Carl von Ossietzky University Oldenburg
Faculty V: Mathematics and Science

Neural Networks for Graph-Structured Time-Series with Application to Seismic Data

Master Thesis
by
Jonas Moritz Zender

1st Supervisor: **Professor Dr. Nils Strodthoff**
2nd Supervisor: **M.Sc. Tiezhi Wang**

Carl von Ossietzky University Oldenburg
Faculty V: Mathematics and Science

Neural Networks for Graph-Structured Time-Series with Application to Seismic Data

A Thesis Submitted in
Partial Fullfillment of the
Requirements of the Degree of

Master of Science
in
Environmental Modeling

by
Jonas Moritz Zender
Matriculation number: 6410897

1st Supervisor: **Professor Dr. Nils Strodthoff**
2nd Supervisor: **M.Sc. Tiezhi Wang**

Submission date:
7th of May, 2025

Abstract

Sensor networks across various domains generate increasingly large volumes of time series data, often with an inherent graph structure - e.g., based on physical location of the sensors. Deep Learning models, particularly Spatiotemporal Graph Neural Network (STGNNs), offer promising data-driven approaches to extract meaningful insights from such data. This work explores Earthquake Early Warning as a time-sensitive application to systematically investigate design choices and training strategies for STGNNs. Previous works in this field focused on region-specific highly specialized models and treated their components as black boxes.

Therefore, the primary focus here is on developing generalizable models and understanding how individual components affect performance and robustness. Key findings include the importance of incorporating static node features - such as sensor location - paired with appropriate preprocessing. Masking parts of the input time series during training improved the robustness to missing data and short signals. A novel method for training deep probabilistic ensembles is introduced, encouraging individual ensemble members to specialize on structured subsets of the data (e.g. specific types of nodes).

The study reveals that even small design changes can substantially affect performance, and highlights a complex trade-off between generalization and region-specific optimization.

Overall, the results contribute toward building more robust, adaptable models for graph structured time series data across different domains. Future works could further explore architectural improvements, region-specific adaptation to general models, and apply a more robust evaluation and hyperparameter tuning.

Keywords: Deep Ensembles, Deep Learning, Earthquake Early Warning, Graph Neural Networks, Graphs, Probabilistic Models, Self-Supervised Learning, Spatio-Temporal Models, Time Series, Time-then-Space Models, Uncertainty Estimation

Contents

List of Abbreviations

Adam	ADAptive Moment estimation
BN	Batch Normalization
CNN	Convolutional Neural Network
CPC	Contrastive Predictive Coding
CV	Computer Vision
DL	Deep Learning
EEG	Electroencephalography
EEW	Earthquake Early Warning
FDSN	International Federation of Digital Seismograph Networks
GAN	Generative Adversarial Network
GAT	Graph Attention Network
GI	Graph INSTANCE
GIN	Graph Isomorphism Network
GNN	Graph Neural Network
GCN	Graph Convolutional Network
GraphSAGE	Graph SAmple and aggreGatE
GRU	Gated Recurrent Unit
HSIC	Hilbert-Schmidt Independence Criterion
IM	Intensity Measurement
INGV	Istituto Nazionale di Geofisica e Vulcanologia
INSN	Italian National Seismic Network
LoRA	Low Rank Adaption
lr	learning rate
LSTM	Long Short-Term Memory
ML	Machine Learning
MLP	Multilayer Perceptron
MP	Message-Passing
MSE	Mean Square Error
NLL	Negative Log-Likelihood
NLP	Natural Language Processing
NN	Neural Network
P	Primary

PGA	Peak Ground Acceleration
PGV	Peak Ground Velocity
RBF	Radial Basis Function
ReLU	Rectified Linear Unit
RKHS	Reproducing Kernel Hilbert-Space
RNG	Random Number Generator
RNN	Recurrent Neural Network
S	Secondary
SA	Peak Spectral Acceleration
SimSiam	Simple Siamese Representation Learning
STGNN	Spatiotemporal Graph Neural Network
STT	space-then-time
S4	Structured State Space Model
T&S	time-and-space
TCN	Temporal Convolutional Network
TTS	time-then-space

List of Figures

List of Tables

List of Code Listings

1. Introduction

There is a steadily increasing amount of time series data available across many different domains. Data may be collected by official monitoring networks, citizen science projects, Internet of Things networks, or more. Often, the time series data can be organized in a graph structure, e.g. through the physical positions of the sensors. Examples for this could be monitoring stations for air quality, river gauges, or even sensors in a medical context such as electrodes measuring electrical activity in the brain in Electroencephalography (EEG). When time series data can be structured as a graph, it can be processed by a class of Deep Learning (DL) models referred to as Spatiotemporal Graph Neural Networks (STGNNs). These models can learn complex non-trivial relationships, enabling data-driven pattern recognition even in noisy or heterogeneous settings. Success of DL for such data can be found even in notoriously complex domains such as weather forecasting, where DL models have achieved performances comparable to traditional physical models, while requiring only a fraction of their computing power and time [Lam et al., 2023].

Such methods thus pose a significant advantage in applications, where explicit modelling is difficult and requires strong expertise in domain knowledge, or in time-sensitive settings. One of these applications is Earthquake Early Warning (EEW), where the goal is to predict the intensity of an earthquake at different locations through so-called Intensity Measurements (IMs), based on the first seconds of the earthquake. Such data is collected by networks of seismic monitoring stations, measuring the ground velocity or acceleration in three components of motion through seismometers or accelerometers.

Accurate, low latency predictions in EEW could enable real-time warning systems that can help mitigate risks to human life. The data in EEW is challenging, as associated graphs can be highly heterogeneous, varying significantly in terms of structure, number of nodes or physical scale. Furthermore, the seismic time series need to be limited in length for effective EEW, and often contain missing data or just background noise.

Previous works have used plain Convolutional Neural Networks (CNNs) or Graph Neural Networks (GNNs) in combination with CNNs for time series encoding to predict IMs [Jozinović et al., 2020; Bloemheuvel et al., 2023]. However, the authors used static graphs and data strongly confined to a specific region.

As a result, the models are not able to generalize and would be of limited use for real-world deployment. Furthermore, design components are often treated as black boxes.

There is a lack of understanding how specific components, design choices, and training methods actually affect the predictive performance and robustness of such models. This raises the need for systematic exploration instead of chasing benchmarks. Furthermore, there should be a paradigm shift from highly specialized, region-specific models to foundational models that can be applied on a broader scale and adapted to local conditions.

This work intends to make a step in this direction by investigating different approaches to data preprocessing, architecture, and training methods, while keeping the model as general as possible. Ideally, derived results should thus allow insights into the general challenges of handling graph-structured time series with Neural Networks (NNs), independent of the domain of application.

Specifically, this work:

- Constructs an EEW dataset for STGNNs built from the INSTANCE dataset [Michelini et al., 2021],
- Evaluates the influence of the learning rate as a critical hyperparameter;
- Investigates graph construction, especially filtering of low-weighted edges;
- Examines the importance of static node features (station position and altitude) and influence of preprocessing;
- Tests self-supervised learning as a pretraining method;
- Explores station dropout - i.e. partial masking of input time series - as a way to improve robustness;
- Provides a proof-of-concept for creating earthquake intensity maps using DL;
- Studies probabilistic models beyond simple point predictions;
- Proposes a method for jointly training deep probabilistic ensembles, while encouraging specialization of their components.

The discussion concludes by addressing practical challenges encountered in building generalizable models, and outlines possible solutions and directions for future work.

2. Background and Literature Review

2.1. Time Series

2.1.1. Definitions

We can consider a time series as a consecutive sequence of synchronously sampled observations. Each time series might be multivariate, e.g. through different sensor channels. We can further consider a collection of N correlated time series, e.g. collected by different stations in a monitoring network. In accordance to Cini et al. [2023], we can define the k-th time series from this collection as a sequence of vectors $\boldsymbol{x}_t^{(k)} \in \mathbb{R}^{d_x}$, representing the d_x-dimensional observation at time step t. Further, it is assumed that the time series are homogenous, that means they are composed of the same observed variables, i.e. the same d_x channels. $\boldsymbol{X}_t \in \mathbb{R}^{N \times d_x}$ then represents all N multivariate time series at time step t. Finally, $\boldsymbol{X}_{t:t+T}$ represents all observations within the time interval $[t, t+T)$.

In the following chapters, *temporal* components of the models refer to processing individual time series $\boldsymbol{x}_{t:t+T}^{(k)}$, while *spatial* components refer to processing the relations among the N time series within a collection (sample), i.e. through the use of graphs and GNNs.

Furthermore, each time series may have a set of d_v-dimensional static features associated with it, denoted by $\boldsymbol{V} \in \mathbb{R}^{N \times d_v}$ for all N time series. Static features could e.g. be the physical location of the N sensors, which does not change over time.

2.1.2. Neural Networks for Time Series

There is a number of different DL architectures for time series processing. One prominent approach are CNNs, which were first succesfully used in image data for document recognition [Lecun et al., 1998], but have later also been applied to time series, e.g. for feature extraction for classification [Zheng et al., 2014].

Formally, the operation used in a 1d CNN for multivariate time series can be expressed as

$$x'(t, c_{out}) = \sum_{c_{in}=0}^{C_{in}-1} \sum_{k=-\lfloor \frac{K}{2} \rfloor}^{\lfloor \frac{K}{2} \rfloor} x(t+k, c_{in}) \cdot w(k + \lfloor \frac{K}{2} \rfloor, c_{in}, c_{out}), \tag{2.1}$$

where $\boldsymbol{x} \in \mathbb{R}^{T \times C_{in}}$ is the input time series of length T with C_{in} channels (i.e. d_x). $\boldsymbol{w} \in \mathbb{R}^{K \times C_{in} \times C_{out}}$ is the kernel or filter with the parameters to optimize and kernel size K and C_{out} output channels. $\boldsymbol{x'} \in \mathbb{R}^{T_{out} \times C_{out}}$ is the output, where T_{out} is the length of the output time series [Dumoulin and Visin, 2018], determined by

$$T_{out} = \left\lfloor \frac{T - K}{\text{stride}} \right\rfloor + 1. \tag{2.2}$$

In a regular discrete convolution, the kernel $\boldsymbol{w}$ slides across the input $\boldsymbol{x}$ and at each point in time t, the sum of the elementwise products between $\boldsymbol{x}$ and $\boldsymbol{w}$ is calculated to yield the output. During implementation, it is usually easier to use the commutative property of convolutions and slide the input $\boldsymbol{x}$ across the kernel $\boldsymbol{w}$, which effectively flips the kernel. However, since the weights in the kernel $\boldsymbol{w}$ are learnable parameters anyway, the flipping does not affect the optimization process. Therefore, usually the unflipped kernel is used, which strictly speaking turns the convolution into a cross-correlation, but in the context of CNNs the terms are used interchangeably [Goodfellow et al., 2016]. An additive bias may be applied to the output of the convolution, as well as a non-linear activation function σ.

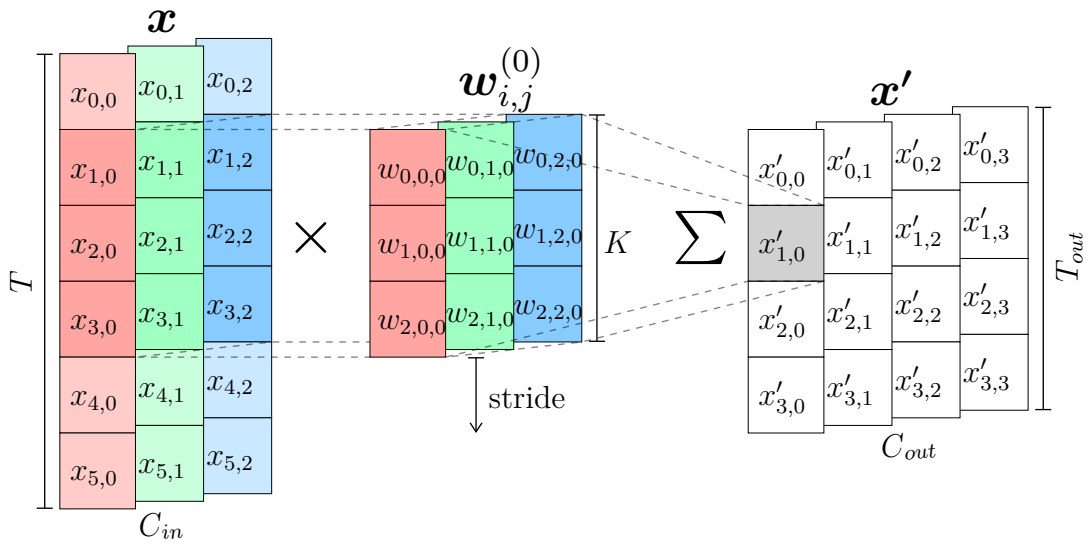


Figure 2.1.: Illustration of the convolution step in a 1d CNN layer. The input time series $\boldsymbol{x}$ is convolved with the filter $\boldsymbol{w}$ to yield the output time series $\boldsymbol{x'}$.

Figure 2.1 illustrates the calculation of the convolution in a 1d CNN layer. Figuratively, the kernel $\boldsymbol{w}$ slides across the input time series $\boldsymbol{x}$; the step size is the stride. At each position t, the elementwise product between $\boldsymbol{x}$ and $\boldsymbol{w}$ is calculated and the results are summed to yield a single entry in the output features $\boldsymbol{x'}$. This is done C_{out} times to yield an output with $C_{out} = 4$ channels in this illustration. The length of the output time series can be calculated with Eq. 2.2, i.e. $T_{out} = \left\lfloor \frac{T-K}{\text{stride}} \right\rfloor + 1 = \left\lfloor \frac{6-3}{1} \right\rfloor + 1 = 4$.

2.2. Graphs

2.2.1. Definitions

A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consists of a set of N nodes or vertices $v_i \in \mathcal{V}$ and a set of edges $(v_i, v_j) \in \mathcal{E}$, where an edge $e_{i,j}$ connects node v_i to node v_j. The structure of a graph can be represented through an adjacency matrix $\boldsymbol{A} \in \mathbb{R}^{N \times N}$, where $A_{i,j} = 1$ indicates a directed edge from node v_j to node v_i, while $A_{i,j} = 0$ refers to the abscence of such an edge. Each row i in the adjacency matrix thus describes which nodes v_j have a directed edge towards v_i. The adjacency matrix may be binary, just indicating presence or abscence of an edge, but can also be weighted, for example encoding the physical distance between two nodes. In the case of an undirected graph, where a connected pair of vertices has an edge in both directions, the adjacency matrix is symmetric $A_{i,j} = A_{j,i} \forall i, j$.

The degree of a node refers to the number of its neighboring nodes. It can be represented by a diagonal degree matrix $\boldsymbol{D} \in \mathbb{R}^{N \times N}$ where $D_{i,i} = \sum_j A_{i,j}$. In directed graphs, the out-degree (number of outgoing edges) and in-degree (number of incoming edges) have to be distinguished. Since GNNs aggregate information from neighboring nodes, the in-degree is used to define the degree matrix in this context.

2.2.2. Neural Networks for Graphs

GNNs are a class of DL architectures used to process data that can be arranged in a graph structure. An older class of models, the previously discussed CNNs, have seen great success for data in a grid structure, such as time series or image data [Krizhevsky et al., 2012]. They enable weight sharing and the aggregation of information within a local neighborhood through the use of convolutions with limited receptive fields. Since then there have been different approaches to define convolutions on graphs and use them in DL [Bloemheuvel et al., 2023].

One popular method are so-called Graph Convolutional Networks (GCNs), which were proposed by Kipf and Welling [2017] with the following propagation rule:

$$\boldsymbol{H}^{(l+1)} = \sigma\left(\tilde{\boldsymbol{D}}^{-\frac{1}{2}}\tilde{\boldsymbol{A}}\tilde{\boldsymbol{D}}^{-\frac{1}{2}}\boldsymbol{H}^{(l)}\boldsymbol{W}^{(l)}\right). \tag{2.3}$$

$\boldsymbol{H}^{(l)} \in \mathbb{R}^{N\times C}$ is the matrix of node features (with C input channels) from the l-th layer and σ is a nonlinear activation function. $\tilde{\boldsymbol{D}} = \sum_j \tilde{A}_{i,j}$ is the degree matrix for the adjacency matrix $\tilde{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}_N$ of the graph $\mathcal{G}$, including self-connections through the addition of the identity matrix $\boldsymbol{I}_N$.

The underlying principle behind this is referred to Message-Passing (MP) [Gilmer et al., 2017]. Multiplying the adjacency matrix $\boldsymbol{A}$ with the feature matrix $\boldsymbol{H}^{(l)}$ corresponds to aggregating the features of the neighbors of a node with a sum across its 1-hop neighborhood. Using $\tilde{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}_N$ instead of $\boldsymbol{A}$ means that a node can send a message to itself, i.e. its own features are also used during the aggregation.

The multiplication with the degree matrix in $\tilde{\boldsymbol{D}}^{-\frac{1}{2}}\tilde{\boldsymbol{A}}\tilde{\boldsymbol{D}}^{-\frac{1}{2}}$ acts as a normalization, which is necessary for numerical stability during training. Each edge weight in the adjacency matrix is thus normalized by the square root of the degree of the sending and the receiving node. The resulting matrix is referred to as the normalized graph Laplacian. Intuitively, it means that incoming messages from highly connected nodes are less relevant to the individual receving node.

The output of the MP is then transformed through the multiplication with a learnable weight matrix $\boldsymbol{W}^{(l)} \in \mathbb{R}^{C\times F}$ and a nonlinear activation function σ to produce an output $\boldsymbol{H}^{(l+1)} \in \mathbb{R}^{N\times F}$ with F output channels. Weight-sharing is thus achieved across all nodes with a single weight matrix.

