul 2019. ISBN 9781728119854. doi: 10.1109/IJCNN.2019.8851855.

F. Dörfler, M. Chertkov, and F. Bullo. Synchronization in complex oscillator networks and smart grids. *Proceedings of the National Academy of Sciences of the United States of America*, 2013. ISSN 00278424. doi: 10.1073/pnas.1212134110.

J. Du, S. Zhang, G. Wu, J. M. F. Moura, and S. Kar. Topology Adaptive Graph Convolutional Networks. oct 2017. URL http://arxiv.org/abs/1710.10370.

M. Fey and J. E. Lenssen. Fast Graph Representation Learning with PyTorch Geometric. mar 2019. URL http://arxiv.org/abs/1903.02428.

L. C. Freeman. A Set of Measures of Centrality Based on Betweenness. *Sociometry*, 1977. ISSN 00380431. doi: 10.2307/3033543.

J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural Message Passing for Quantum Chemistry, jul 2017. ISSN 1938-7228. URL `http://proceedings.mlr.press/v70/gilmer17a.html`.

I. J. Goodfellow, O. Vinyals, and A. M. Saxe. Qualitatively characterizing neural network optimization problems. dec 2014. URL `http://arxiv.org/abs/1412.6544`.

M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. In *Proceedings of the International Joint Conference on Neural Networks*, 2005. ISBN 0780390482. doi: 10.1109/IJCNN.2005.1555942.

R. H. Hahnloser, R. Sarpeshkar, M. A. Mahowald, R. J. Douglas, and H. S. Seung. Digital selection and analogue amplification coexist in a cortex- inspired silicon circuit. *Nature*, 2000. ISSN 00280836. doi: 10.1038/35016072.

W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, 2017.

K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision*, 2015. ISBN 9781467383912. doi: 10.1109/ICCV.2015.123.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2016. ISBN 9781467388504. doi: 10.1109/CVPR.2016.90.

S. Hershey, S. Chaudhuri, D. P. Ellis, J. F. Gemmeke, A. Jansen, R. C. Moore, M. Plakal, D. Platt, R. A. Saurous, B. Seybold, M. Slaney, R. J. Weiss, and K. Wilson. CNN architectures for large-scale audio classification. In *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, 2017. ISBN 9781509041176. doi: 10.1109/ICASSP.2017.7952132.

G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. jul 2012. URL `http://arxiv.org/abs/1207.0580`.

G. E. Hinton, A. Krizhevsky, I. Sutskever, and N. Srivastava. SYSTEM AND METHOD FOR ADDRESSING OVERFITTING IN A NEURAL NETWORK, 2016.

G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, 2017. ISBN 9781538604571. doi: 10.1109/CVPR.2017.243.

S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *32nd International Conference on Machine Learning, ICML 2015*, 2015. ISBN 9781510810587.

K. Janocha and W. M. Czarnecki. On Loss Functions for Deep Neural Networks in Classification. feb 2017. URL http://arxiv.org/abs/1702.05659.

K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun. What is the best multi-stage architecture for object recognition? BT - Computer Vision, 2009 IEEE 12th International Conference on. In *Computer Vision, 2009 . . .*, 2009. ISBN 9781424444199.

K. Kawaguchi and L. P. Kaelbling. Elimination of All Bad Local Minima in Deep Learning. jan 2019. URL http://arxiv.org/abs/1901.00279.

C. Kim, K. Kim, P. Balaprakash, and M. Anitescu. Graph Convolutional Neural Networks for Optimal Load Shedding under Line Contingency. *Proceedings in IEEE Power Engineering Society General Meeting (to appear, preprint)*, 2019. URL https://kibaekkim.github.io/papers/PES-GCN-preprint.pdf.

D. P. Kingma and J. L. Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, 2015.

T. N. Kipf and M. Welling. Semi-Supervised Classification with Graph Convolutional Networks. sep 2016. URL http://arxiv.org/abs/1609.02907.

G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter. Self-Normalizing Neural Networks. jun 2017. URL http://arxiv.org/abs/1706.02515.

A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. apr 2014. URL http://arxiv.org/abs/1404.5997.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2012. ISBN 9781627480031.

A. Krogh and J. a. Hertz. A Simple Weight Decay Can Improve Generalization. *Advances in Neural Information Processing Systems*, 1992.

Y. Kuramoto. Self-entrainment of a population of coupled non-linear oscillators. In *International Symposium on Mathematical Problems in Theoretical Physics*. 2005. doi: 10.1007/bfb0013365.

S. Leng, W. Lin, and J. Kurths. Basin stability in delayed dynamics. *Scientific Reports*, 2016. ISSN 20452322. doi: 10.1038/srep21449.

M. Li, T. Zhang, Y. Chen, and A. J. Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 661–670. Association for Computing Machinery, 2014. ISBN 9781450329569. doi: 10.1145/2623330.2623612.

Y. Li, R. Zemel, M. Brockschmidt, and D. Tarlow. Gated graph sequence neural networks. In *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, 2016.

Y. Li, R. Yu, C. Shahabi, and Y. Liu. Diffusion convolutional recurrent neural network: Data-driven traffic forecasting. In *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, 2018.

S. Liu, D. Papailiopoulos, and D. Achlioptas. Bad Global Minima Exist and SGD Can Reach Them. jun 2019. URL `http://arxiv.org/abs/1906.02613`.

A. L. Maas, A. Y. Hannun, and A. Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013.

P. J. Menck, J. Heitzig, N. Marwan, and J. Kurths. How basin stability complements the linear-stability paradigm. *Nature Physics*, 2013. ISSN 17452481. doi: 10.1038/nphys2516.

A. Micheli. Neural network for graphs: A contextual constructive approach. *IEEE Transactions on Neural Networks*, 2009. ISSN 10459227. doi: 10.1109/TNN.2008.2010350.

P. Milan, M. Wächter, and J. Peinke. Turbulent character of wind energy. *Physical Review Letters*, 2013. ISSN 00319007. doi: 10.1103/PhysRevLett.110.138701.

F. Monti, D. Boscaini, J. Masci, E. Rodolà, J. Svoboda, and M. M. Bronstein. Geometric deep learning on graphs and manifolds using mixture model CNNs. In *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, 2017. ISBN 9781538604571. doi: 10.1109/CVPR.2017.576.

K. Nakamura and B.-W. Hong. Adaptive Weight Decay for Deep Neural Networks. *IEEE Access*, 2019. doi: 10.1109/access.2019.2937139.