It is important to note that the graph convolution defined defined in GCNs imposes a limit on how far information can spread through the graph. Each MP step only aggregates information from the 1-hop neighborhood of a node. Therefore, to spread information from one node to another node, that is n edges away, n steps of MP - and thus n GCN layers - are necessary. Simultaneously, deep GCNs can suffer from over-smoothing [Li et al., 2018], i.e. node features becoming less distinguishable, and vanishing gradient problems [Li et al., 2019]. Chosing an appropriate number of layers in a GCN is thus crucial for the performance of the model.

Figure 2.2 illustrates the MP in a GCN without applying the transformation on the aggregated features. It shows limitations of GCNs for long range connections, similar to those of CNNs. As the number of MP steps increases, the features are smoothed across the graph as they approach some steady state.

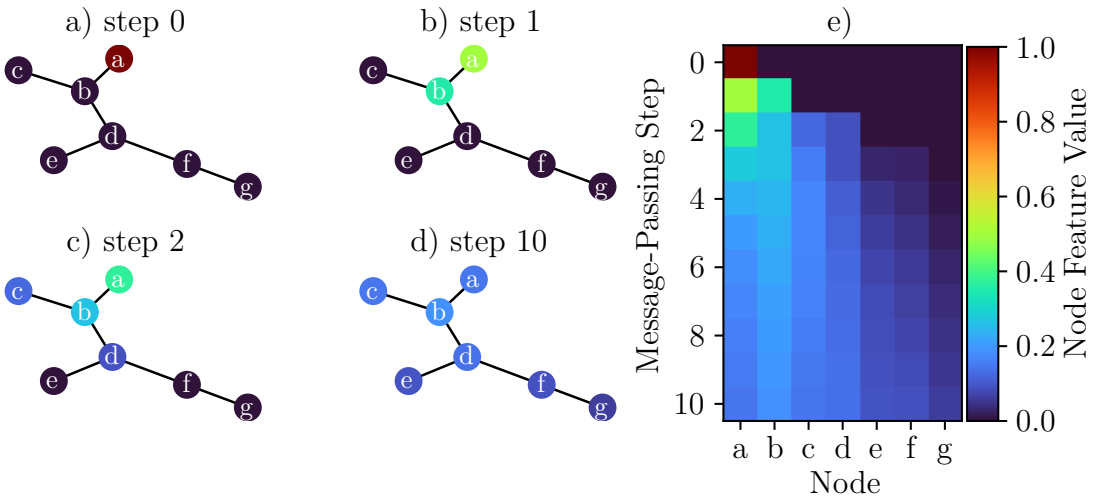


Figure 2.2.: Illustration of how information spreads through a graph during Message-Passing (MP) of a GCN. a-d) show different MP steps for a graph with a scalar node feature, where node a has a value of 1 and the remaining nodes have a value of 0. e) shows the development of the node feature during the MP. The calculation was done using Eq. 2.3 without transforming the aggregated features; i.e. $\boldsymbol{H}^{(l+1)} = \tilde{\boldsymbol{D}}^{-\frac{1}{2}}\tilde{\boldsymbol{A}}\tilde{\boldsymbol{D}}^{-\frac{1}{2}}\boldsymbol{H}^{(l)}$. Adapted from [Jost, 2020].

2.3. Related Works

DL models that operate on time series and graphs are referred to as STGNNs. Cini et al. [2023] provide a foundational framework for such models, as well as guidelines for their variants and design principles, performance evaluation, open challenges and research directions. Most importantly, they discuss three different categories of STGNNs:

- time-then-space (TTS) models
- space-then-time (STT) models
- time-and-space (T&S) models

These models differ mainly in the order in which they apply *temporal* and *spatial* components. TTS models first apply temporal components, e.g. a CNN or a Long Short-Term Memory (LSTM), to process the time series, and then use their output features as input for the spatial model components, e.g. a GCN, to attend to the graph structure of the data. STT models simply apply the inverse order of model components, first doing the spatial and then the temporal processing. In contrast, T&S models do not strictly separate spatial and temporal processing and instead combine them, e.g. by alternating spatial and temporal components, or by modifying the network layers to enable them to process both at the same time, e.g. through recurrent GNNs [Seo et al., 2018]. Generally, TTS and T&S models tend to perform better than STT models. The computational complexity is lower

for TTS models than for STT and T&S models [Cini et al., 2023]. Therefore, this project focuses on TTS architectures, which are also not as much explored in research as T&S models.

The main application in this project is seismic data, most importantly the prediction of ground motion intensity at various positions during an earthquake. Jozinović et al. [2020] built an appropriate dataset of ground motion time series and applied a CNN to it. [Bloemheuvel et al., 2023] used this as a basis and added GCN layers to also account for the spatial component of the data through a TTS model. However, their approach does not generalize well as they use a dataset limited to a small area and use a static graph, which consists of 39 stations. In case data was missing for a station, the real ground motion intensities, the targets during optimization, were unknown. Instead, for monitoring stations without data, they replaced the input time series with 0s and the target with simulated values. Furthermore, the creation of their graph was optimized to this specific graph and dataset, through evaluating the Mean Square Error (MSE) loss for different configurations. Finally, they fix the size of graph also in the output layers of their architecture. This makes their model unable to generalize, neither in terms of their application of ground motion intensity prediction, nor in terms of other applications. The models developed in this project shall take this a step further and make them as generalizable as possible using dynamic graphs from a more diverse dataset.

3. Data and Methodology

3.1. Seismic Dataset (Graph INSTANCE)

The seismic dataset used here is based on the INSTANCE dataset [Michelini et al., 2021], which comprises almost 1.2 million trace samples from 50000 earthquakes observed by different sensor networks. In this work, only samples from network IV of the International Federation of Digital Seismograph Networks (FDSN) are used. This network, the Italian National Seismic Network (INSN), operated by the Istituto Nazionale di Geofisica e Vulcanologia (INGV) [INGV, 2005], comprises 92.6% of the INSTANCE samples and uses the same devices across its monitoring stations. The dataset was further filtered for earthquakes of magnitude $M \geq 3.0$, which were recorded by at least 10 stations. This yields $S = 2853$ earthquakes, observed by 457 monitoring stations.

3.1.1. Input Data

Each sample s is observed by a different number of stations N_s. Each station may have results from up to three different seismic measurement devices, all of which consist of three component (E: East-West; N: North-South; Z: Up-Down) waveform data sampled at 100Hz for 120s. These devices provide either weak-motion recordings (EH and HH channels), which measure the ground velocity in m/s, or strong-motion recordings (HN channel), which measure the ground acceleration in m/s^2. EH/HH channels are more sensitive and are thus better suited for detecting weaker earthquakes, while on the contrary they might saturate for stronger earthquakes [Jozinović et al., 2020]. Therefore, since earthquakes with magnitude $M < 3.0$ were filtered out, the HN channels were preferred and EH/HH channels were only used if there was no HN data available. The input data for the models need to be consistent in their units, therefore, the time series for EH/HH channels were differentiated to obtain accelerations in m/s^2.

The objective of the models is to perform EEW, which aims to predict the intensity of an earthquake at different locations. EEW systems utilize the fact that data can be transmitted faster than seismic waves are propagated, to provide early warnings on imminent ground motion in more distant locations [Jozinović et al., 2020]. For such systems to be effective, they need to make predictions

rapidly and already at a time, where the seismic wave is only detected by one or a limited number of stations. Consequently, the time series were trimmed to a length of only 10s. Previous works on more confined datasets used the earthquake origin time [Bloemheuvel et al., 2023] as starting time. However, if the earthquake needs more than 10s from its epicenter to the nearest seismic monitoring station, then the input signal in the time window would just be noise. An earthquake is composed of a fast Primary (P) wave and a slower Secondary (S) wave; their arrival times at each station are included in the INSTANCE metadata. To ensure a proper signal and introduce variability, the time series were set to randomly start 0 to 5s before the first P arrival time.

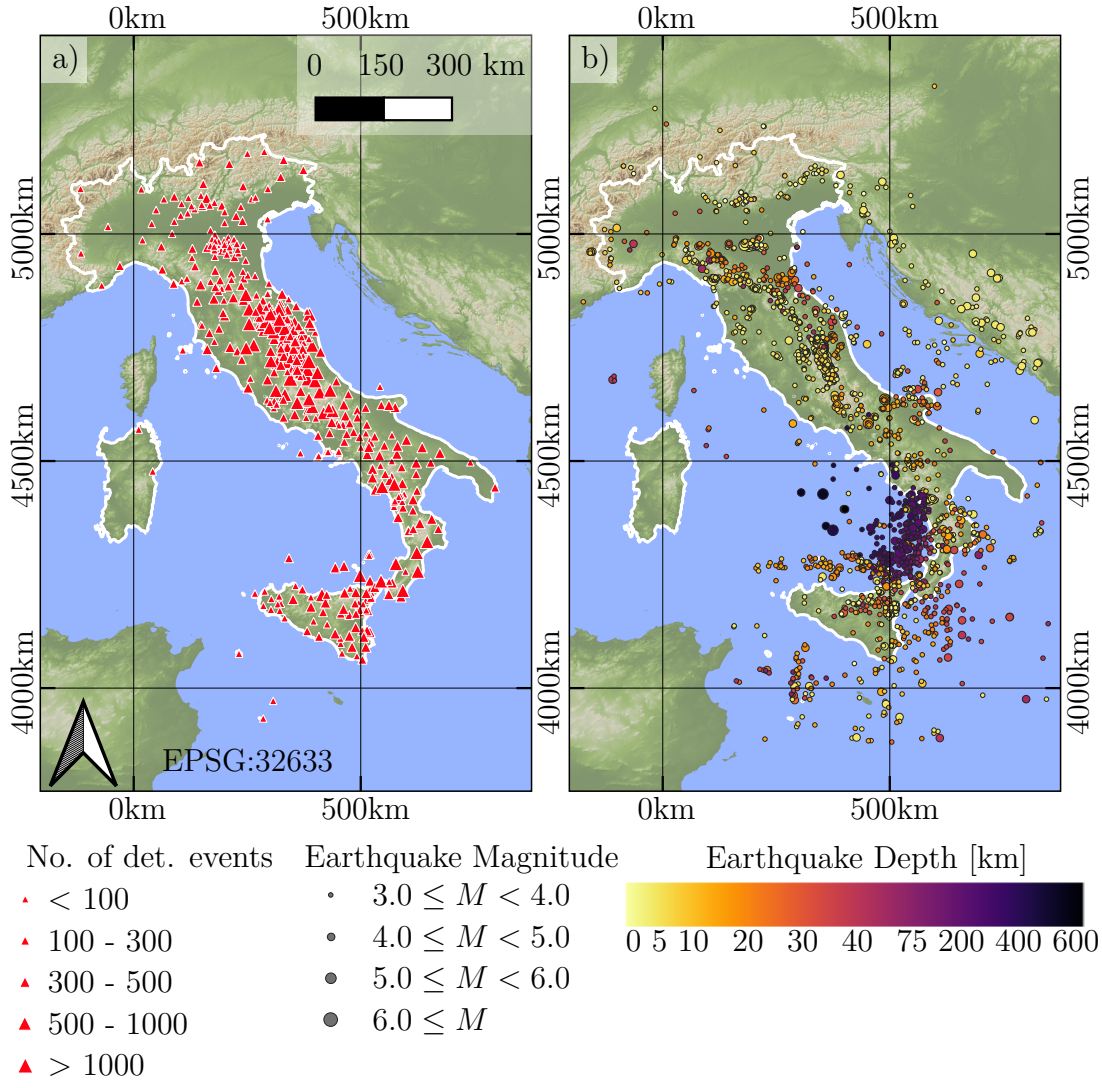


Figure 3.1.: Map of a) the 457 monitoring stations from network IV as triangles, where the marker size represents the number of detected earthquake events, and b) the 2853 earthquake events as solid circles, where the marker size corresponds to the earthquake magnitude and color encodes the depth. Adapted from [Michelini et al., 2021].

For some stations and earthquakes, the time series may not cover the entire time frame of interest. In these cases, they were padded to the appropriate length using zeros. After differentiating the EH/HH channels, the maximum amplitude measured across all channels and stations of a sample was used to normalize the time series as in [Jozinović et al., 2020; Bloemheuvel et al., 2023].

In total, there are $S = 2853$ samples in the dataset, where each sample s contains a collection of N_s (number of stations that recorded the earthquake) correlated multivariate ground motion time series $\boldsymbol{X}^{(s)}_{t:t+T} \in \mathbb{R}^{N_s \times T \times d_x}$, where $T = 100\text{Hz} \cdot 10\text{s} = 1000$ is the number of time steps. Each time series $\boldsymbol{x}^{(s,k)}_{t:t+T} \in \mathbb{R}^{T \times d_x}$ in the collection/sample has $d_x = 3$ channels, one for each component of motion (ENZ).

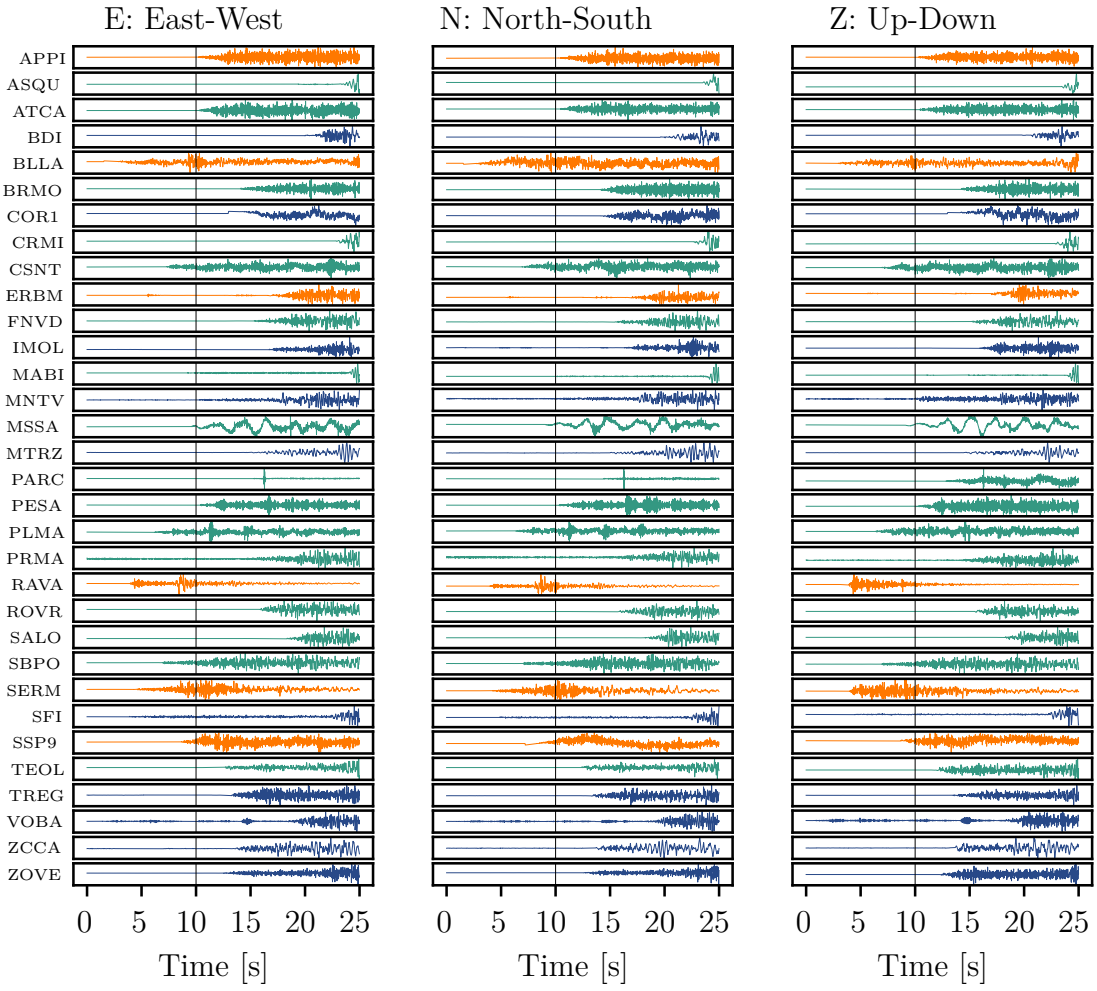


Figure 3.2.: Unnormalized time series for earthquake 772141, which occurred at 2012-05-19 23:13:25, with a magnitude of $M = 4.0$ at a depth of 9.3km. Each row represents one of the 32 stations that recorded the event. Each column represents one component of motion (ENZ). $t = 0$s corresponds to 3.90s (randomly chosen per sample) before the earliest P arrival time. The marker at $t = 10$s represents the end of the window which the models use as input; the time series afterwards is hidden from the models. The colors represent the measurement device channels, which the data stems from (blue: HN, orange: EH, green: HH).

Fig. 3.1 shows an overview over the filtered dataset. a) shows the positions of the monitoring stations, where the size of the triangles refers to the number of events the stations have detected. b) shows the locations of the earthquake epicenters, where the size of the points refers to earthquake magnitude and color refers to the depth.

Fig. 3.2 shows the unnormalized time series of an individual earthquake (ID: 772141) for all three components of motion (ENZ). The colors correspond to the channel the data originates from (HN, EH, HH). The models only use the first 10s of the time series; the rest remains hidden. At this point, the earthquake has only reached two stations (RAVA and SERM).

In addition to the multivariate time series, station position and elevation were used as static input data for each node. Formally, they make up the static input $\boldsymbol{V}^{(s)} \in \mathbb{R}^{N_s \times d_v}$, where N_s is the number of stations that recorded the earthquake in sample s, and $d_v = 3$ is dimensionality of the static features (x-coordinate, y-coordinate, elevation). The elevation was normalized by the maximum elevation of all stations in a sample. The positions of the stations are given in latitude and longitude in the INSTANCE dataset. In order to not overfit on the absolute coordinates, they were translated into relative coordinates. A reference point was randomly chosen within the latitude and longitude ranges of the stations of each sample s. Then the distances in North-South and East-West direction between each node and the reference point were calculated, where negative values indicate nodes located to the South or West of the reference point. Finally, these relative coordinates were normalized using the maximum absolute distance component across all nodes to yield values in $[-1, 1]$.