Y. Nesterov. A method for solving the convex programming problem with convergence rate O(1/k^2), 1983.

Y. Nesterov. Introductory Lectures on Convex Programming Volume I: Basic course. *Lecture Notes*, 1998. doi: 10.1007/978-1-4419-8853-9.

M. Newman. *Networks: An Introduction*. 2010. ISBN 9780191594175. doi: 10.1093/acprof:oso/9780199206650.001.0001.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 2014. doi: 10.1016/0370-2693(93)90272-J.

J. Nitzbon, P. Schultz, J. Heitzig, J. Kurths, and F. Hellmann. Deciphering the imprint of topology on nonlinear dynamical network stability. *New Journal of Physics*, 2017. ISSN 13672630. doi: 10.1088/1367-2630/aa6321.

J. P. Onnela, J. Saramäki, J. Kertész, and K. Kaski. Intensity and coherence of motifs in weighted complex networks. *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics*, 2005. ISSN 15393755. doi: 10.1103/PhysRevE.71.065103.

J. O'Sullivan, A. Rogers, D. Flynn, P. Smith, A. Mullane, and M. O'Malley. Studying the maximum instantaneous non-synchronous generation in an Island system-frequency stability challenges in Ireland. *IEEE Transactions on Power Systems*, 2014. ISSN 08858950. doi: 10.1109/TPWRS.2014.2316974.

D. Owerko, F. Gama, and A. Ribeiro. Optimal Power Flow Using Graph Neural Networks. oct 2019. URL `http://arxiv.org/abs/1910.09658`.

A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury Google, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. K. Xamla, E. Yang, Z. Devito, M. Raison Nabla, A. Tejani, S. Chilamkurthy, Q. Ai, B. Steiner, L. F. Facebook, J. B. Facebook, and S. Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. Technical report, 2019.

N. Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 1999. ISSN 08936080. doi: 10.1016/S0893-6080(98)00116-6.

C. Rackauckas and Q. Nie. DifferentialEquations.jl A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia. *Journal of Open Research Software*, 2017. ISSN 2049-9647. doi: 10.5334/jors.151.

D. E. Rumelhart, J. L. McClelland, and C. Asanuma. *Parallel Distributed Processing: Foundations*. 1986. ISBN 9780262181204.

P. Schultz, J. Heitzig, and J. Kurths. A random growth model for power grids and other spatially embedded infrastructure networks. *European Physical Journal: Special Topics*, 2014a. ISSN 19516401. doi: 10.1140/epjst/e2014-02279-6.

P. Schultz, J. Heitzig, and J. Kurths. Detours around basin stability in power networks. *New Journal of Physics*, 2014b. ISSN 13672630. doi: 10.1088/1367-2630/16/12/125001.

S. Seong, Y. Lee, Y. Kee, D. Han, and J. Kim. Towards flatter loss surface via nonmonotonic learning rate scheduling. In *34th Conference on Uncertainty in Artificial Intelligence 2018, UAI 2018*, 2018. ISBN 9781510871601.

D. I. Shuman, S. K. Narang, P. Frossard, A. Ortega, and P. Vandergheynst. The emerging field of signal processing on graphs. *IEEE Signal Processing Magazine*, 2013. ISSN 10535888. doi: 10.1109/MSP.2012.2235192.

L. N. Smith. Cyclical learning rates for training neural networks. In *Proceedings - 2017 IEEE Winter Conference on Applications of Computer Vision, WACV 2017*, 2017. ISBN 9781509048229. doi: 10.1109/WACV.2017.58.

R. K. Srivastava, K. Greff, and J. Schmidhuber. Highway Networks. may 2015a. URL `http://arxiv.org/abs/1505.00387`.

R. K. Srivastava, K. Greff, and J. Schmidhuber. Training very deep networks. In *Advances in Neural Information Processing Systems*, 2015b.

I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *30th International Conference on Machine Learning, ICML 2013*, 2013.

United Nations. PARIS AGREEMENT, 2015. URL `https://unfccc.int/sites/default/files/english{_}paris{_}agreement.pdf`.

P. Veličković, A. Casanova, P. Liò, G. Cucurull, A. Romero, and Y. Bengio. Graph attention networks. In *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, 2018.

J. Wang, W. Wang, and N. Srebro. Memory and Communication Efficient Distributed Stochastic Optimization with Minibatch-Prox. In *Proceedings of Machine Learning Research*, volume 65, feb 2017. URL `http://arxiv.org/abs/1702.06269`.

P. Werbos. Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. *PhD Thesis, Harvard U.*, 1974. doi: 10.1.1.41.8085.

F. Wu, T. Zhang, A. H. de Souza, C. Fifty, T. Yu, and K. Q. Weinberger. Simplifying Graph Convolutional Networks. feb 2019a. URL `http://arxiv.org/abs/1902.07153`.

Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A Comprehensive Survey on Graph Neural Networks. jan 2019b. URL `http://arxiv.org/abs/1901.00596`.

K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How Powerful are Graph Neural Networks? oct 2018. URL `http://arxiv.org/abs/1810.00826`.

R. Yedida and S. Saha. A novel adaptive learning rate scheduler for deep neural networks. feb 2019. URL `http://arxiv.org/abs/1902.07399`.

K. You, M. Long, J. Wang, and M. I. Jordan. How Does Learning Rate Decay Help Modern Neural Networks? aug 2019. URL `http://arxiv.org/abs/1908.01878`.

S. Zhang, H. Tong, J. Xu, and R. Maciejewski. Graph convolutional networks: a comprehensive review. *Computational Social Networks*, 2019. ISSN 21974314. doi: 10.1186/s40649-019-0069-y. URL `https://link.springer.com/article/10.1186/s40649-019-0069-y`.

J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun. Graph Neural Networks: A Review of Methods and Applications. dec 2018. URL `http://arxiv.org/abs/1812.08434`.

C. Zhuang and Q. Ma. Dual Graph Convolutional Networks for Graph-Based Semi-Supervised Classification. 2018. doi: 10.1145/3178876.3186116.

A. Additional information on the fundamentals and state-of-the-art

This chapter includes information about the fundamentals that extend the covered topics in chapter 2. The first section is about ANNs in general, and the second about GNNs in particular.

A.1. ANN

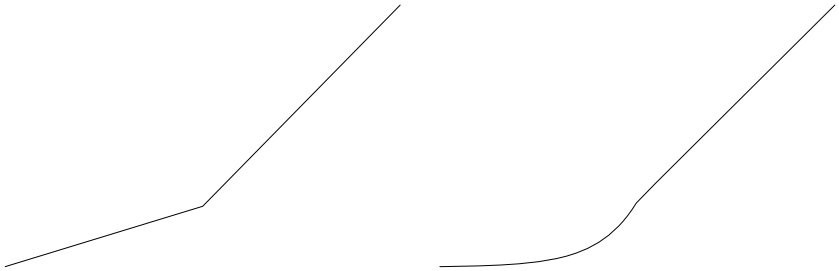
We show additional information on components of ANNs as well as information about the training.

A.1.1. Further information on components of ANNs

One important component of ANNs are the activation functions, so we consider the ReLU activation function in more detail here.

Activation function: ReLU and its problems

To overcome the problem with negative dying neurons activation functions based on ReLU are introduced and shown in fig. A.1 and table A.1. Instead of setting all negative values to zero, a liner function with slope $a > 0$ is used. For $a = .01$, the method is called Leaky ReLU and is introduced by Maas et al. [2013]. He et al. [2015] introduce Parametric Rectified Linear Unit(PReLU), which learns the slope parameter as part of the training problem. The details are given in table 2.1. To increase the robustness and to train deep networks with multiple layers scaled exponential linear units" (SELUs) are introduced by Klambauer et al. [2017]. The idea is to conserve the mean and variance of the output across the layers. As long as the input x is within the desired intervals, y should remain in them too. Other techniques of normalization are given in section 2.3.3.



(a) ReLU with slope for $x < 0$ (b) SeLU

Figure A.1.: Modified ReLU functions to tackle the problem of dying neurons

name	equation	output range
ReLU with slope	$\begin{cases} ax, & \text{if } x < 0, a > 0 \\ x, & \text{otherwise} \end{cases}$	$(-\infty, \infty)$
SeLU	$\lambda \begin{cases} x, & \text{if } x > 0 \\ \alpha e^x - \alpha, & \text{otherwise} \end{cases}$ $\alpha = 1.67326, \lambda = 1.0507$	$(-\alpha, \infty)$

Table A.1.: Activation functions

A.1.2. Additional concepts of choosing the learning rate

Smith [2017] introduces a cyclically varying learning rate between specified boundaries and Yedida and Saha [2019] introduce a method to compute the learning rate based on the Lipschitz constant of the loss function. Another motivation is to set the learning rate to flatten the loss surface to increase the robustness of the optimization. Seong et al. [2018] introduce a method of computing the learning rate to avoid sharp regions of loss surfaces.

A.1.3. Additional concepts of applying momentum

Based on the work by Nesterov [1983, 1998] on optimization, convergence rates and convex programing, Sutskever et al. [2013] introduce the update based on Nesterovs accelerated gradient which is often referred to by Sutskever Nesterov Momentum. The difference lies in the evaluation of the gradient, which is evaluated at an indermediate position in between $\mathcal{P}_{t-1}$ and $\mathcal{P}_t$ instead of evaluating at $\mathcal{P}_t$. On the contrary to the previously introduced gradients $\nabla f_{\mathcal{L}} = \nabla f_{\mathcal{L}}(\mathcal{P}_t)$, we have $\nabla f_{\mathcal{L}}(\mathcal{P}_t + p\Delta\mathcal{P}_{t-1})$ and get the following update rule:

$$\Delta\mathcal{P}_t = p\Delta\mathcal{P}_{t-1} - \eta\nabla f_{\mathcal{L}}(\mathcal{P} + p\Delta\mathcal{P}_{t-1})). \tag{A.1}$$

Again, in some applications the multiplication of the learning rate is applied as shown in eq. (2.51).

A.2. GNN

The following architectures are not investigated in this thesis, so they are only listed in the appendix.

Message Passing Neural Network

In 2017 Gilmer et al. [2017] introduce Message Passing Neural Network (MPNNs) which scheme consists of two message functions $\mathcal{M}$:

$$\mathcal{M}_v^k = \sum_{u \in \mathcal{N}(v)} M_k(h_v^{(k-1)}, h_u^{(k-1)}, X_{vu}^e), \tag{A.2}$$

and vertex update functions U:

$$H_v^{(k)} = U_k(H_v^{(k-1)}, \mathcal{M}^k) \tag{A.3}$$

where $\mathcal{M}$ is the message, t the time step, $\mathcal{N}(v)$ are the neighbors of vertex v, X_{vu}^e is the edge feature vector of edge (v,u) and H_v is a vector containing the hidden states at node v and M and U are learned differentiable functions. Lastly, there is a readout function $\mathcal{R}$:

$$Y = R(H_v^T | v \in \mathcal{G}), \tag{A.4}$$

to obtain feature vector for the entire graph. When choosing $\mathcal{R}, M, U$ appropriately many convolution types such as Kipf and Welling [2016] are covered by this formalization.

GraphSAGE

Hamilton et al. [2017] introduce an inductive framework called GraphSAGE where the convolution consists of three steps:

1. Aggregation information from node's neighborhood
2. Concatenation of the representation of the aggregated information
3. Feeding result to fully connected layer with an activation function

leading to the following scheme:

$$\begin{aligned} H_{\mathcal{N}(v)}^{(k-1)} &= AGGREGATE_{k-1}(H_u^{(k-1)}, \forall u \in \mathcal{N}(v)) \\ H_v^{(k)} &= \sigma\left(CONCAT(\boldsymbol{X}^{(k-1)}(v,:), H_{\mathcal{N}(v)}^{(k-1)})\boldsymbol{\Theta}^{(k-1)}\right), \end{aligned} \tag{A.5}$$

where $AGGREGATE$ is a learnable function and possible functions can be mean or pooling aggregation. The second clue of GraphSAGE is to sample a fixed number of neighbor nodes. When

using this framework one does not have to know the entire graph in advance, because one splits the graph in mutliple mini-batches.