3.1.2. Target Data

As mentioned before, the objective in EEW is the prediction of ground motion intensities, which are referred to as IMs. The INSTANCE dataset provides extensive meta data for each trace and earthquake event, including commonly used IMs. These are the Peak Ground Acceleration (PGA) in cm/s^2, the Peak Ground Velocity (PGV) in cm/s, and the Peak Spectral Acceleration (SA), also in cm/s^2, at periods of 0.3s, 1.0s, and 3.0s. The SA refers to the maximum acceleration on an object during an earthquake, specifically on a damped harmonic oscillator. The periods are inversely related to the natural frequency of oscillation of the object, which could be a building. An $SA_{0.3}$ thus refers to the SA of an oscillator, e.g. a building, with a natural frequency of $\frac{1}{0.3s} = 3.\bar{3}$Hz, with a given damping factor and stiffness, during a specific earthquake. All IMs are first obtained/calculated for each component of motion (ENZ) individually. The maximum value across all components is then used as a target. The node-level targets for the

models can thus be formulated as $\boldsymbol{Y}^{(s)} \in \mathbb{R}^{N_s \times 5}$, where 5 is the number of IMs (PGA, PGV, $SA_{0.3}$, $SA_{1.0}$, $SA_{3.0}$) for each station. According to [Jozinović et al., 2020] the base-10 logarithm has been applied to all IMs (i.e. $\log_{10}$ IM).

In addition, the position of the earthquake epicenters, normalized and relative to the random reference point chosen for the node positions, was included as a graph-level regression target $\boldsymbol{y}^{(s)} \in \mathbb{R}^2$.

3.1.3. Graphs

To apply GNNs to the dataset, graphs were created for each sample in a dynamic way. Each of the N_s seismic monitoring stations, which detected a specific earthquake s, represents a node in the graph $\mathcal{G}_s$ for that sample s. The adjacency/weight matrix $A_{i,j}^{(s)}$ was populated using all the pairwise geodesic distances in km between the stations belonging to a sample. Similar to [Bloemheuvel et al., 2023], the matrix was min-max-normalized and transformed with

$$A_{i,j}^{(s)} \leftarrow 1 - \frac{A_{i,j}^{(s)} - \min(A^{(s)})}{\max(A^{(s)}) - \min(A^{(s)})}. \tag{3.1}$$

This yields a matrix with values in $[0, 1]$, where far apart stations have a low weight and close stations have a high weight. This graph is (almost) complete, since each node i is connected with every other node j via an edge $e_{i,j}$, except for the furthest apart node pair, whose edge-weight is 0. Self-loops, i.e. edges where $i = j$, are omitted too as they are added in by the GCN layers of the models.

[Bloemheuvel et al., 2023] used a fixed set of 39 stations and thus created a static graph, where they adjusted the sparseness by filtering out all edges with a weight below an experimentally found threshold value. In contrast, the graphs in this dataset are dynamic, as they change with each sample, depending on the stations which detected the specific earthquake. A 'good' threshold value may be different for each sample and thus cannot easily be found experimentally. Using an arbitrary value may also be a risk as it could split a graph into two or more disconnected components. Two methods were tested to adjust the sparseness of dynamic graphs by filtering out low-weighted edges.

The first approach introduced a hyperparameter d, which represents the minimum node degree (i.e. the number of edges connected to that node), which is allowed in the graph. The edges in the graph were ordered by their weight in ascending order. The currently lowest-weighted edge was then removed (by setting its weight to 0) if it did not split the graph into multiple components, and if it did not decrease the node degree of one of the nodes below d. As soon as one of these conditions would be violated, the filtering was stopped. This made the graph sparseness adjustable, while guaranteeing that it remains connected.

The second approach was based on the diameter $\mathcal{D}$ of the graph instead of a minimum node degree. The baseline model used here only has 2 GCN layers and thus only 2 steps of MP. In the first approach, the graph diameter was sometimes much larger than that, depending on the value of d and the size of the graph. To ensure that information could still flow between all nodes, the stopping criterium was set to whether the removal of the lowest-weighted edge would increase the diameter of the graph above 2, and if so, leave the edge and stop the filtering. Both approaches were investigated, as well as applying both at the same time; for more details refer to Sec. 4.3.

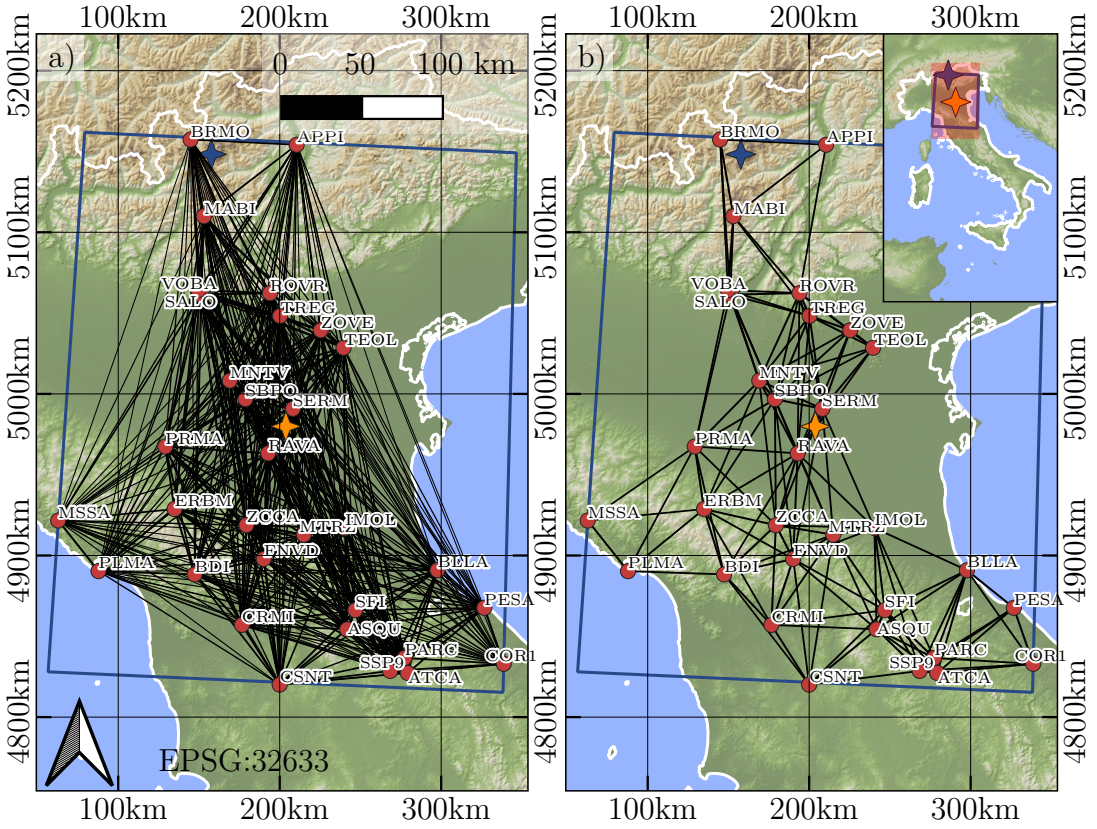


Figure 3.3.: Map displaying the graph for earthquake 772141, which occurred at 2012-05-19 23:13:25, with a magnitude of 4.0 at a depth of 9.3km. The orange star marks the epicenter of the earthquake. The red filled points mark monitoring stations, which detected it. Black lines represent the edges of the graphs; the line thickness corresponds to the edge weight. The blue bounding box shows the area from which the reference point (blue star) is randomly sampled. a) shows the complete graph with an edge connecting each node pair (495 edges). b) shows the reduced graph with a minimum node degree of $d = 3$ (136 edges).

Fig. 3.3 illustrates the graph of the previously shown sample (earthquake 772141, see Fig. 3.2). Seismic monitoring stations, i.e. the graph nodes, are shown as red points. Edges are shown as solid black lines, where the line thickness corresponds to the edge weight. a) shows the complete graph without filtering out nodes; b) shows the reduced graph with a minimum node degree of

$d = 3$. The blue bounding box shows the area from which the reference point (blue star) is randomly sampled. The relative normalized station positions, used in the static input features (along with station elevation) $\boldsymbol{V}^{(s)}$, as well as the earthquake epicenter position $\boldsymbol{y}^{(s)}$, are calculated with respect to that reference point.

The dataset was then split into a training (80%), validation (10%) and test (10%) split. Since earthquake magnitude is a key feature, particular care was taken to ensure that the distribution of samples w.r.t. their magnitudes was preserved in each split. Earthquake magnitude is a discrete feature (in steps of 0.1) in the INSTANCE dataset, where larger magnitudes are significantly rarer than smaller ones. The splitting was done using a stratified sampling method to ensure that the magnitude distribution was conserved across the splits. Sparse magnitude classes (i.e. earthquake magnitudes with 7 or less samples) could not be split using this method and were handled separately. The 29 samples with sparse magnitudes (4.9, 5.1 - 6.5) were randomly added to the training, validation and test split in the same 80:10:10 ratio.

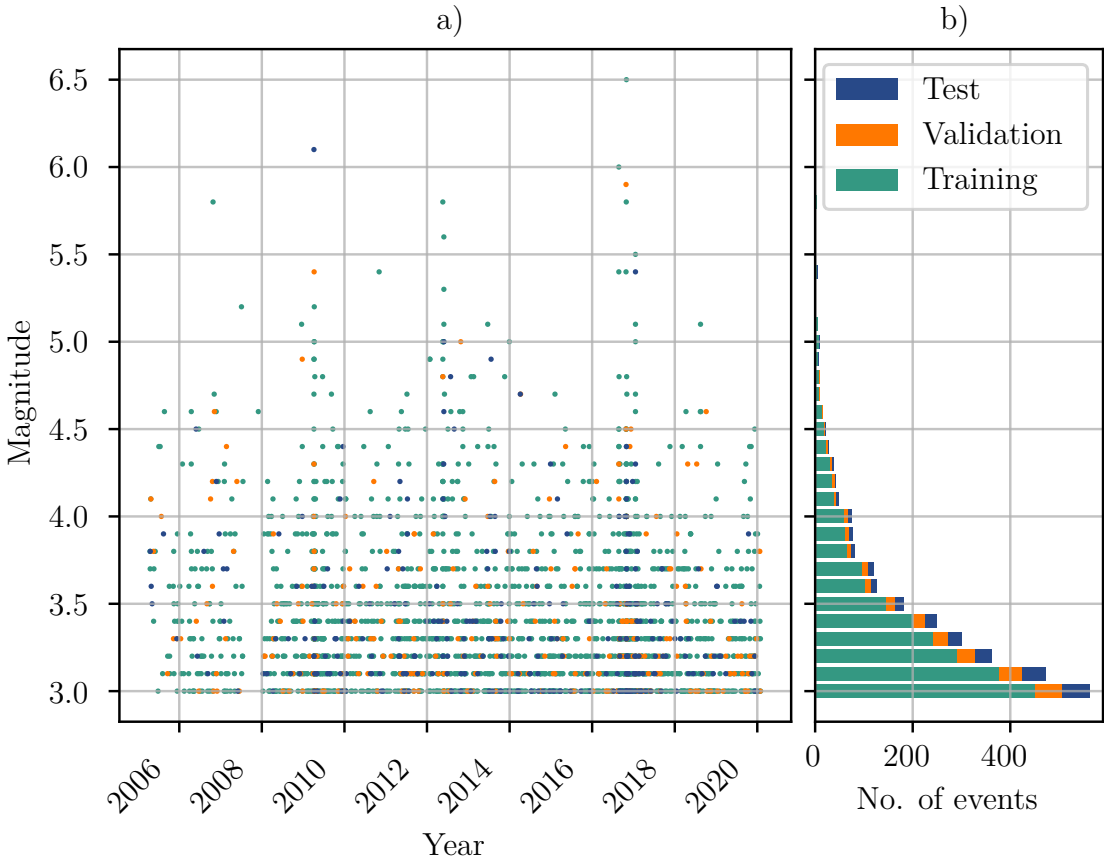


Figure 3.4.: a) occurrence of earthquakes and their magnitude over time in the GI dataset. b) Number of earthquakes events by magnitude and split.

From hereon, this seismic dataset will be referred to as the Graph INSTANCE (GI) dataset. Fig. 3.4a) shows the occurrence of earthquakes and their magni-

tudes over time in the GI dataset. Fig. 3.4b) shows the number of earthquake events for all individual magnitudes ranging within $[3.0, 6.5]$ across training, validation and test split.

Since the start of the time series is shifted randomly 0 to 5s before the first arrival time of the P wave, the same sample returns slightly differently trimmed time series each time it is accessed in the PyTorch dataset class. To make the evaluation comparable, there is an option to turn the sampling deterministic. This is done by using the ID of the earthquake as a seed for the Random Number Generator (RNG) before trimming the sample, such that each sample is always trimmed to the same time window. During training the shifting can be left random and then turned deterministic for the validation and test dataset.

The data processing was done using ObsPy [Beyreuther et al., 2010], Pandas [McKinney, 2010], Geopy [Geopy Contributors, 2008], NumPy [Harris et al., 2020], Networkx [Hagberg et al., 2008], scikit-learn [Pedregosa et al., 2011], PyTorch [Paszke et al., 2019] and PyTorch Geometric [Fey and Lenssen, 2019].

3.2. Baseline Model

The basis for the models developed in this project is the model by Bloemheuvel et al. [2023], which is illustrated in Fig. 3.5b. Blue boxes refer to input data, yellow boxes to NN layers, green boxes to reshaping of features, and purple boxes to output features. The arrows indicate the flow of the features through the model; the dimensions next to them are the shape of the features in between each processing step, for a single sample.

In [Bloemheuvel et al., 2023]'s model, the input is restricted to contain 39 nodes. The input time series are first processed through a 2-layer CNN, which processes the features of each node individually, without taking other nodes into account. The output is flattened to yield a single large feature vector for each node. This is concatenated with the static features, which, for [Bloemheuvel et al., 2023], are the latitude and longitude coordinates. Afterwards, the features are passed through a 2-layer GCN and then flattened to receive a single feature vector for the whole graph. After applying a dropout, the vector is passed through a linear layer. Finally, the result is passed to 5 separate linear layers, which map to a 39-dimensional vector (corresponding to the 39 nodes), each representing one of the 5 IMs used as targets. While the model base is general, the model head (after the GCN layers) is restricted to a static graph with 39 nodes.

Fig. 3.5a illustrates the more general model used as a baseline in this project. The model base is identical to [Bloemheuvel et al., 2023], except for the node features $\boldsymbol{V}^{(s)}$, which are the station coordinates w.r.t. a random reference point, and include an additional dimension for the station elevation. The output from the GCN layers is passed to two separate model heads, one for the prediction of IMs and one for the earthquake epicenter.

The prediction of IMs is done through a single linear layer, which maps the feature vector of each node to a 5-dimensional vector representing the 5 IMs. Since this is applied node-wise, it can handle any dynamic graph.

The prediction of the earthquake epicenter is a graph-level optimization target. Therefore, a global average pooling is applied, which effectively reduces each graph in the batch to a single node, representing the graph as a whole. The output of the pooling has thus always the same dimension independent of graph size. This feature vector is then passed through a linear layer which maps to the coordinates of the epicenter. An implementation of this baseline model in Pytorch can be found in Sec. A.1.

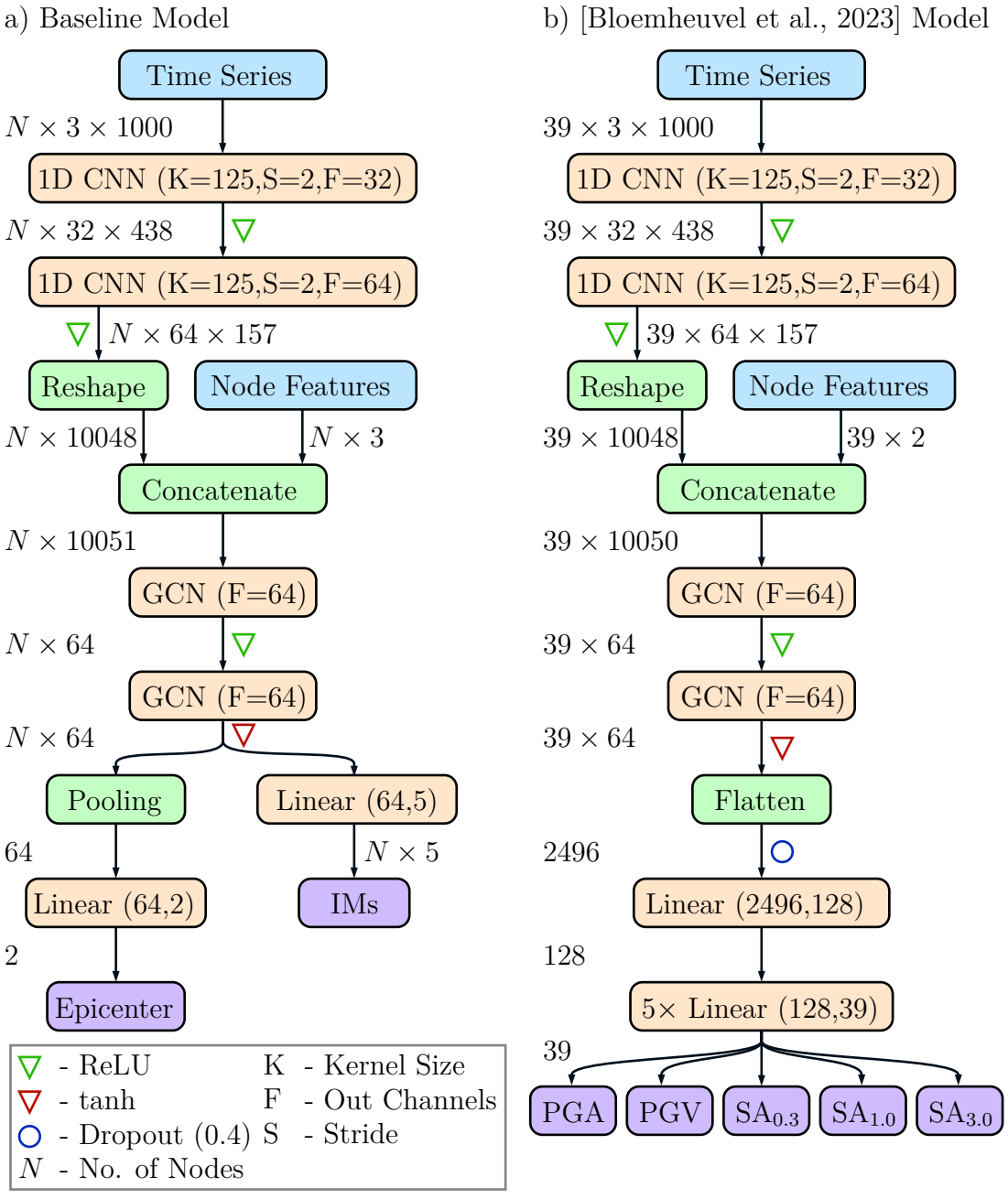


Figure 3.5.: Comparison between a) the baseline model used in this project, and b) the model by [Bloemheuvel et al., 2023]. The arrows show how the features are passed through the model; the dimensions next to the arrows are the feature dimensions of a single sample processed by the different building blocks of the model. Blue boxes refer to input data, yellow boxes to NN layers, green boxes to reshaping of features, and purple boxes to output features. An implementation of the baseline model using Pytorch can be found in Sec. A.1.

3.3. Model Optimization

3.3.1. Mean Square Error Loss

Similar to [Bloemheuvel et al., 2023], the model can be optimized using the MSE loss with

$$\mathcal{L}_{\text{node}} = \frac{1}{N} \sum_{i=1}^{N} \left(\boldsymbol{Y}_i - \hat{\boldsymbol{Y}}_i \right)^2, \tag{3.2}$$

$$\mathcal{L}_{\text{graph}} = \frac{1}{B} \sum_{j=1}^{B} \left(\boldsymbol{y}_j - \hat{\boldsymbol{y}}_j \right)^2, \tag{3.3}$$

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{node}} + \mathcal{L}_{\text{graph}}, \tag{3.4}$$

where N is the number of nodes in the batch, B is the batch size, $\boldsymbol{Y}_i$ are the actual values and $\hat{\boldsymbol{Y}}_i$ are the predicted values for the nodes (i.e. the IMs), and $\boldsymbol{y}_j$ are the actual values and $\hat{\boldsymbol{y}}_j$ are the predicted values for the graphs (i.e. earthquake epicenter position).

3.3.2. Probabilistic Loss

An alternative to the MSE loss for regression tasks has been proposed by Lakshminarayanan et al. [2017] with the goal of providing an uncertainty estimation for the model predictions. Such predictive uncertainties are often not straightforward to implement or require significant additional computational costs. The authors proposed a method where the NN itself outputs a predictive uncertainty, without requiring additional labels for the optimization.

Instead of making simple point predictions for the target variables, the NN is trained to provide parameters of a Gaussian distribution, i.e. mean μ and variance σ^2 for each target variable. μ can then be treated as a point prediction and σ^2 is the predictive uncertainty, where a high σ^2 reflects a higher uncertainty.

The authors have shown that this approach can lead to better point predictions and is thus beneficial even if predictive uncertainties are irrelevant to the downstream task. However, in many applications - including EEW - having an estimate of predictive uncertainty can be useful. On one hand, a high variance can indicate to the end user to interpret the point prediction μ more carefully; e.g. taking the incoming EEW more seriously. On the other hand, a high variance can show out-of-distribution samples; e.g. if a model trained on earthquakes in Italy and is applied to earthquakes in the USA, it should output a higher predictive uncertainty σ^2.

To implement this in the model in Fig. 3.5a for the prediction of the IMs, another linear output layer of identical dimensions is necessary, such that for each node and IM a mean μ_θ and variance σ_θ^2 is predicted, where θ are the model parameters. To train the model, the Negative Log-Likelihood (NLL) loss is minimized with:

$$\mathcal{L}_{\text{NLL}} = \sum_{j=1}^{B} \sum_{i=1}^{N_j} \left(\frac{\log \boldsymbol{\sigma}_{\theta,j,i}^2(\boldsymbol{X}_j)}{2} + \frac{(\boldsymbol{Y}_{j,i} - \boldsymbol{\mu}_{\theta,j,i}(\boldsymbol{X}_j))^2}{2\boldsymbol{\sigma}_{\theta,j,i}^2(\boldsymbol{X}_j)} \right), \tag{3.5}$$

where B is the batch size, N_j is the number of nodes in the j-th sample of the batch, $\boldsymbol{X}_j$ is the input data (time series, graph, static features), $\boldsymbol{Y}_{j,i}$ are the IMs for the i-th node in the j-th sample, and $\boldsymbol{\mu}_{\theta,j,i}$ and $\boldsymbol{\sigma}_{\theta,j,i}^2$ are mean and variance for the i-th node in the j-th sample as predicted by the model parametrized through θ [Lakshminarayanan et al., 2017].

The variance σ_θ^2 is constrained to non-negative values, but this constraint is not automatically fulfilled through the architecture. Therefore, the model output is passed through the softplus function:

$$\sigma_\theta^2 \coloneqq \text{softplus}(\sigma_\theta^2) = \log(1 + \exp(\sigma_\theta^2)) + \epsilon. \tag{3.6}$$

For numerical stability, a constant minimum variance $\epsilon = 1\text{e}{-6}$ is added; the result is then used in Eq. 3.5 (c.f. Sec. A.10).

3.4. Deep Ensembles

Lakshminarayanan et al. [2017] further suggest to improve the model performance by training multiple models and applying them jointly as a uniformly weighted mixture model. The final output is computed as the mean and variance of the mixture $M^{-1} \sum \mathcal{N}\left(\mu_{\theta_m}(\boldsymbol{X}), \sigma_{\theta_m}^2(\boldsymbol{X})\right)$ with

$$\mu_*(\boldsymbol{X}) = M^{-1} \sum_m \mu_{\theta_m}(\boldsymbol{X}) \tag{3.7}$$

and

$$\sigma_*^2(\boldsymbol{X}) = M^{-1} \sum_m \left(\sigma_{\theta_m}^2(\boldsymbol{X}) + \mu_{\theta_m}^2(\boldsymbol{X})\right) - \mu_*^2(\boldsymbol{X}), \tag{3.8}$$

where M is the number of models in the mixture and θ_m are the parameters of the m-th model in the mixture. To test whether this can further improve model performance, $M = 5$ models, each randomly initialized, were trained using Eq. 3.5 for optimization. Similar to Lakshminarayanan et al. [2017], each model was trained individually, and only afterwards were their outputs combined to yield the final outputs $\mu_*(\boldsymbol{X})$ and $\sigma_*^2(\boldsymbol{X})$.

This section extends the ideas by [Lakshminarayanan et al., 2017] further and to better utilize the strengths of their approach.

Each model in the mixture predicts a variance $\sigma^2_{\theta_m}(\boldsymbol{X})$ for each node and target. In [Lakshminarayanan et al., 2017] this is - apart from its role in the loss of each model - only used as a convenience for the end user to assess the reliability of the point predictions. However, it would be natural to include this uncertainty in the way that the mean and variance of the mixture are calculated. Instead of a uniform weighting, one could assign a higher weight to models that are 'more convinced' of their result, that is models which predicted a lower variance. Meaningful weights could be derived for each node and target variable by applying a softmax function to the negative of the variances (such that low variances yield a high weight) predicted by each model in the ensemble, i.e.

$$w_{\theta_m,i,t} = \frac{e^{-\sigma^2_{\theta_m,i,t}}}{\sum_{k=1}^{K} e^{-\sigma^2_{\theta_k,i,t}}} \forall m = 1, ..., M \tag{3.9}$$

This yields a weight distribution across the M models of the ensemble for a specific node i and target t. The weighting can thus be different for each node and target. Finally, the mixture mean and variance (c.f. Sec. A.11) can incorporate those weights with

$$\mu_*(\boldsymbol{X}) = \sum_m w_{\theta_m} \mu_{\theta_m}(\boldsymbol{X}) \tag{3.10}$$

and

$$\sigma^2_*(\boldsymbol{X}) = \sum_m w_{\theta_m} \left(\sigma^2_{\theta_m}(\boldsymbol{X}) + \mu^2_{\theta_m}(\boldsymbol{X})\right) - \mu^2_*(\boldsymbol{X}). \tag{3.11}$$

Applying this weighting scheme on the individually trained models only marginally changed the results, because the predicted variances for a given node and target did not vary much across models. This caused the calculated weights to also be close to the default uniform weight M^{-1}.

This may be due to the way the models are trained. In the mixture model of Lakshminarayanan et al. [2017], each model is optimized individually, which means that each model is encouraged to be a 'jack of all trades', i.e. make good predictions for every node and target variable. Intuitively, it could be better to train a set of 'specialist' models, which could focus on e.g. a certain target variable or specific types of nodes or samples. To realize this, the models would need to be optimized jointly, which can be done by first calculating the model weights with Eq. 3.9, then calculating a joint weighted mean and variance with Eq. 3.10 and 3.11, and then using these in the probabilistic loss function in Eq. 3.5. The

loss function is then only applied on the output of the whole ensemble, not on the individual models.

This allows models in the ensemble to focus on specific things as it does not require them to make decent predictions across all nodes and targets. E.g. a model could focus on a specific target variable, where it predicts a low variance, such that it is weighted highly and contributes a lot to this specific ensemble output; simultaneously it can predict a high variance for other targets such that its output for these is neglected in the ensemble. What a model focuses on is left completely open this way.

A possible issue in deep ensembles with such uncertainty-based weighting is that some of the models may predict high variances across all nodes and targets and thereby receive low weights in the ensemble. As their contribution to the final output vanishes, their gradient updates are diminished as well. This phenomenon is from hereon referred to as model collapse or inactivity.

3.4.1. Monitoring during Training

During training it would be interesting to monitor if and how the models in the ensemble use their freedom to specialize, either on certain target variables, or certain kinds of nodes or samples, or both. This would be reflected in the weights of each model obtained from the softmax function in Eq. 3.9. To do this, the weights obtained for each model and node were accumulated across the validation dataset after each epoch. Then, the mean and standard deviation of the model weights were calculated across all nodes for the different targets with

$$\mu_{w_{\theta_m},t} = N_d^{-1} \sum_{i=1}^{N_d} w_{\theta_m,i,t} \forall t = 1, ..., T \tag{3.12}$$

and

$$\sigma_{w_{\theta_m},t} = \sqrt{\frac{1}{N_d - 1} \sum_{i=1}^{N_d} \left(w_{\theta_m,i,t} - \mu_{w_{\theta_m},t} \right)^2} \forall t = 1, ..., T, \tag{3.13}$$

where N_d is the number of nodes in the validation dataset.

The means show the average contribution of each model to each of the target variables. Thus, they directly indicate whether contributions of each model to the targets were similar or different; if they differ it shows that models have specialized on certain target variables. Complementary, the standard deviation shows how this contribution differs across the nodes. A low standard deviation would mean that the model contributed rather equally to the ensemble output for all nodes, whereas a high standard deviation would indicate that it contributed more to certain kinds of nodes or nodes from specific samples.

If all models were 'jack of all trades', one would thus expect equal means across targets and models, and low standard deviations. In contrast, 'specialist' models would have different means across the targets or have higher standard deviations. Tracking this for each epoch on the validation dataset allows to monitor how this specialization develops - or if it does not.

3.4.2. Regularization Methods

To addess the issue of model collapse, regularization mechanisms need to be designed to encourage each model to participate in the ensemble prediction in a meaningful way.

3.4.2.1. Mean Weight Regularization

A simple approach might be to encourage each model to contribute equally by directly regularizing the weights. For this, the mean weight of each model across all nodes and targets was calculated with

$$\bar{w}_{\theta_m} = T^{-1} N_s^{-1} \sum_{t=1}^{T} \sum_{i=1}^{N_s} w_{\theta_m,i,t} \forall m = 1, ..., M, \tag{3.14}$$

where T is the number of target variables. Then, the mean weight across all models was calculated with

$$\bar{w} = M^{-1} \sum_{m=1}^{M} \bar{w}_{\theta_m}. \tag{3.15}$$

Deviations from this mean weight could be punished using the MSE loss to get the mean-weight regularization loss (c.f. Sec. A.12):

$$\mathcal{L}_{\text{mw}} = M^{-1} \sum_{m=1}^{M} \left(\bar{w}_{\theta_m} - \bar{w}\right)^2. \tag{3.16}$$

Finally, combining the probabilistic loss on the ensemble with this, the total loss for the deep ensemble could be obtained with

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{NLL}} + \lambda_{\text{mw}} \mathcal{L}_{\text{mw}}, \tag{3.17}$$

where λ_{mw} is a weighting factor for the mean-weight regularization term.

3.4.2.2. HSIC and Participation Loss

The previously discussed regularization only intends to enforce equal contributions by the models in the ensemble. However, even though the models are free to develop into complementary specialists, they are not actively encouraged to do so.

A more sophisticated approach could be to apply the Hilbert-Schmidt Independence Criterion (HSIC) to promote diversity in the ensemble and encourage models to specialize in different target variables or structured subsets of the data (e.g. peripheral vs. central nodes). Ideally, this should enable models to develop expertise in a complementary way and allow the ensemble to be more robust to the heterogeneity of the dataset.

Originally, HSIC is a method based on Reproducing Kernel Hilbert-Space (RKHS) theory and was developed to measure statistical dependence, but has been adopted for a wide range of tasks in Machine Learning (ML), ranging from dimensionality reduction and clustering to optimization problems in DL [Wang et al., 2021].

A Hilbert space $\mathcal{H}$ is defined as a complete inner product space: a vector space, for which an inner product $\langle f, g \rangle$ is defined, that allows measuring lengths and angles between elements. In the case of RKHS, the elements of the space are not vectors in $\mathbb{R}^d$, but functions $f : \mathcal{X} \to \mathbb{R}$. Completeness in this context means that all Cauchy sequences of such functions in the RKHS converge to another function in the space.

Let $\mathcal{X}$ be an input space. Each point $x \in \mathcal{X}$ can be associated with a function $k(x, \cdot) \in \mathcal{H}$ in the RKHS via a mapping $\Phi : \mathcal{X} \to \mathcal{H}$. $\Phi(x) \coloneqq k_x \coloneqq k(x, \cdot)$ where (x) is a fixed parameter of the function k_x and $(\cdot)$ is the argument. So, the point from input space $x \in \mathcal{X}$ is mapped to the function $k_x : \mathcal{X} \to \mathbb{R}, k_x(y) = k(x, y)$, where $y \in \mathcal{X}$.

An RKHS is uniquely defined by a kernel function (or kernel for short) $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, which defines the inner product in $\mathcal{H}$ and satisfies the so-called reproducing property:

$$f(x) = \langle f, k(x, \cdot) \rangle_{\mathcal{H}} \tag{3.18}$$

for any function $f \in \mathcal{H}$. That means, evaluating a function f at a point x is equivalent to calculating the inner product of the function with the kernel function $k(x, \cdot)$ at x. Furthermore, the kernel has the property

$$k(x, y) = \langle \Phi(x), \Phi(y) \rangle_{\mathcal{H}} = \langle k(x, \cdot), k(y, \cdot) \rangle_{\mathcal{H}}. \tag{3.19}$$

The kernel function allows the direct computation of an inner product in the high-dimensional feature space $\mathcal{H}$, without explicitly performing the mapping from $\mathcal{X}$ to $\mathcal{H}$ [von Luxburg, 2020], which is illustrated in Fig. 3.6.

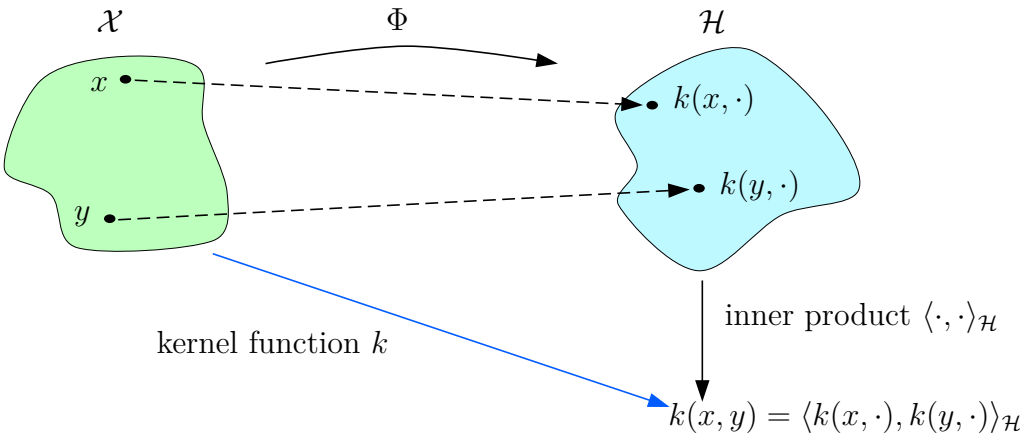


Figure 3.6.: Illustration of input space $\mathcal{X}$, the mapping Φ to the RKHS $\mathcal{H}$ and calculation of the inner product. The kernel function acts as a shortcut for calculating the inner product between functions in $\mathcal{H}$. Adapted from [von Luxburg, 2020].

The HSIC is a measure for statistical dependence using RKHS theory. It quantifies the dependence between two random variables using the Hilbert-Schmidt norm of their cross-covariance operator between two RKHSs. Two random variables are only independent if and only if $\mathrm{HSIC}(\mathrm{P}_{xy}, \mathcal{H}_k, \mathcal{H}_l) = 0$; the higher the HSIC, the higher the dependence.