Graph Isomorphism Network

Xu et al. [2018] show that GCN and GraphSAGE cannot learn to distinguish certain simple graph structures, so they develop an architecture called Graph Isomorphism Network (GIN). GIN adds an additional learnable scalar parameter τ to the convolutional layer:

$$h_v^{(k)} = \sigma\left((1+\tau^{(k)})H_v^{(k-1)} + \sum_{u\in\mathcal{N}(v)} h_u^{(k-1)}\boldsymbol{W}^{(k-1)}\right) \tag{A.6}$$

Graph Attention Network

Graph Attention Network (GAT) by Veličković et al. [2018] enable to assign different importances to nodes of a same neighborhood and there is no need of knowing the full graph in advance, on the contrary to GCN. GAT use an attention coefficient κ between nodes, which indicates the importance of the features of one node to the other node and is computed by:

$$\kappa_{vu} = \sigma_1\left(\sigma_2\left(\theta^T[\boldsymbol{W}^{(k-1)}H_v||\boldsymbol{W}^{(k-1)}H_u]\right)\right), \tag{A.7}$$

where σ_1 is a Sigmoid function and σ_2 is a Leaky ReLU. The convolution is given by:

$$H_v^{(k)} = \sigma\left(\sum_{u\in\mathcal{N}(v)\cup v} \kappa_{vu}\boldsymbol{W}^{(k-1)}H_u^{(k-1)}\right) \tag{A.8}$$

GAT reduces the computational effort by avoid matrix-matrix multiplications and can also be parallelized across all nodes.

FastGCN

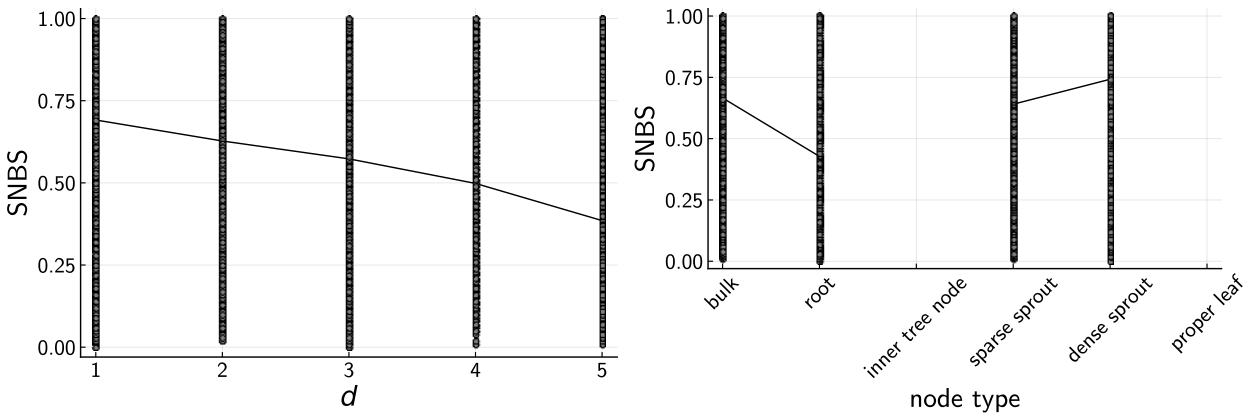
FastGCN by Chen et al. [2018] reduce the computational costs by sampling over the nodes and only use a subset of all nodes in each convolutional layer, which is a difference to GraphSAGE which sample over neighbors. Using FastGCN is of magnitudes faster than GCN or GraphSAGE and the prediction performance is comparable.

B. Additional information on the used methods

This chapter extends the methods from chapter 3. Before introducing the ML-methods, we show detailed information of the used datasets.

B.1. Additional information on datasets

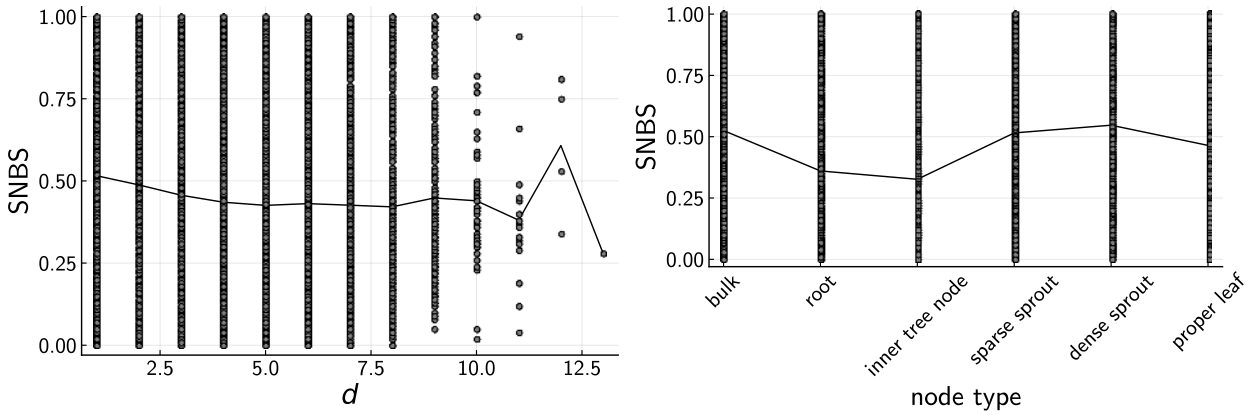
Figures B.1 to B.5 show additional information on the datasets. We can clearly see that AvBS varies with the degree d and different type of nodes.



(a) SNBS depending on the degree d and the solid line represents AvBS

(b) SNBS for different node types and the solid line shows AvBS

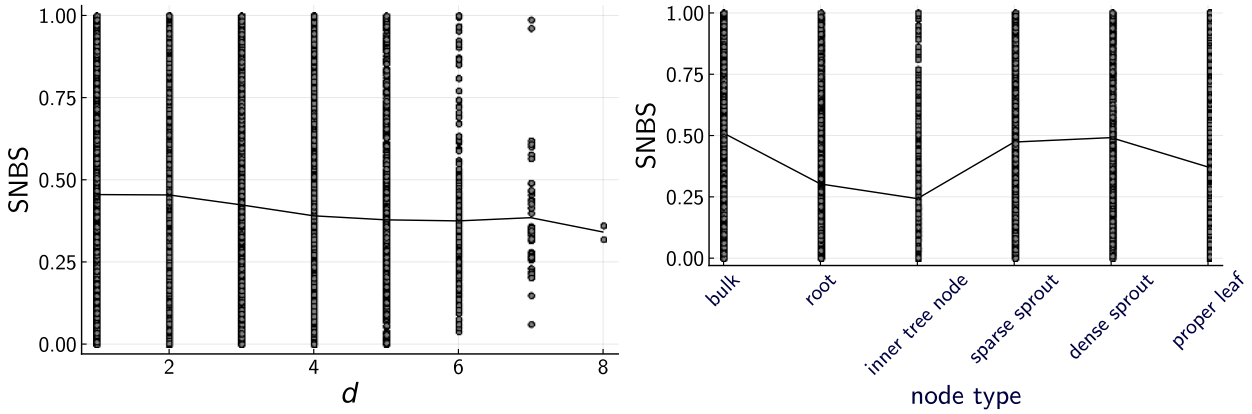
Figure B.1.: SNBS of fGN10 (fixed graph with 10 nodes)