Let $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ be two random variables drawn jointly from a distribution P_{xy}. Furthermore, let $\Phi : \mathcal{X} \to \mathcal{H}_k$ and $\varphi : \mathcal{X} \to \mathcal{H}_l$ be mappings to RKHSs induced by corresponding reproducing kernels $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ and $l : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$. The population HSIC is then defined as

$$\mathrm{HSIC}(\mathrm{P}_{xy}, \mathcal{H}_k, \mathcal{H}_l) = ||C_{xy}||^2_{HS}, \tag{3.20}$$

where C_{xy} is the cross-covariance operator between $\mathcal{H}_k$ and $\mathcal{H}_l$, and $||\cdot||^2_{HS}$ is the Hilbert-Schmidt norm.

The empirical HSIC can be estimated from a finite number of n observations $D = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^n$ drawn from the joint probability distribution P_{xy}. Using the kernels k and l, two kernel matrices $\mathbf{K}, \mathbf{L} \in \mathbb{R}^{n \times n}$ can be constructed, where $\mathbf{K}_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$ and $\mathbf{L}_{ij} = l(\mathbf{y}_i, \mathbf{y}_j)$. The kernel matrices are centered in feature space by multiplying them on both sides with a centering matrix $\mathbf{H} = \mathbf{I}_n - \mathbf{e}_n\mathbf{e}_n^{\mathrm{T}}/n \in \mathbb{R}^{n \times n}$, where $\mathbf{I}_n \in \mathbb{R}^{n \times n}$ is the identiy matrix, $\mathbf{e}_n \in \mathbb{R}^n$ is a vector of ones, and $\mathbf{e}_n^{\mathrm{T}}$ is its transpose [Wang et al., 2021]. $\mathbf{H}$ centers the kernel matrices in feature space and ensures a mean of zero across the mapped data points. The estimator can then be expressed as

$$\mathrm{HSIC}_b(D, \mathcal{H}_k, \mathcal{H}_l) = \frac{1}{(n-1)}\mathrm{Tr}(\mathbf{HKHHLH}), \tag{3.21}$$

where $\mathrm{Tr}(\cdot)$ is the trace operator. The expression can be slightly simplified, since the centering matrix is idempotent $\mathbf{H}^2 = \mathbf{H}$ and the trace operator is invariant under cyclic permutations, yielding

$$\begin{aligned}
\mathrm{HSIC}_b(D, \mathcal{H}_k, \mathcal{H}_l) &= \frac{1}{(n-1)}\mathrm{Tr}(\mathbf{HKHHLH}) \\
\mathrm{HSIC}_b(D, \mathcal{H}_k, \mathcal{H}_l) &= \frac{1}{(n-1)}\mathrm{Tr}(\mathbf{KHHLHH}) \\
\mathrm{HSIC}_b(D, \mathcal{H}_k, \mathcal{H}_l) &= \frac{1}{(n-1)}\mathrm{Tr}(\mathbf{KH^2LH^2}) \\
\mathrm{HSIC}_b(D, \mathcal{H}_k, \mathcal{H}_l) &= \frac{1}{(n-1)}\mathrm{Tr}(\mathbf{KHLH})
\end{aligned} \tag{3.22}$$

While this estimator is simple, it also introduces a small bias of $O(n^{-1})$ [Gretton et al., 2005], which might be relevant for small sample sizes. To improve this, an unbiased estimator was proposed by [Song et al., 2012], defined as

$$\mathrm{HSIC}_u(D, \mathcal{H}_k, \mathcal{H}_l) = \frac{1}{n(n-3)}\left[\mathrm{Tr}(\tilde{\mathbf{K}}\tilde{\mathbf{L}}) + \frac{\mathbf{e}_n^{\mathrm{T}}\tilde{\mathbf{K}}\mathbf{e}_n\mathbf{e}_n^{\mathrm{T}}\tilde{\mathbf{L}}\mathbf{e}_n}{(n-1)(n-2)} - \frac{2}{n-2}\mathbf{e}_n^{\mathrm{T}}\tilde{\mathbf{K}}\tilde{\mathbf{L}}\mathbf{e}_n\right], \tag{3.23}$$

where $\tilde{\mathbf{K}}$ and $\tilde{\mathbf{L}}$ are the kernel matrices $\mathbf{K}$ and $\mathbf{L}$ with their main diagonal set to 0. This removes self-similarities that introduce a bias to the estimator. Both the biased and unbiased estimators have a computational complexity of $O(n^2)$ due to the calculation of the kernel matrices $\mathbf{K}$ and $\mathbf{L}$. Therefore, the unbiased estimator from Eq. 3.23 was used (c.f. Sec. A.14).

A common choice for the kernel functions k and l is the Radial Basis Function (RBF) kernel, because it is able to capture non-linear dependencies in the input space. It is given by

$$k(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\frac{||\mathbf{x}_i - \mathbf{x}_j||^2}{2\beta^2}\right), \tag{3.24}$$

where $||\cdot||^2$ is the Euclidean norm, and β is a hyperparameter (here set to $\beta = 1$) (c.f. Sec. A.13).

To apply the unbiased HSIC as a loss function for a deep ensemble, it was calculated between each model pair in the ensemble. For an ensemble of M models, $\binom{M}{2} = \frac{M!}{2!(M-2)!}$ combinations had to be evaluated. The results were aggregated and normalized by the number of evaluated combinations. The predicted variances by both models in a pair were used as random variables $\mathcal{X}$ and $\mathcal{Y}$ and the RBF kernel was used to calculate the kernel matrices. The predicted variances originally had the form of $\boldsymbol{\sigma}^2_{\theta_m} \in \mathbb{R}^{N_b \times T}$ where N_b is the number of nodes in a batch, and T is the number of target variables. This matrix was flattened to

$\boldsymbol{\sigma}^2_{\theta_m} \in \mathbb{R}^{N_b \cdot T}$. In the context of HSIC each vector in the inputs is thus equivalent to the variance of a specific node-target pair (i.e. a scalar). This should encourage the models to have different variance distributions across all node-target pairs. The unbiased HSIC loss for the whole ensemble can be defined as

$$\mathcal{L}_{\text{HSIC}} = \frac{2}{M(M-1)} \sum_{m=1}^{M} \sum_{m'=m+1}^{M} \text{HSIC}_u \left(D^{(m,m')}, \mathcal{H}_k, \mathcal{H}_l \right), \tag{3.25}$$

where $D^{(m,m')} = \left\{ \left(\boldsymbol{\sigma}^2_{\theta_m,i}, \boldsymbol{\sigma}^2_{\theta_{m'},i} \right) \right\}_{i=1}^{N_b}$ is the joint set of predicted variances from model m and m' over a batch of N_b nodes, and $\mathcal{H}_k$ and $\mathcal{H}_l$ are the RKHSs induced by the kernels applied to each model's output (c.f. Sec. A.15).

While this encourages the models in the ensemble to focus on different aspects of the data, it does not prevent models from collapsing due to overall high variance predictions and diminishing gradients. To address this issue, another loss term was added, from hereon referred to as participation loss $\mathcal{L}_{part}$, which was defined as

$$\mathcal{L}_{part} = M^{-1} \sum_{m=1}^{M} \text{ReLU}(\gamma - \bar{w}_{\theta_m}), \tag{3.26}$$

where $\bar{w}_{\theta_m}$ is the mean weight of the m-th model, according to Eq. 3.14, and $\gamma = 0.05$ is a participation threshold. If the mean weight of any model in the ensemble falls below γ, it is penalized. This means, models are not forced to contribute to the ensemble result by equal amounts, but they are forced to contribute at least 5% to the final output in a meaningful way (c.f. Sec. A.16).

These loss functions were then tied together with

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{NLL}} + \lambda_{\text{HSIC}} \mathcal{L}_{\text{HSIC}} + \lambda_{\text{part}} \mathcal{L}_{\text{part}}, \tag{3.27}$$

where λ_{HSIC} and λ_{part} are weighting factors for the unbiased HSIC loss and participation loss.

3.5. Model Pretraining

The task of EEW is a supervised learning problem, where the model learns a mapping from an input (here time series, static data and a graph for each sample) to a given label or target variable (here the IMs and epicenter position). In EEW it is important to make reliable predictions as soon as possible, which requires the model to work on sparse and short inputs, where the signal from the earthquake has reached only the first few stations surrounding it. Therefore, only the first 10s of the time series, randomly shifted 0 to 5s before the first station receives the signal, are used, as explained in Sec. 3.1.1.

While only 10s of the time series are actually used, the complete time series are 2 minutes long and usually contain a signal after the initial 10s. Furthermore, the input in this task is often sparse in the sense that the signal has usually only reached a few stations, while the remaining time series only contain background noise, or even just padded 0s, if the stations are further away and started recording the ground motion later on. This means that the majority of the signal remains unused in the supervised EEW task, even though it could be a useful learning signal for the models.

While this is a limitation imposed by the task of EEW, the rest of the time series could still be used as a means of pretraining the models. Pretraining [Han et al., 2021] refers to dividing the model training into a petraining and a finetuning phase. During the pretraining phase, the model is trained on a larger dataset, usually with a more generic learning task, which helps the model learn general features from the data. These pretrained representations can then be used as a starting point to finetune the model on the downstream task, instead of initializing the parameters from scratch. This makes pretraining a promising approach for this project, since the GI dataset itself is relatively small and most of its signals cannot be used in the downstream task.

The pretraining allows models to transfer general knowledge and learned representations from the pretraining dataset to the downstream task. Pretrained models tend to start from a more favorable point in terms of optimization of model parameters, enhancing their performance on downstream tasks and reducing the need for possibly expensive or hard-to-obtain labeled data for the downstream task.

There is a multitude of pretraining paradigms for different domains that have seen great success, an early example being supervised pretraining on ImageNet [Deng et al., 2009] in Computer Vision (CV). Another paradigm that can be used for pretraining is self-supervised learning, where the model is trained on a pretext task, that does not require any human-made labels. Instead, the labels are derived directly from the input itself, for example by predicting masked out sections in an image or predicting ahead values in a sequence. Self-supervised learning has been successfully applied to all kinds of data domains like CV [Chen et al., 2020; Grill et al., 2020; He et al., 2021a], Natural Language Processing (NLP) [Devlin et al., 2019a; Brown et al., 2020] and time series [van den Oord et al., 2019].

To apply self-supervised learning as a pretraining method using the GI dataset, an appropriate pretext task needed to be defined. For this, another sampling method for the time series was used to also include the sections that cannot be used in the finetuning. The two sampling methods are illustrated in Fig. 3.7. b3) and c3) show the complete 120s time series for the E (East-West) component

of the first (RAVA) and last station (COR1) to receive the signal for earthquake 772141 (c.f. 3.2 and 3.3).

The start and end times of the time series are different, since far away stations receive the signal later and thus the recording starts at a later time as well. Time series, missing recordings in the beginning or end because of this, are padded with 0s (dotted lines). The arrival times of the P wave at each station are shown as dashed lines.

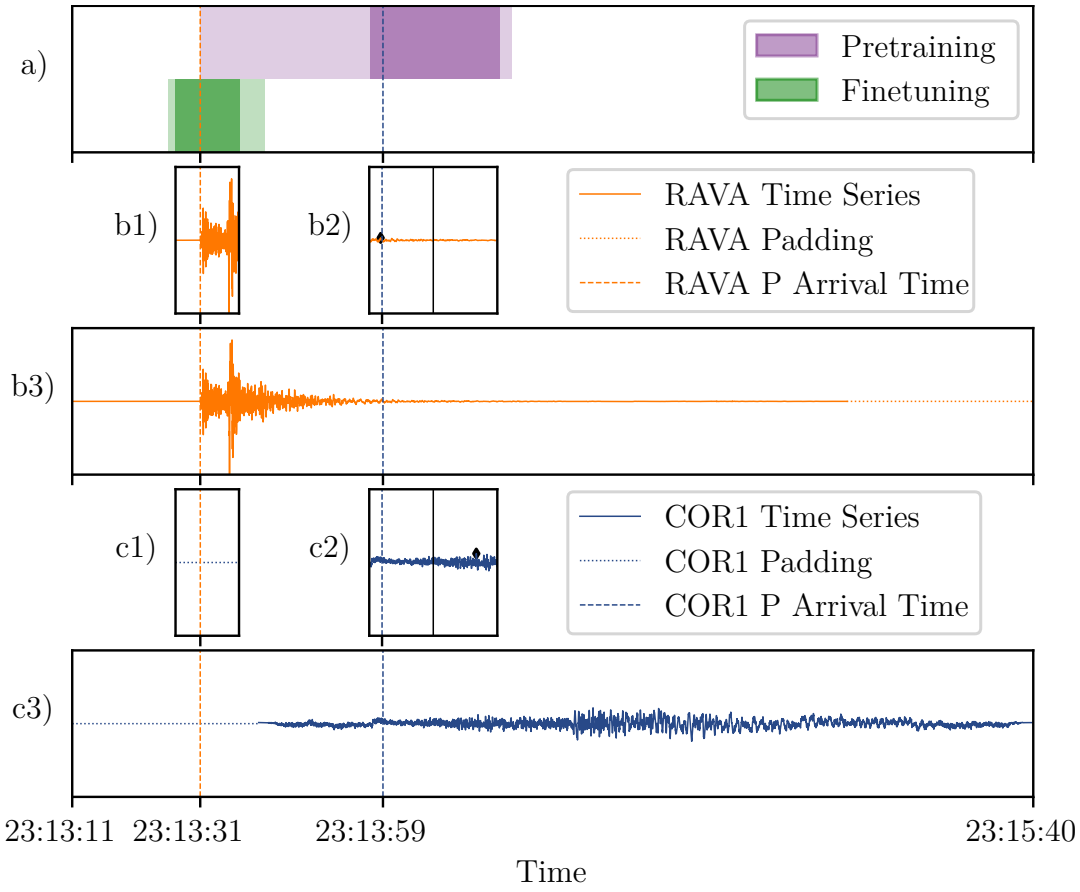


Figure 3.7.: Illustration of the two different sampling methods used for pretraining and finetuning. a) shows the complete sampling time window in light colors and the actual sample time window, which is chosen randomly, in dark colors. b) and c) show the E (East-West) component of the time series for the first station to receive the earthquake signal (RAVA) and the last station (COR1) for earthquake 772141 (c.f. 3.2 and 3.3). b3 and c3 show the complete (padded) time series and arrival times of the P wave. b1 and c1 show the how a randomly selected sample for finetuning would look like. b2 and c2 show how randomly selected samples for pretraining would look like. The model input is the first half of the pretraining time series sample, while the target (maximum absolute acceleration) is obtained from the whole sample (black diamond).

The sampling scheme is illustrated in Fig. 3.7a), where green refers to the finetuning (EEW) task and violet refers to the pretraining task. For finetuning, the time series is trimmed to a 10s window, starting 0 to 5s before the P wave

reaches the first station. The whole sampling time frame has thus a length of 15s (light green), from which a random 10s is chosen (dark green) and all components of all time series in the sample are cut to this 10s time frame, as shown in b1) and c1), which is the input for the models. The targets are the IMs calculated over the whole time series, as contained in the INSTANCE meta data.

For the pretraining task, a random 20s section is cut from the time series (dark violet), within the sampling time frame (light violet) starting with the first P arrival time and ending with the last P arrival time plus 20s. The length of this sampling time frame thus differs from sample to sample. Fig. 3.7b2) and c) show the time series trimmed to the randomly chosen time window from a). The black vertical line splits the sample into two 10s time series. The first 10s are used as inputs to the models; input dimensions are thus the same as in the finetuning. The whole 20s are then used to automatically determine targets for the pretext task. The target is the PGA, i.e. the maximum absolute value of the ground acceleration, indicated by the black diamond. This is done before normalizing the time series, and the PGA is also $\log_{10}$ transformed, similar to the finetuning task. Deriving the targets from the input time series directly may lead to problems in case the time series subsample contains only 0s, which - although rare - occurs in the GI dataset. Specifically, using the described (deterministic) pretraining sampling method, the fraction of nodes with only 0s in their time series subsample are:

- 0.293% (254/86402) for the training split,
- 0.487% (52/10621) for the validation split, and
- 0.394% (44/11127) for the test split.

The $\log_{10}$ transform, applied to the targets, requires them to be strictly positive and non-zero. Otherwise, Pytorch returns `inf` for `torch.log10(0)`, breaking the model optimization. The least complicated way to resolve this issue, was to mask out those few nodes with missing target data before calculating the loss. Nodes with missing target data thus did not contribute to the optimization.

The pretext task of predicting the PGA within a reduced 20s time window is reasonably close to the actual EEW task of predicting IMs for the whole time series, to potentially provide a decent learning signal that can improve model performance in the downstream task of EEW.

3.6. Station Dropout

[Bloemheuvel et al., 2023] use static graphs of the same 39 stations to train their model. However, often not all of the stations actually recorded the earthquake,

for example due to failures of the sensors. The authors then used 0s as input to the model and derived the IMs, used as targets, from simulated data from ShakeMap [Worden et al., 2020]. This way - even though relying on simulated targets - their model may learn to deal with missing inputs in the data and still make decent predictions.

In contrast, the graphs in the GI dataset are dynamic and only include stations that actually recorded the earthquake. However, the GI dataset is not confined to a small area but includes earthquakes, whose epicenters are located even in other countries like Greece or the mediterranean sea. Due to this larger area, the time difference between the arrival times of the P wave at the different stations of a sample may be much higher compared to the datasets [Jozinović et al., 2021; Bloemheuvel, 2021] used by [Bloemheuvel et al., 2023]. This means that the input for further away stations, which receive the signal later, may only be low-amplitude noise or even just padded 0s, as shown in Fig. 3.2 or 3.7c1. So, while the model may learn to make predictions for stations for which there is seemingly no data, it is not directly comparable to [Bloemheuvel et al., 2023], because the reason for the missing data is not a sensor failure but the large distance of some stations to the epicenter. This information is contained in the graph structure of the samples and the models might (or actually should) learn this relation.

An additional step might be necessary to enable the models to learn to handle missing data, caused by sensor failure etc. instead of just the large distance. This can be implemented by randomly masking out individual stations by setting their input time series to 0s. The first station to receive the signal is omitted, the remaining stations are eligible for dropout/masking. This guarantees that at least one valid signal remains in the sample. Another hyperparameter, the station dropout rate, was introduced, which is the fraction of the eligible stations, which is masked out.