(a) SNBS depending on the degree d and the solid line represents AvBS

(b) SNBS for different node types and the solid line shows AvBS

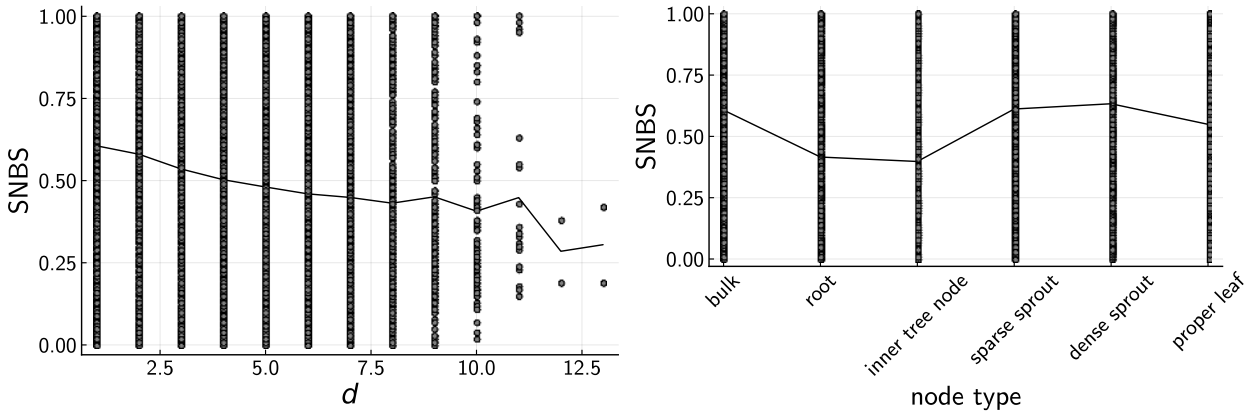
Figure B.2.: SNBS of P1N50



(a) SNBS depending on the degree d and the solid line represents AvBS

(b) SNBS for different node types and the solid line shows AvBS

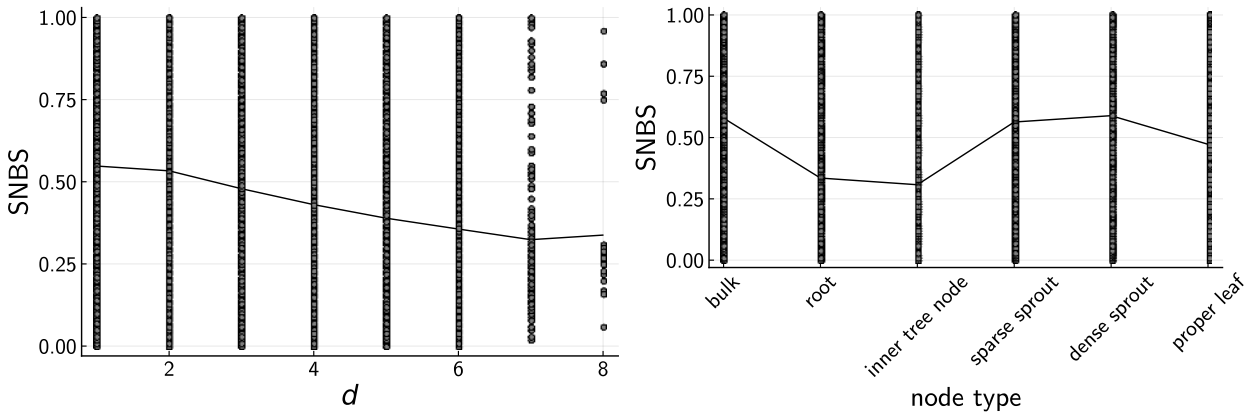
Figure B.3.: SNBS of P1N10



(a) SNBS depending on the degree d and the solid line represents AvBS

(b) SNBS for different node types and the solid line shows AvBS

Figure B.4.: SNBS of PrandN50



(a) SNBS depending on the degree d and the solid line represents AvBS

(b) SNBS for different node types and the solid line shows AvBS

Figure B.5.: SNBS of PrandN10

B.2. AlexNet

We make some modifications from the default AlexNet configuration of Pytorch, so we show the setup of the sequential layers:

```
nn.Conv2d(1, 64, kernel_size=2, stride=1, padding=1),
nn.ReLU(inplace=True),
nn.MaxPool2d(kernel_size=3, stride=2),
nn.Conv2d(64, 192, kernel_size=5, padding=2),
nn.ReLU(inplace=True),
nn.MaxPool2d(kernel_size=3, stride=2),
nn.Conv2d(192, 384, kernel_size=3, padding=1),
nn.ReLU(inplace=True),
nn.Conv2d(384, 256, kernel_size=3, padding=1),
nn.ReLU(inplace=True),
nn.Conv2d(256, 256, kernel_size=3, padding=1),
nn.ReLU(inplace=True),
nn.MaxPool2d(kernel_size=2, stride=2),
)
```

B.3. ResNet

For ResNet we change the number of classes (num_classes) to 1 and also the number of input channels for the first convolution conv2d to the number of *num_node_features* instead of 3. So two changes are applied to the class ResNet and we show the first lines of class ResNet, where we applied our changes.

```
class ResNet(nn.Module):
    def __init__(self, block, layers, num_classes=1,
    ↪  zero_init_residual=False,
                 groups=1, width_per_group=64,
                 ↪  replace_stride_with_dilation=None,
                 norm_layer=None, num_node_features = 1):
        super(ResNet, self).__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        self._norm_layer = norm_layer
```

```
        self.inplanes = 64
        self.dilation = 1
        if replace_stride_with_dilation is None:
            # each element in the tuple indicates if we should replace
            # the 2x2 stride with a dilated convolution instead
            replace_stride_with_dilation = [False, False, False]
        if len(replace_stride_with_dilation) != 3:
            raise ValueError("replace_stride_with_dilation should be None "
                             "or a 3-element tuple, got
                             ↪  {}".format(replace_stride_with_dilation))
        self.groups = groups
        self.base_width = width_per_group
        self.conv1 = nn.Conv2d(num_node_features, self.inplanes,
        ↪  kernel_size=7, stride=2, padding=3,
                               bias=False)
        self.bn1 = norm_layer(self.inplanes)
        ...
```

The models in this thesis have the following number of learnable parameters:

- ResNet18: 11170753
- ResNet34: 21278913
- ResNet50: 23503809

B.4. Training settings

The detailed training information are given in the following.