Since the dropout is done randomly, the Pytorch dataset class, again, optionally allows to make this deterministic for the validation/test splits, by setting the earthquake ID as a seed for the RNG. If this option is used, then the seed is reset again, before randomly shifting the time series start 0 to 5s as explained in Sec. 3.1.3. This ensures that using station dropout does not change the time series sampling results.

The station dropout was implemented both for the finetuning and pretraining task. Furthermore, the targets for dropped out stations are based on real sensor measurements, instead of just simulated data as in [Bloemheuvel et al., 2023]. Fig. 3.8 shows station dropout applied to earthquake 772141 (c.f. Fig. 3.2, 3.3, and 3.7) with a station dropout ratio of 0.4 using the pretraining sampling scheme.

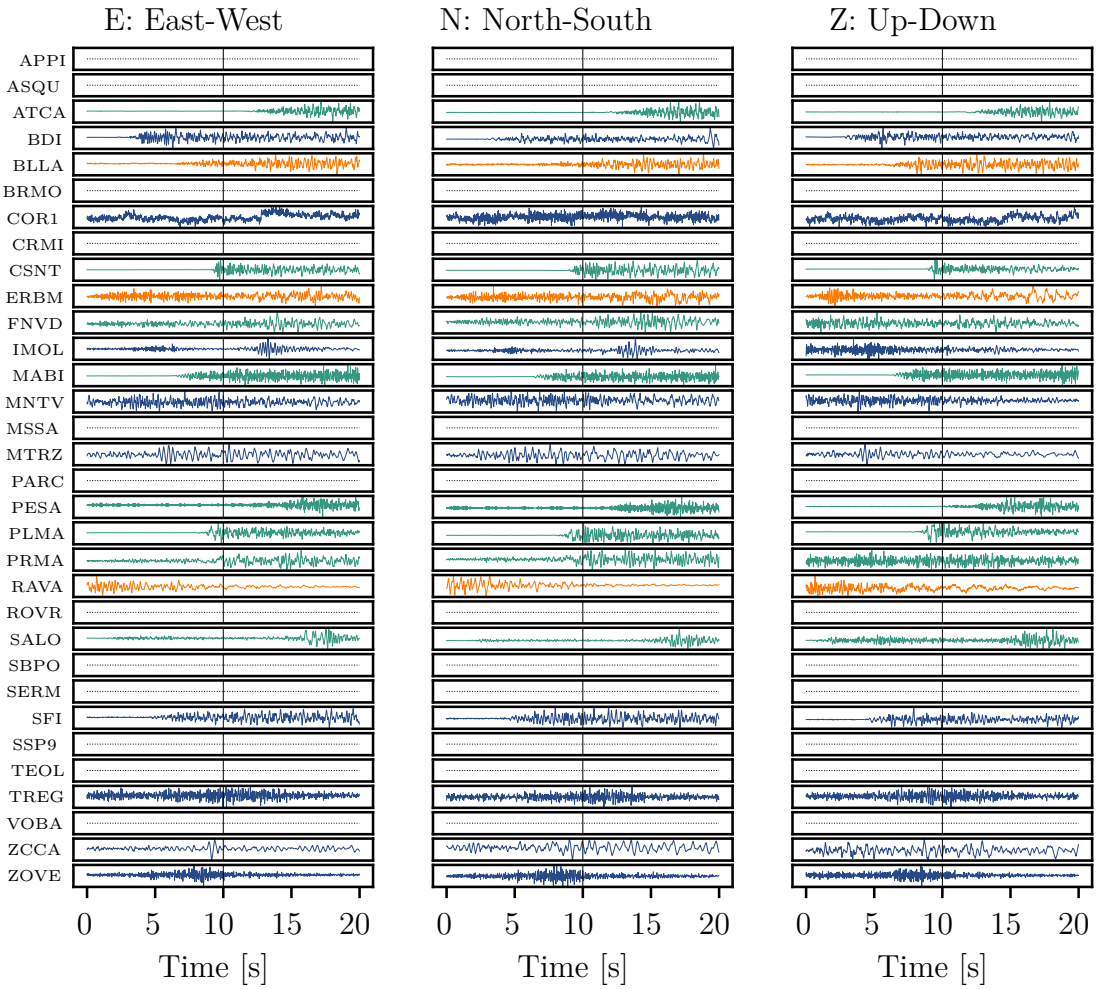


Figure 3.8.: Unnormalized time series for earthquake 772141 (c.f. Fig. 3.2, 3.3, and 3.7) using the pretraining sampling scheme. The first 10s are used as model inputs, while the whole 20s are used for deriving the targets (PGA). Afterwards the station dropout is applied (black dotted lines) using a station dropout rate of 0.4 in this example. The first station to receive the earthquake signal (RAVA) always remains unmasked, the remaining stations are eligible for dropout.

3.7. Creating Earthquake Maps

If a model can be succesfully trained to handle missing input time series of a fraction of the stations, e.g. through station dropout, it could possibly be able predict the earthquake intensity in other locations not contained in the graph. To achieve this, an artificial node, i.e. a location without an actual monitoring station, could be inserted into the graph, with only the geographical information and input time series set to 0s. Extending this idea, one could generate earthquake intensity maps by adding artificial nodes in a grid structure and plotting the results in a heatmap.

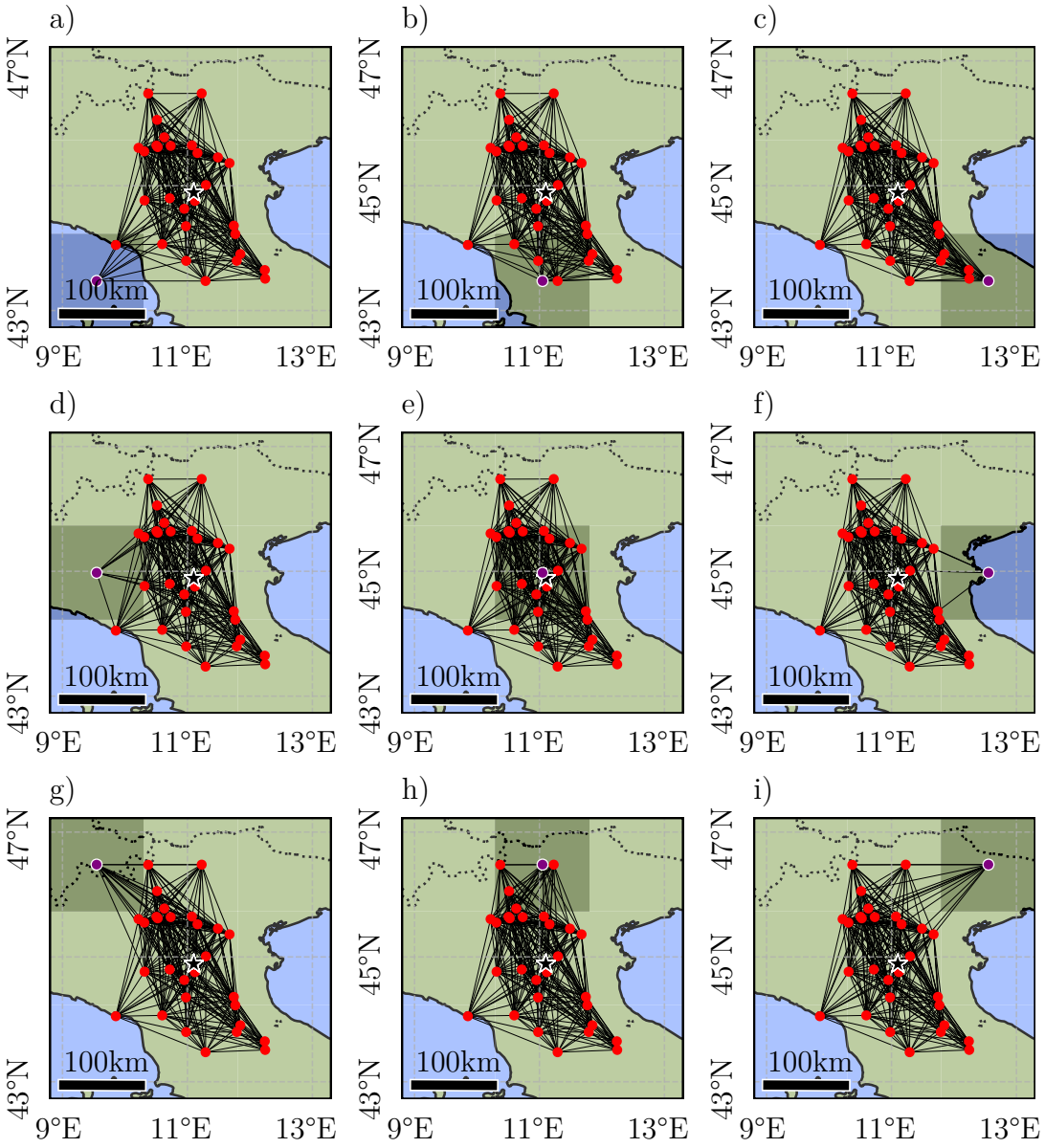


Figure 3.9.: Illustration of the sampling scheme for earthquake maps using sample 778161 and a grid size of $g = 3$. The regular graph of the sample is shown with nodes in red and edges as black lines. The earthquake epicenter is shown as a black star. The artificial node is shown in purple; the shaded square is the area in the earthquake map covered by that node. Map projection: Plate Carrée.

To achieve this, the Pytorch dataset class was extended to allow the insertion of a single artificial node. An iterator class was developed, which takes a sample from the chosen dataset and computes latitude-longitude-coordinate-pairs in a grid of desired resolution. It then allows iterating over this grid, where each iteration returns the sample with one of the coordinate-pairs added as artificial node in the graph. For each artificial node, one forward-pass of the model is applied to get the intensity prediction for that specific location.

More specifically:

1. **Coordinate grid sampling scheme**
   - The iterator class receives the dataset class, a sample ID, and the desired resolution g of the grid (e.g. $g = 3$ for a 3×3 grid).
   - The min/max latitude and longitude across all nodes of the sample are found and the latitude and longitude range is calculated with $D_{\text{lat}} = \max(\text{lat}) - \min(\text{lat})$ and $D_{\text{lon}} = \max(\text{lon}) - \min(\text{lon})$.
   - Then a step size s is calculated with $s = \max(D_{\text{lat}}, D_{\text{lon}})/(g-1)$.
   - If $D_{\text{lat}} \geq D_{\text{lon}}$, g latitudes are sampled equidistantly between min(lat) and max(lat) with a step size of s. Then g longitudes are sampled equidistantly between $\min(\text{lon}) - (D_{\text{lat}} - D_{\text{lon}})/2$ and $\min(\text{lon}) - (D_{\text{lat}} - D_{\text{lon}})/2 + s \cdot (g-1)$ with a step size of s. If $D_{\text{lat}} \leq D_{\text{lon}}$, the calculation of latitudes and longitudes is reversed.
   - Finally, a grid was created by taking all possible combinations of the sampled latitudes and longitudes. This sampling scheme finds the smallest possible square coordinate grid to contain the whole graph.
2. **Calculation of the earthquake intensity map**
   - The iterator class iterates over all coordinate pairs in the grid and calls the Pytorch dataset class for each coordinate pair:
     - The graph was constructed, edges were filtered out, and the input time series of the regular sample was obtained.
     - An artificial node was added to the graph, located at the coordinate pair, and edge weights to all other nodes were calculated. Then the new weights were transformed using the parameters of the regular graph as in Eq. 3.1.
     - If a new edge was longer than the longest edge in the regular graph, this transformation makes the new edge weight negative. To prevent this, the new edge weights w were clamped to a small positive weight with $A_{i,j}^{(s)} \leftarrow \max(A_{i,j}^{(s)}, 0.01)$.
     - The new edges were then filtered using the same criteria as for the regular graph.
     - The input time series of the new station were set to 0s.
     - To obtain the static input data, the coordinate pair was translated to relative coordinates, similar to the regular nodes, and normalized using the normalization factor of the regular graph. Then the

altitude of this location was obtained using the Open-Meteo API [Zippenfenig, 2023].

- The returned modified sample was then given to the model to predict the earthquake intensity for a single point in the grid. This was done for each coordinate pair in the grid. A grid size of $g = 3$ would thus require $3 \cdot 3 = 9$ forward passes of the model to generate the earthquake map.
- The predictions were then aggregated into a grid and plotted using Matplotlib [Hunter, 2007], Cartopy [Elson et al., 2024], Networkx [Hagberg et al., 2008], Geopy [Geopy Contributors, 2008] and Shapely [Gillies et al., 2025].

Fig. 3.9 illustrates the sampling scheme for the earthquake maps, using earthquake 778161 from the test dataset and a grid size of $g = 3$. The red nodes are actual earthquake monitoring stations; the purple node is the artificial node. The model processes each of the 9 modified graphs shown indidivually and the predictions for the purple node are gathered. The shaded square shows the area in the earthquake map, covered by the artificial node.

3.8. Model Selection and Evaluation

For model selection during training, the MSE loss on the deterministic validation dataset was used. Specifically, the padding of the time series and masking for station dropout were kept constant across epochs. The loss was calculated batch-wise with a batch size of 4, averaged over the nodes in each batch, and accumulated in a running total. The final validation loss was obtained by dividing this total by the number of batches. This approach is computationally efficient but may introduce minor rounding errors due to repeated division and accumulation. If the model achieved a lower validation loss than in previous epochs, the weights were saved for further evaluation.

To provide a more robust evaluation of the model from the best-performing epoch, bootstrapping [Efron, 1992] was applied to the test dataset to estimate the uncertainty of performance metrics. Model outputs were aggregated across each dataset split, yielding 2-dimensional arrays of shape $N_{\text{nodes}} \times N_{\text{targets}}$. The MSE loss was calculated separately for each individual target variable, as well as its average over all targets, which is the metric used for model selection.

Bootstrapping is used to estimate the uncertainty and variability in model performance due to the influence of individual samples (here nodes). 1000 bootstrap samples were generated, each with the same shape $N_{\text{nodes}} \times N_{\text{targets}}$, by drawing nodes randomly with replacement from the original arrays. Consequently, some

nodes may occur multiple times within a bootstrap sample, while others may be excluded. For each bootstrap sample, the MSE loss was calculated for each target and their average, resulting in 1000 loss values per target. These loss distributions may reflect the variability in model performance across different subsets of nodes.

To construct confidence intervals, the differences between the bootstrap losses and the original loss were calculated. The 2.5th and 97.5th percentiles of these differences were used to define a 95% confidence interval, indicating the range within which the true model performance is expected to lie with 95% probability. Additionally, the standard deviation of the differences was calculated to quantify the spread and variability of the performance metric.

To get more robust results, the evaluation was done using different fixed paddings $\{0\text{s}, 1\text{s}, 2\text{s}, 3\text{s}, 4\text{s}, 5\text{s}\}$ for the time series. A padding of 0s means that the 10s time window starts with the earthquake arriving at the station to first receive the signal. A padding of 5s would mean that the earthquake reaches the first station after 5s, such that half of the signal is noise. A higher padding thus increases the difficulty of the task.

4. Experiments and Results

4.1. Learning Rate

The learning rate (lr), determining the step size in each optimization step of the model parameters, is a critical hyperparameter. As a first step, the baseline model (c.f. Sec. A.1) was optimized using different learning rates lr $\in \{0.01, 0.001, 0.0001\}$, without filtering out edges. Optimization was done both on the node-level and the auxiliary graph-level task. In all experiments, the training was carried out for 200 epochs, using ADAptive Moment estimation (Adam) as optimizer [Kingma and Ba, 2017].

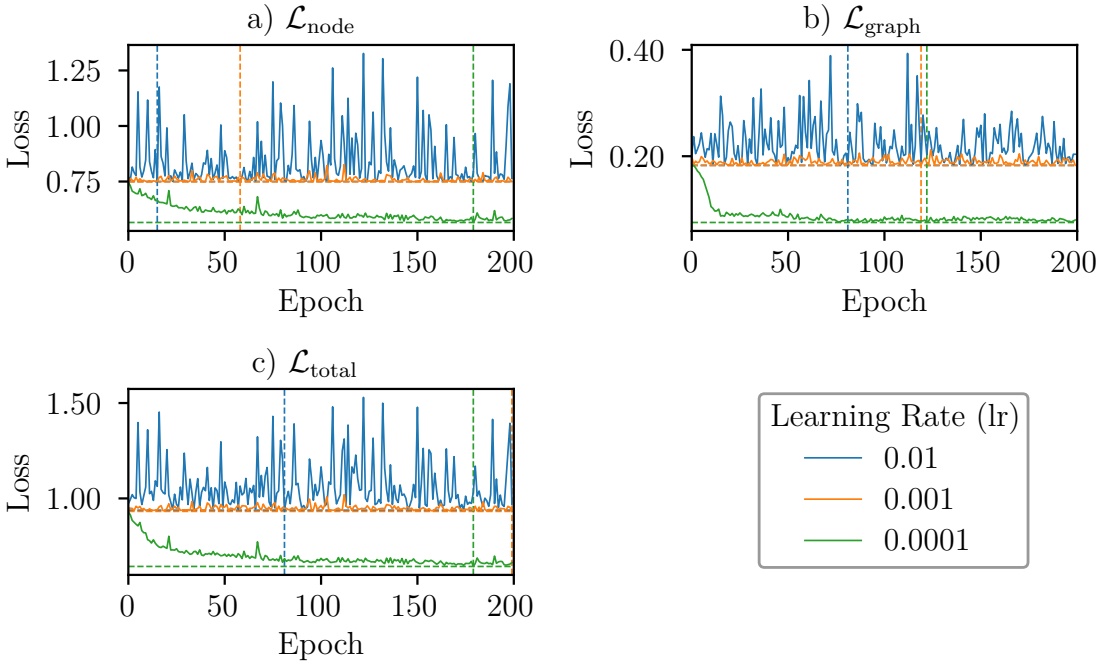


Figure 4.1.: Validation loss of the model during training for different lr for the node-level, graph-level, and total loss as in Eq. 3.2 - 3.4. The dashed lines represent the lowest validation loss achieved in that configuration.

Fig. 4.1 shows the validation loss according to Eq. 3.2 - 3.4. Tab. 4.1 shows the results of $\mathcal{L}_{\text{node}}$ for each lr and time-series padding; the overall best result for each time series padding is highlighted in bold. The presented results are averaged across the targets (IMs) to not convolute the paper. Fig. 4.2 visualizes the results from Tab. 4.1 as errorbars and the mean loss $\mu_{\mathcal{L}_{\text{node}}}$ as a dashed line.

Table 4.1.: MSE loss (Eq. 3.2) on the test dataset of models trained on different learning rates. $\mathcal{L}_{\text{node}}$ is the average across all target variables, which is the metric used for model selection. M is the metric, U and L are the upper and lower bounds of the 95% confidence interval. The best result for each time-series padding is highlighted in bold.

		$\mathcal{L}_{\text{node}}$ **for different Time-Series Paddings**						
lr		0s	1s	2s	3s	4s	5s	$\mu_{\mathcal{L}_{\text{node}}}$
0.01	U	0.7630	0.7630	0.7630	0.7630	0.7630	0.7630	0.7439
	M	0.7439	0.7439	0.7439	0.7439	0.7439	0.7439	
	L	0.7240	0.7240	0.7240	0.7240	0.7240	0.7240	
0.001	U	0.7606	0.7606	0.7606	0.7606	0.7606	0.7606	0.7414
	M	0.7414	0.7414	0.7414	0.7414	0.7414	0.7414	
	L	0.7215	0.7215	0.7215	0.7215	0.7215	0.7215	
0.0001	U	0.5406	0.5415	0.5524	0.5708	0.5861	0.5942	**0.5500**
	M	**0.5265**	**0.5277**	**0.5386**	**0.5559**	**0.5712**	**0.5799**	
	L	0.5126	0.5143	0.5249	0.5406	0.5566	0.5654	

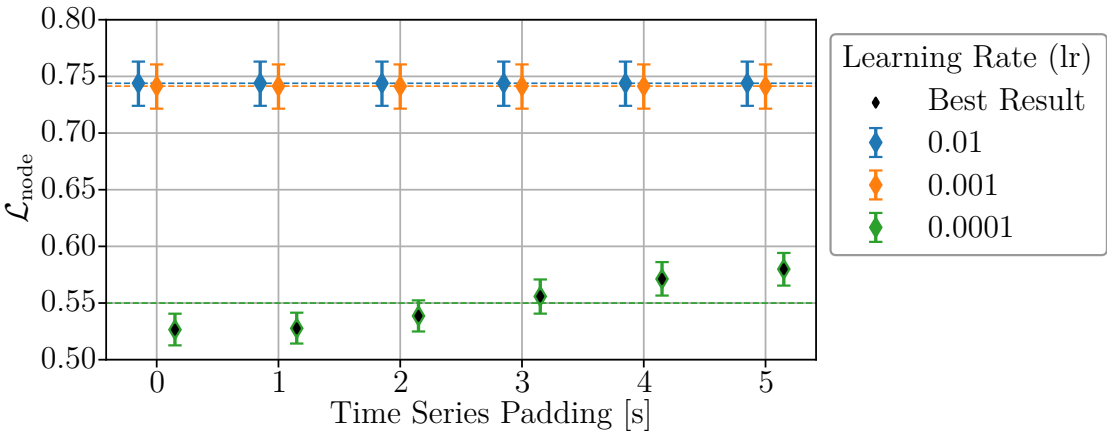


Figure 4.2.: Visualization of Tab. 4.1. The markers show $\mathcal{L}_{\text{node}}$ of the test dataset for different time series paddings. The caps mark the corresponding 95% confidence interval.

A slight horizontal offset has been applied to the markers relative to the time series paddings (shown as vertical lines) solely to facilitate the visual comparison of different models within a single figure.

A learning rate of 0.0001 performed best, while the model did not significantly converge further after the first epochs for the higher learning rates. Investigating the predictions for the models trained with learning rates of 0.01 and 0.001 revealed that the models collapsed and predicted identical values across all nodes. For the model trained with learning rate 0.0001, the loss increased with the padding, as could be expected, since the actual earthquake signal becomes shorter. In further experiments, a learning rate of 0.0001 was used. Training scripts can be found in the supplementary material [Zender, 2025].

4.2. Earthquake Epicenter Prediction

Fig. 4.3 shows the results on the test dataset for the auxiliary task of earthquake epicenter prediction, based on the model trained using a lr of 0.0001 from the previous experiment. For evaluation, a fixed time series padding of 2.5s was used. a) shows the distance between predicted and ground truth (dot) epicenter as lines. b) and c) show this distance (i.e. the error) plotted against b) the number of stations that were reached by the P wave within the time window, and c) the distance between the random reference point and the real epicenter.

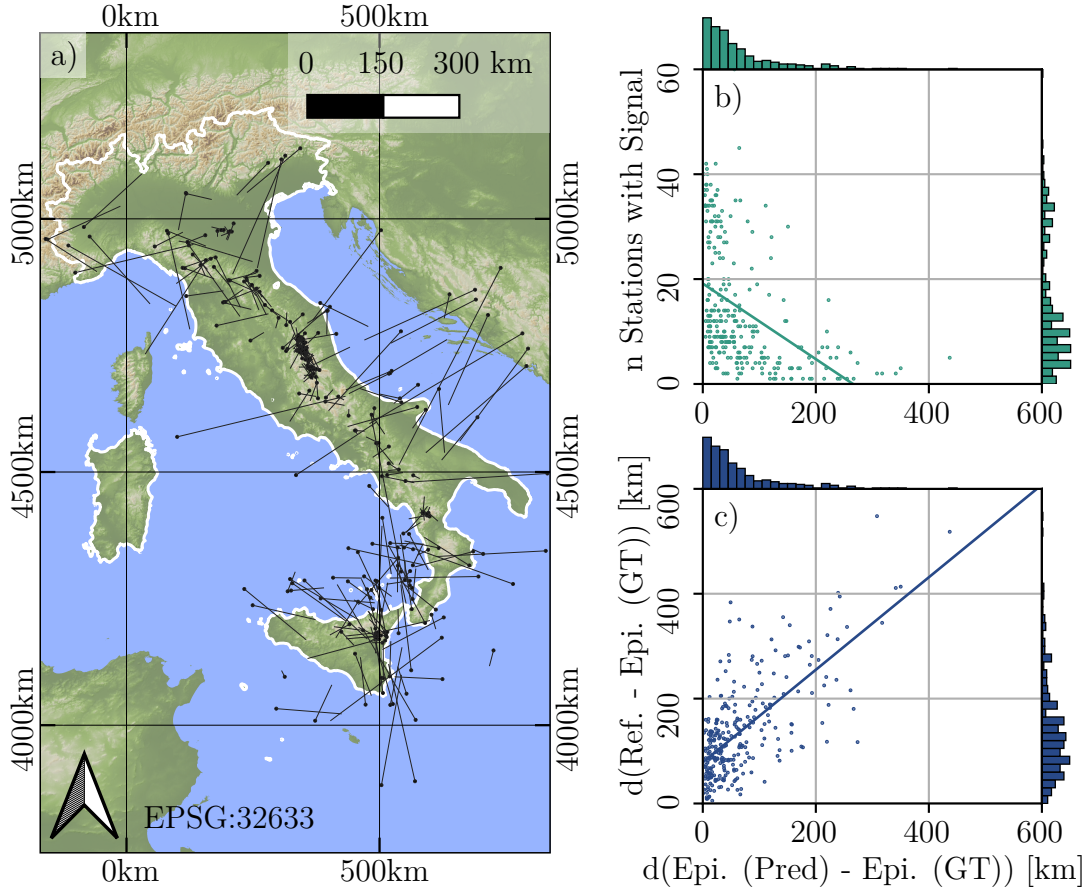


Figure 4.3.: a) Distances between predicted (Pred) and ground truth (GT) epicenters. The ground truth positions are marked by the dots. b) These distances vs. the number of stations that recorded a signal within the first 7.5s. c) These distances vs. the distance between the random reference point and real epicenter.

As can be seen, the error increases as the distance between real epicenter and reference point increases (c). This is especially the case for samples, where the real epicenter is far offshore in the Mediterranean Sea or in neighboring contries like Croatia (a). Furthermore, the fewer stations received the signal in the given time (b), the higher the error, as could be expected. Even though most errors remain rather small, there are some samples for which the deviation is very strong. Thus, the method does not yet yield reliable results.

4.3. Graph Filter Methods

4.3.1. Minimum Node Degree and Graph Diameter

This section briefly investigates graph statistics to identify challenging aspects in filtering the dynamic graphs of the GI dataset and problems in the proposed filtering methods.

Fig. 4.4 shows the graph diameter for each sample in the dataset for a minimum node degree d in $[1, 50]$ and without filtering; the bar plot shows the number of nodes N_s for each graph $\mathcal{G}_s$. The graph diameter can be imagined as the longest shortest-path in the graph. A diameter of > 2 means that not every node can reach every other node within 2 steps of MP. The figure shows that the dataset is highly diverse, and that the first approach, using the minimum node degree d, can lead to graphs with a high diameter. This appears to be worse for large graphs, as this filtering method may leave the graph well-connected locally but without long-range connections.

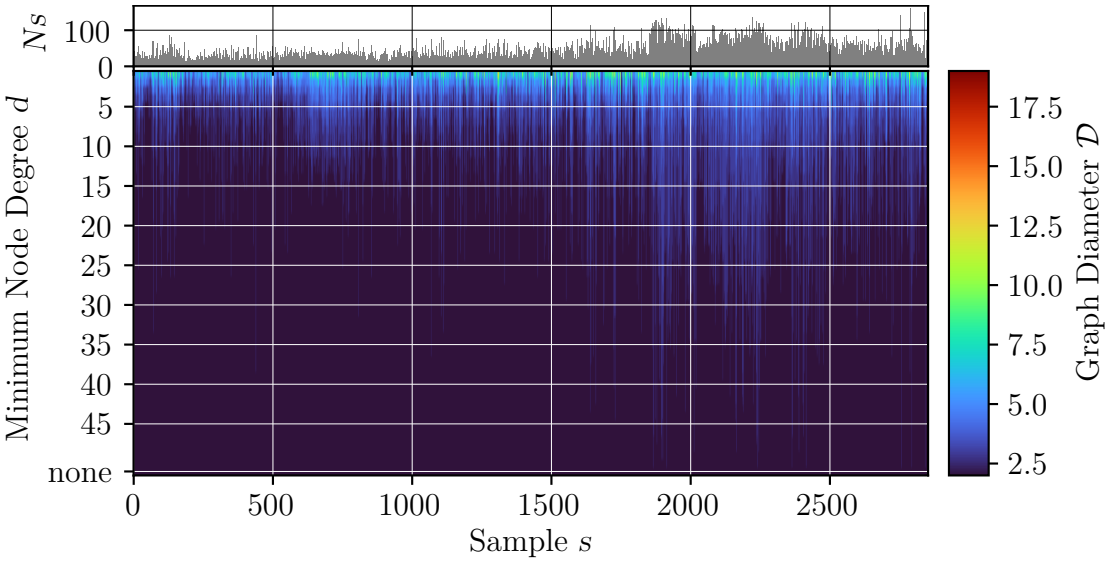


Figure 4.4.: Graph diameter $\mathcal{D}$ for each sample s in the dataset for a minimum node degree d in $[1, 50]$ and without filtering. The bar plot shows the number of nodes N_s for each sample s.

The graph diameter is sensitive to small local structures; e.g. a number of nodes connected in a chain drastically increases the graph diameter. A more representative way of evaluating the effects of the minimum node degree would be to use an average measure, here called the 2-hop-reachability as shown in Fig. 4.5. It is the fraction of nodes that an average node in a graph can reach within 2 MP steps. The value for the minimum node degree d, for which information can travel well through the graph, appears to be highly dependent on the individual

graphs. This suggests that for a diverse dataset with dynamic graphs, this may not be the best approach.

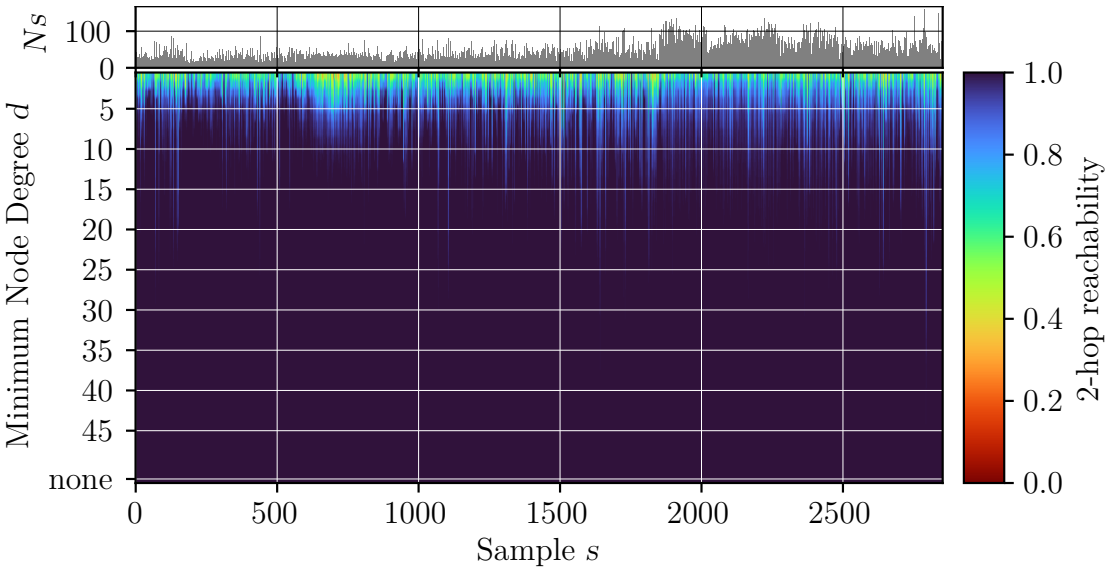


Figure 4.5.: 2-hop-reachability for each sample s in the dataset for a minimum node degree d in $[1, 50]$ and without filtering. The bar plot shows the number of nodes N_s for each sample s.

The second approach to filtering used the graph diameter $\mathcal{D}$ directly as a stopping criterium; i.e. if the removal of the lowest weighted edge increases the graph diameter above a threshold (i.e. 2 for 2 GCN layers), then it is not removed and the filtering is stopped. In contrast to the first approach, this method ensures that in any graph, any node can exchange information with any other node. This could further be adjusted to match the number of layers of the GNN.

4.3.2. Experimental Comparison of Graph Filter Methods

The baseline model was optimized using only the node-level loss from Eq. 3.2 (c.f. Sec. A.2) and the minimum node degree filtering with $d \in \{3, 6, 9\}$, where $d = 9$ achieved the lowest validation loss and $d = 6$ achieved the lowest test loss. The experiment was repeated using the maximum graph diameter method (maximum diameter $\mathcal{D} = 2$) and using the unfiltered graph for comparison. Finally, the model was trained on both approaches at once, using a maximum graph diameter $\mathcal{D} = 2$ and minimum node degree $d = 9$.

Fig. 4.6 visualizes the test loss for different time series paddings and filter methods (c.f. Tab. 4.2) as errorbars. Overall, the differences in the test loss were not large on average; the unfiltered graph performed worst. The minimum edge degree filtering performed slightly better for larger time series paddings. The lowest test loss was achieved by using a maximum graph diameter $\mathcal{D} = 2$ (identical to the number of GCN layers in the baseline model) as filter method, especially

for small time series paddings. Applying both filter methods simultaneously did not improve the results.

Table 4.2.: MSE loss (Eq. 3.2) on the test dataset of models trained using different edge filter methods (d: minimum node degree, $\mathcal{D}$: graph diameter). $\mathcal{L}_{\text{node}}$ is the average across all target variables, which is the metric used for model selection. M is the metric, U and L are the upper and lower bounds of the 95% confidence interval. The best result for each time-series padding is highlighted in bold.