B.4.1. Default settings using convolutional neural networks and training of static test cases

We use SGD for training with a momentum of 0.9 if not mentioned differently. All CNNs are pretrained. Pretrained means that we reuse proper weights that are appropriate for other applications. In case we make changes to the architecture we copy all possible parameters, so that the used networks have as many pretrained values as possible.

Training of static test case 1

In case we use stepLR, the following parameters are used if not mentioned differently:

- step_size=2
- gamma =0.1

And in case of ReduceLROnPlateau we use the following settings:

- mode='min'
- factor=0.1
- patience=4
- min_lr=0.0001

Training of static test case 2

The same settings are used as for static test case 1 shown in appendix B.4.1. The LR is set to 0.001 and we do not use any schedulers. The CNN-architecture is ResNet18.

Visual analysis of static test case 1

Pictures of the graphs as shown in fig. 3.10 are used as inputs to CNNs to predict the ability to synchronize. For ResNet 50 and 150 epochs the training process is shown in fig. B.6. The losses decrease only for the first epochs. Afterwards we overfit, because only the training loss decreases, but there is no improvement in the validation loss. The accuracy fluctuates and is in general relatively low. When testing the models we also do not obtain convincing results as shown in fig. B.7. As assumed from the training process, the losses using the validation and testing set only decrease for the first epochs. Afterwards the losses increase. The accuracy fluctuates too strongly and does not show any significant improvements. So we draw the conclusions that training of CNNs appears to be difficult based on pictures and that we do not obtain satisfying results. We use the following properties for the training:

- We use Adam optimization with ReduceLROnPlateau scheduling with the parameters from appendix B.4.1, but a learning rate of 0.0000001.
- We use a batch size of 2.
- The inputs are pictures of size 756×756.

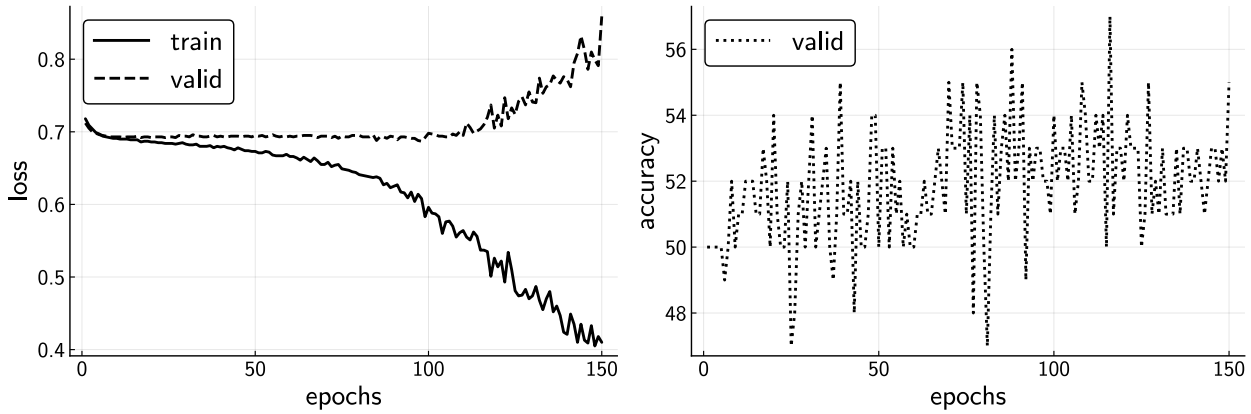


Figure B.6.: Training progress of static test case 1 using plots of power grids as input for ResNet50

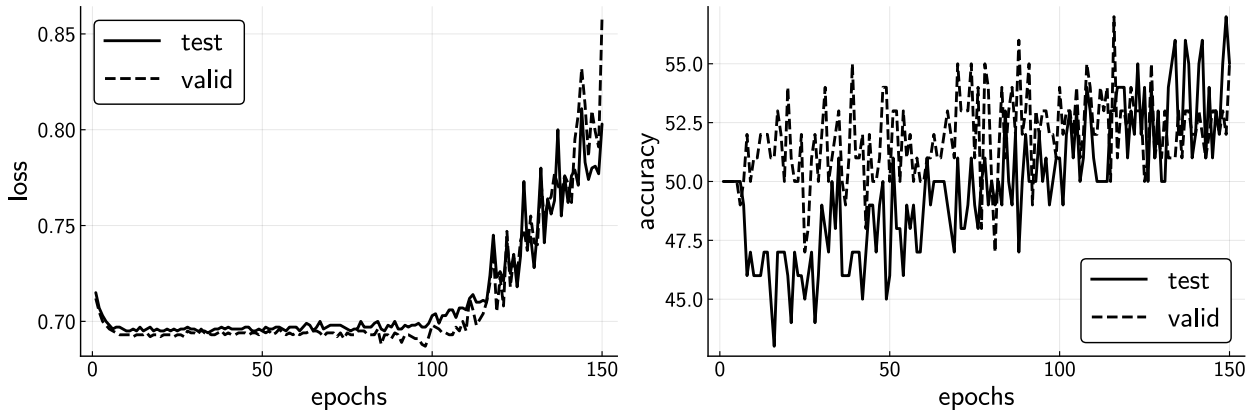


Figure B.7.: Testing of ResNet50-models using static test case 1 where plots of power grids are used as inputs

B.4.2. Training of dynamical stability using convolutional neural networks

The details for the trainings are given for each dataset in the following.

fGN50

The following parameters are used:

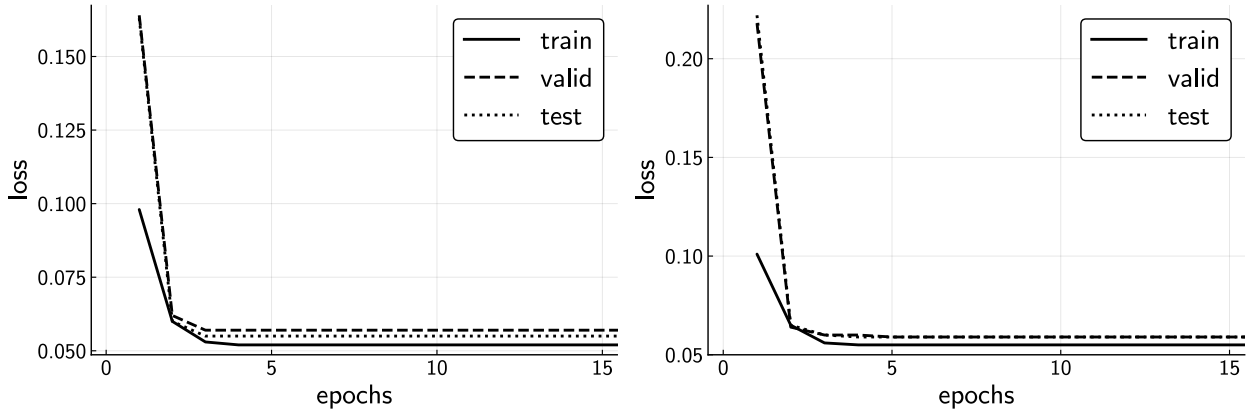
- SGD optimizer with LR=0.3 and momentum=0.9
- BS = 200
- scheduler: ReduceLROnPlateau
- manualSeed = 1

fGN10

The following parameters are used:

- SGD optimizer with LR=0.3 and momentum=0.9
- BS = 200
- scheduler: ReduceLROnPlateau
- manualSeed = 1

fig. B.8 shows the training process.



(a) ResNet18 (b) ResNet34

Figure B.8.: Training progress using CNNs and the dataset fGN10

P1N50

The following parameters are used:

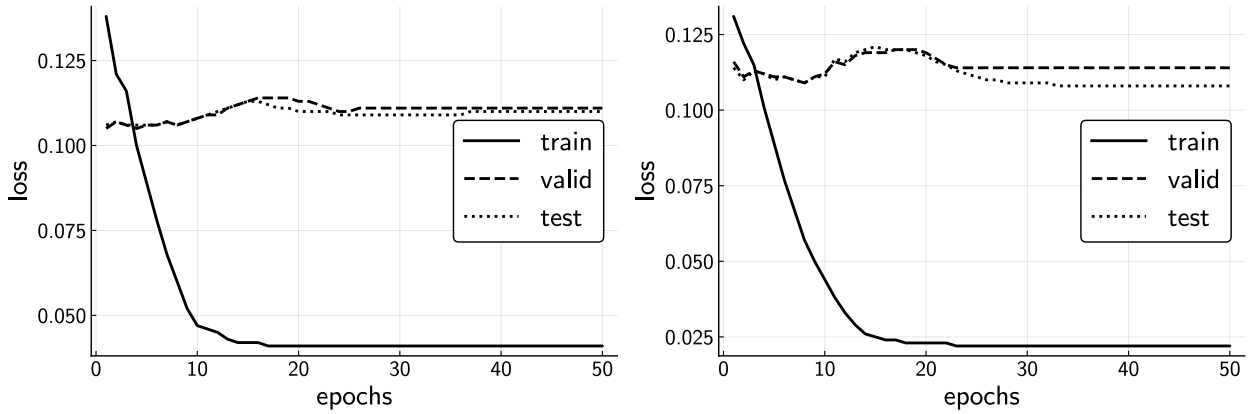
- SGD optimizer with LR=0.3 and momentum=0.9
- BS = 200
- scheduler: ReduceLROnPlateau
- manualSeed = 1

fig. B.9 shows the training process.

P1N10

The following parameters are used:

- SGD optimizer with LR=0.3 and momentum=0.9
- BS = 200
- scheduler: ReduceLROnPlateau
- manualSeed = 1



(a) ResNet18 (b) ResNet34

Figure B.9.: Training progress using CNNs and the dataset P1N50

PrandN50

The following parameters are used:

- SGD optimizer with LR=0.3 and momentum=0.9
- BS = 210
- scheduler: ReduceLROnPlateau
- manualSeed = 1

PrandN10

The following parameters are used:

- SGD optimizer with LR=0.3 and momentum=0.9
- BS = 200
- scheduler: ReduceLROnPlateau
- manualSeed = 1

B.4.3. Training settings using graph neural networks

The training settings are sorted by the datasets.

Training using fGN50 and one node feature as input

When investigating fGN50 the following parameters are used:

- SGD optimizer with LR=0.3 and momentum=0.9,
- BS = 200,
- scheduler: ReduceLROnPlateau,
- manualSeed = 1,

except for the SGNet3 which use:

- manualSeed = 7.

As an example, we show the training process for SG03 in fig. B.10.

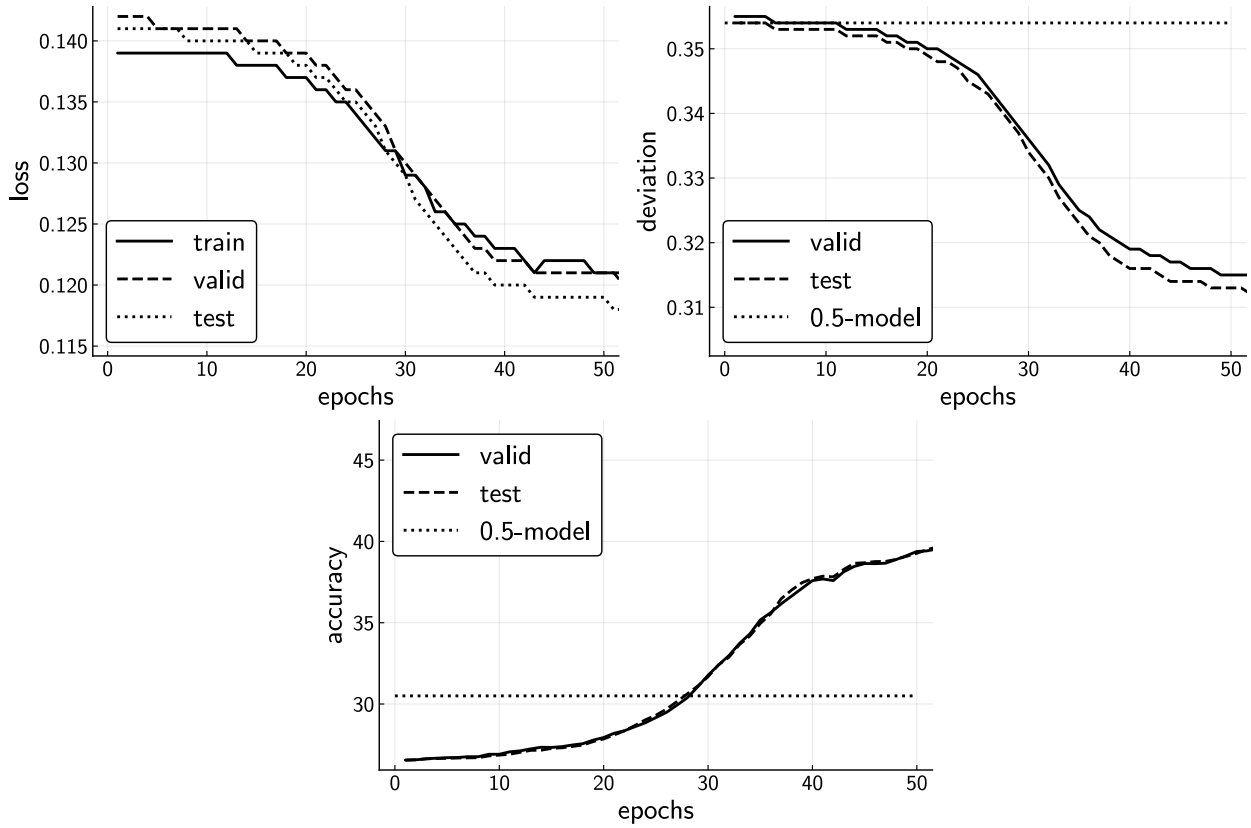


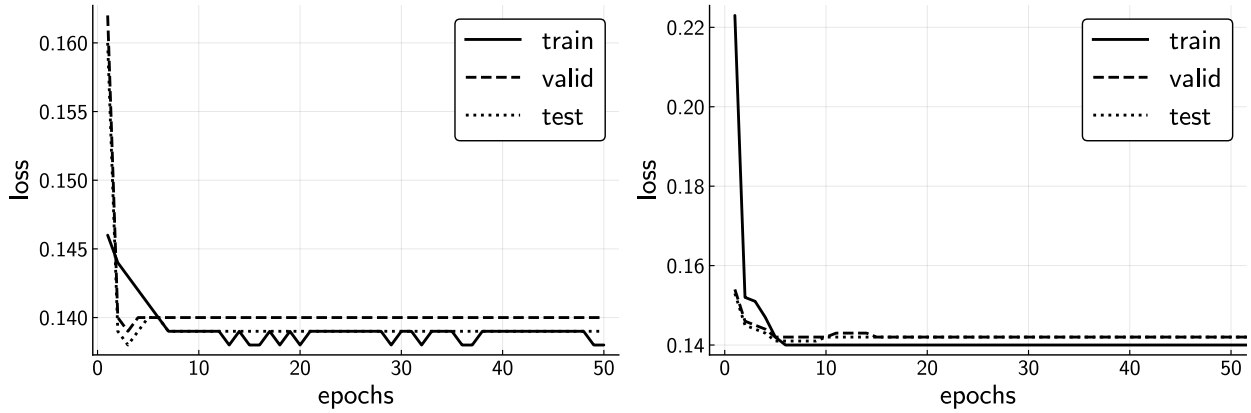
Figure B.10.: Training progress using SG03 and one node feature as input for fGN50

Training using fGN50 and six node features as input

Tthe following parameters are used:

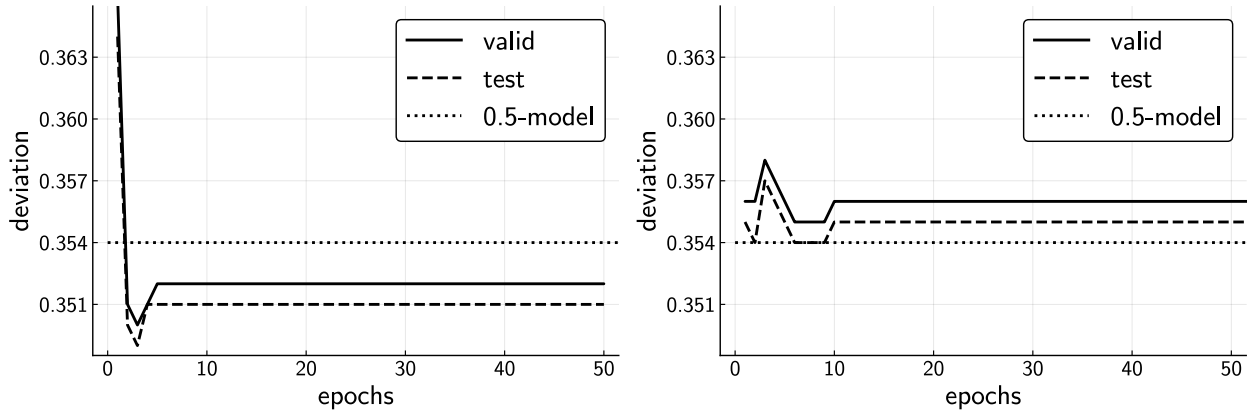
- SGD optimizer with LR=0.3 and momentum=0.9,
- BS = 200,
- scheduler: ReduceLROnPlateau,
- manualSeed = 1.

We do not see much training progress shown in fig. B.11 and the deviation does not decrease very much fig. B.12. In case of GCNNet2, the deviation is even above the .5-model.



(a) ArmaNet2 (b) GCNNet2

Figure B.11.: Training loss of fGN50 when providing six node features as input



(a) ArmaNet2 (b) GCNNet2

Figure B.12.: Deviation of fGN50 when providing six node features as input

Training using fGN10

Tthe following parameters are used:

- SGD optimizer with LR=0.3 and momentum=0.9,
- BS = 200,
- scheduler: ReduceLROnPlateau,
- manualSeed = 2.

Training using P1N50 one node feature as input

Tthe following parameters are used:

- SGD optimizer with LR=0.3 and momentum=0.9,
- BS = 200,
- scheduler: ReduceLROnPlateau,
- manualSeed = 2.

Training using PrandN50 one and multiple node features as input

Tthe following parameters are used:

- SGD optimizer with LR=0.3 and momentum=0.9,
- BS = 210,
- scheduler: ReduceLROnPlateau,
- manualSeed = 2.

Training using PrandN10 one node features as input

Tthe following parameters are used:

- SGD optimizer with LR=0.3 and momentum=0.9,
- BS = 200,
- scheduler: ReduceLROnPlateau,
- manualSeed = 2.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Masterarbeit selbständig angefertigt habe. Es wurden nur die in der Arbeit ausdrücklich benannten Quellen und Hilfsmittel benutzt. Wörtlich oder sinngemäSS übernommenes Gedankengut habe ich als solches kenntlich gemacht.

Aachen, April 2020

Unterschrift des Verfassers