Filter Method		$\mathcal{L}_{\text{node}}$ **for different Time-Series Paddings** 0s	1s	2s	3s	4s	5s	$\mu_{\mathcal{L}_{\text{node}}}$
None	U	0.5655	0.5513	0.5450	0.5554	0.5716	0.5912	0.5494
	M	0.5511	0.5376	0.5321	**0.5422**	0.5575	0.5759	
	L	0.5367	0.5239	0.5187	0.5282	0.5431	0.5609	
$d = 3$	U	0.5444	0.5464	0.5479	0.5729	0.5726	0.5863	0.5479
	M	0.5310	0.5316	0.5345	0.5591	0.5591	0.5724	
	L	0.5168	0.5177	0.5201	0.5448	0.5442	0.5585	
$d = 6$	U	0.5572	0.5470	0.5486	0.5568	0.5650	0.5820	0.5448
	M	0.5420	0.5320	0.5341	0.5426	**0.5510**	**0.5670**	
	L	0.5275	0.5178	0.5201	0.5285	0.5361	0.5528	
$d = 9$	U	0.5492	0.5477	0.5453	0.5565	0.5669	0.5883	0.5451
	M	0.5352	0.5337	**0.5317**	0.5424	0.5536	0.5740	
	L	0.5207	0.5195	0.5176	0.5284	0.5393	0.5588	
$\mathcal{D} = 2$	U	0.5397	0.5367	0.5499	0.5686	0.5691	0.5825	**0.5437**
	M	**0.5255**	**0.5227**	0.5361	0.5539	0.5555	0.5688	
	L	0.5110	0.5087	0.5220	0.5390	0.5413	0.5548	
$\mathcal{D} = 2$, $d = 9$	U	0.5524	0.5532	0.5504	0.5578	0.5709	0.5934	0.5494
	M	0.5387	0.5388	0.5371	0.5445	0.5576	0.5800	
	L	0.5247	0.5245	0.5236	0.5309	0.5438	0.5653	

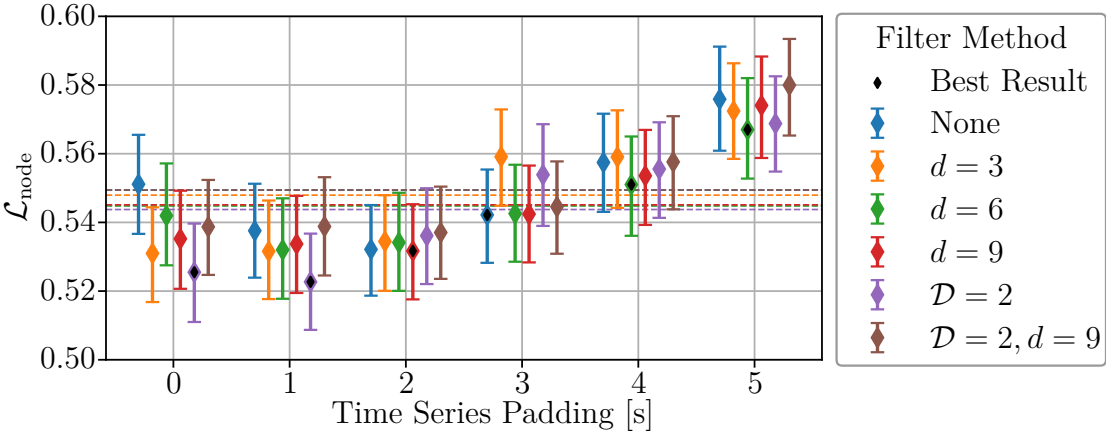


Figure 4.6.: Visualization of Tab. 4.2. The markers show $\mathcal{L}_{\text{node}}$ of the test dataset for different time series paddings. The caps mark the corresponding 95% confidence interval.

Overall, filtering out low-weighted edges did not drastically improve the (average) test loss, but still generally outperformed the unfiltered graph. A more sophisticated approach might improve this further, but the inhomogeneity of the graphs in the dataset, both in terms of number of nodes, as well as their spatial distribution, remains a challenge for developing a general filter method.

In further experiments, models were trained using a maximum graph diameter of $\mathcal{D} = 2$ and the results of this experiment were used as a reference model for assessing performance.

4.4. Static Features

In this experiment, the influence of the static features, i.e. station position and elevation, on model performance was evaluated. One model was trained without any static features (c.f. Sec. A.3). For simple concatenation of the static features with the time-series features derived from the CNN, the reference model from the previous experiment (c.f. Sec. 4.3.2) was used. Finally, a third model was trained, where the static features are first processed through a small 3-layer Multilayer Perceptron (MLP), before the output features are concatenated with the CNN output (c.f. Sec. A.4). The MLP maps the three static input features to 16, 32, and finally 64 output features. After each linear layer, ReLU was used as an activation function and after the first two linear layers, 1D Batch Normalization (BN) layers and dropout layers with a 0.3 dropout ratio were applied. The input size of the first GCN layer was changed according to each model variation.

Table 4.3.: MSE loss (Eq. 3.2) on the test dataset of models trained without static data, with concatenated static data or concatenated static data processed in an MLP. $\mathcal{L}_{\text{node}}$ is the average across all target variables, which is the metric used for model selection. M is the metric, U and L are the upper and lower bounds of the 95% confidence interval. The best result for each time-series padding is highlighted in bold.

Static Features		$\mathcal{L}_{\text{node}}$ **for different Time-Series Paddings** 0s	1s	2s	3s	4s	5s	$\mu_{\mathcal{L}_{\text{node}}}$
None	U	0.5492	0.5490	0.5530	0.5579	0.5673	0.5718	0.5443
	M	0.5354	0.5354	0.5392	0.5443	0.5538	0.5576	
	L	0.5214	0.5221	0.5258	0.5303	0.5393	0.5434	
Concat.	U	0.5397	0.5367	0.5499	0.5686	0.5691	0.5825	0.5437
	M	0.5255	0.5227	0.5361	0.5539	0.5555	0.5688	
	L	0.5110	0.5087	0.5220	0.5390	0.5413	0.5548	
MLP + Concat.	U	0.5379	0.5296	0.5301	0.5436	0.5563	0.5686	**0.5307**
	M	**0.5238**	**0.5161**	**0.5167**	**0.5301**	**0.5428**	**0.5550**	
	L	0.5099	0.5023	0.5036	0.5166	0.5288	0.5407	

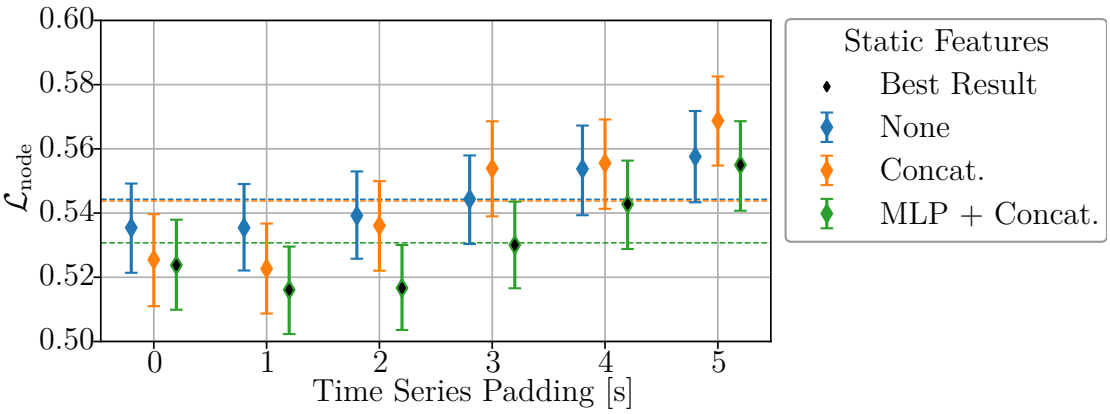


Figure 4.7.: Visualization of Tab. 4.3. The markers show $\mathcal{L}_{\text{node}}$ of the test dataset for different time series paddings. The caps mark the corresponding 95% confidence interval.

Fig. 4.7 shows the experimental results from Tab. 4.3 as errorbars. Simple concatenation of input features yielded better results for small time-series paddings but not for large paddings, compared to the model trained without any static features. Both models were clearly outperformed for all paddings by the third model including the MLP for preprocessing of the static data.

Thus, including static data did not appear to make a large difference to the average model performance unless it was processed by an MLP first.

4.5. Concatenation of Raw Tokens

Ideally, the CNN captures all relevant information from the raw time-series tokens and encodes them in its output features. This experiment tested whether there was complementary information still left within the raw time series that was not contained in the CNN features. The last second of the time-series was assumed to be most relevant for predicting ground motion in other nodes, which have not yet received the signal. Therefore, the last 100 tokens of the raw time-series were flattened and appended to the output of the CNN. The input size of the first GCN layer was increased accordingly (c.f. Sec. A.5).

Fig. 4.8 shows the experimental results from Tab. 4.4 visualized as errorbars. The reference model generally achieved better results than the model trained with appended raw time-series tokens. There appears to be no advantage to also include raw time-series tokens in the GCN.

Table 4.4.: MSE loss (Eq. 3.2) on the test dataset of the reference model and a model trained using raw time-series tokens (last 1s) concatenated to the CNN output. $\mathcal{L}_{\text{node}}$ is the average across all target variables, which is the metric used for model selection. M is the metric, U and L are the upper and lower bounds of the 95% confidence interval. The best result for each time-series padding is highlighted in bold.

Static Features		$\mathcal{L}_{\text{node}}$ **for different Time-Series Paddings** 0s	1s	2s	3s	4s	5s	$\mu_{\mathcal{L}_{\text{node}}}$
Reference Model	U	0.5397	0.5367	0.5499	0.5686	0.5691	0.5825	**0.5437**
	M	**0.5255**	**0.5227**	**0.5361**	0.5539	**0.5555**	**0.5688**	
	L	0.5110	0.5087	0.5220	0.5390	0.5413	0.5548	
With Raw Tokens	U	0.5591	0.5554	0.5503	0.5643	0.5707	0.5849	0.5502
	M	0.5446	0.5408	0.5372	**0.5502**	0.5570	0.5711	
	L	0.5302	0.5260	0.5230	0.5352	0.5419	0.5561	

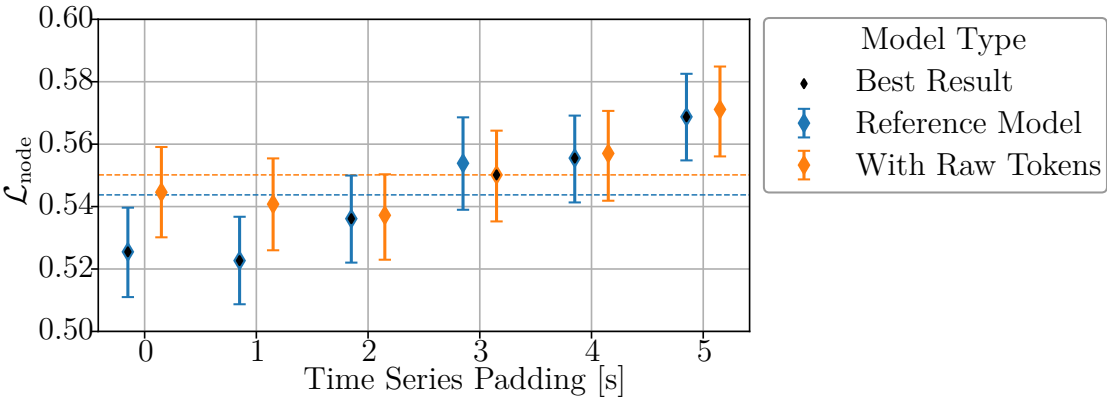


Figure 4.8.: Visualization of Tab. 4.3. The markers show $\mathcal{L}_{\text{node}}$ of the test dataset for different time series paddings. The caps mark the corresponding 95% confidence interval.

4.6. Pretraining and Station Dropout

The following section investigates the effect of station dropout (Sec. 3.6) and pretraining (Sec. 3.5). Three experiments were carried out:

1. Experiment: Direct Finetuning
   - 200 epochs: finetune randomly initialized model (c.f. Sec. A.2).
2. Experiment: Pretraining + Finetuning
   - 100 epochs: pretrain randomly initialized model on the pretext task (c.f. Sec. A.6).
   - 200 epochs: switch the last linear layer back to the regular one (predicting 5 IMs), and finetune the randomly initialized head and pretrained base (c.f. Sec. A.2).

3. Experiment: Pretraining + Frozen Base + Finetuning
    - 100 epochs: pretrain randomly initialized model on the pretext task (c.f. Sec. A.6).
    - 10 epochs: switch the last linear layer back to the regular one (predicting 5 IMs), freeze the pretrained base, and finetune the head only (c.f. Sec. A.2).
    - 200 epochs: unfreeze the base and finetune the whole model.

Each experiment was repeated six times using different station dropout ratios $\{0.0, 0.1, 0.2, 0.3, 0.4, 0.5\}$, resulting in six models per experiment. For model selection, the models were evaluated on the validation dataset using a station dropout ratio of 0.0. The regular MSE loss from Eq. 3.2 was used for optimization both during pretraining and finetuning.

Table 4.5.: MSE loss (Eq. 3.2) on the test dataset for models trained using different station dropout ratios without pretraining. The values are averaged across the results from the test dataset evaluated for different station dropout ratios. This table corresponds to the results visualized in Fig. 4.9c) and d). The best result for each time-series padding is highlighted in bold.

Station Dropout during Training	$\mathcal{L}_{\text{node}}$ **for different Time-Series Paddings** 0s	1s	2s	3s	4s	5s	$\mu_{\mathcal{L}_{\text{node}}}$
0%	**0.5375**	0.5441	0.5443	0.5493	0.5653	0.5875	0.5547
10%	0.5580	0.5512	0.5530	0.5586	0.5665	0.5837	0.5618
20%	0.5399	**0.5322**	0.5337	0.5425	0.5523	0.5691	**0.5450**
30%	0.5545	0.5485	0.5426	0.5417	**0.5458**	**0.5652**	0.5497
40%	0.5464	0.5360	**0.5297**	**0.5388**	0.5533	0.5798	0.5473
50%	0.5576	0.5603	0.5559	0.5608	0.5743	0.5840	0.5655

Table 4.6.: MSE loss (Eq. 3.2) on the test dataset for models trained using different station dropout ratios without pretraining. The values are averaged across the results from the test dataset evaluated for different time-series paddings. This table corresponds to the results visualized in Fig. 4.9b1-b6) and d). The best result for each time-series padding is highlighted in bold.

Station Dropout during Training	**Station Dropout during Evaluation** 0%	10%	20%	30%	40%	50%	$\mu_{\mathcal{L}_{\text{node}}}$
0%	0.5405	0.5472	0.5492	0.5568	0.5634	0.5712	0.5547
10%	0.5462	0.5527	0.5565	0.5637	0.5727	0.5792	0.5618
20%	**0.5386**	**0.5428**	**0.5416**	**0.5433**	**0.5474**	**0.5561**	**0.5450**
30%	0.5430	0.5459	0.5434	0.5475	0.5552	0.5634	0.5497
40%	0.5406	0.5466	0.5421	0.5455	0.5500	0.5591	0.5473
50%	0.5574	0.5601	0.5604	0.5668	0.5715	0.5768	0.5655

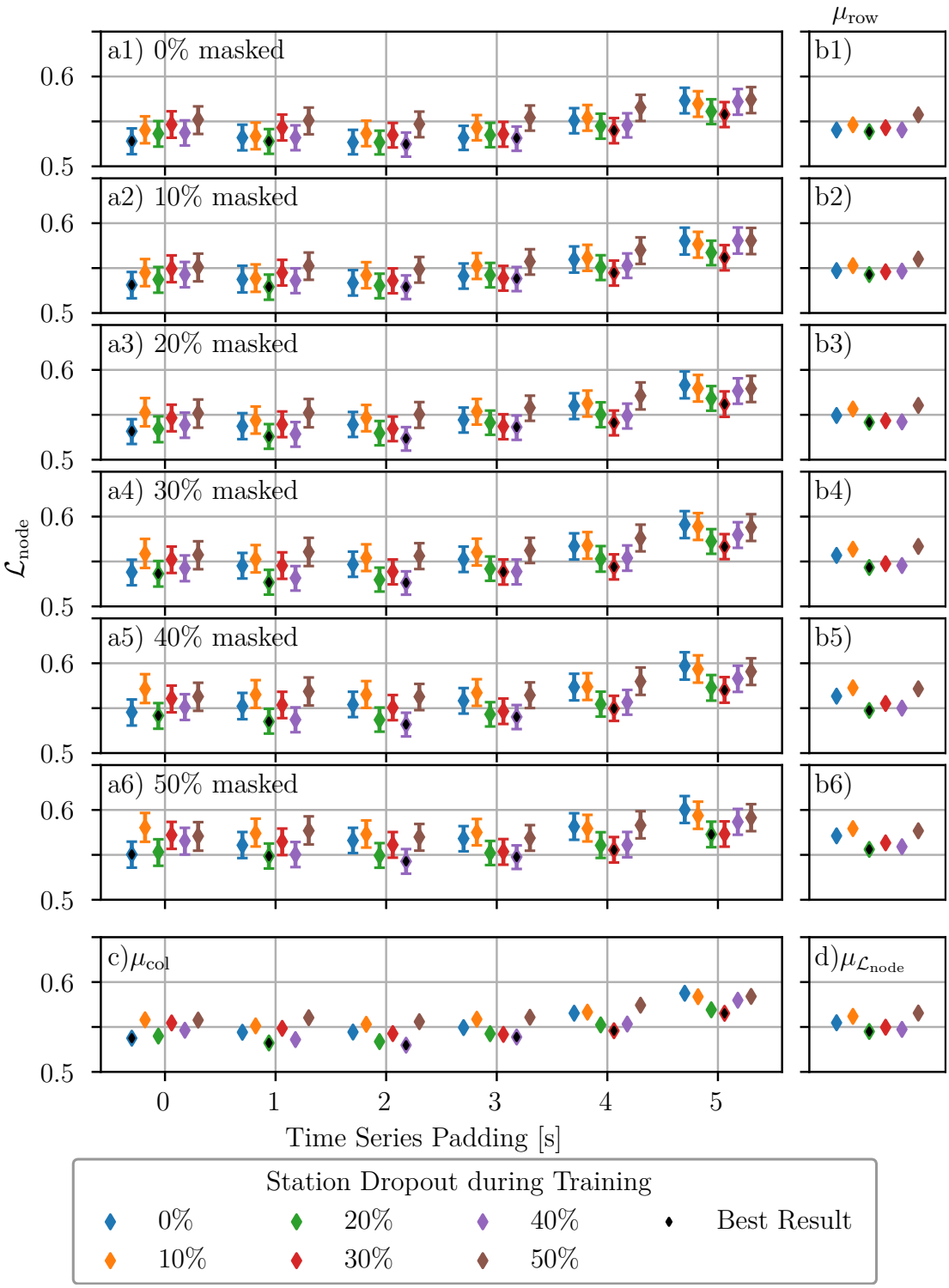


Figure 4.9.: Results of the six models, trained on different station dropout ratios, from experiment 1 (direct finetuning). Each row in a1-6) corresponds to a different station dropout ratio during evaluation (not training). The markers show $\mathcal{L}_{\text{node}}$ of the test dataset for different time series paddings. The caps mark the corresponding 95% confidence interval. b1-6) show the mean results averaged over different time series paddings (c.f. Tab. 4.6). c) shows the mean results averaged over different station dropout ratios during evaluation (c.f. Tab. 4.5). d) shows the overall results, averaged over both time series paddings and station dropout ratios during evaluation.

Fig. 4.9 shows the results from experiment 1 (direct finetuning) for six models trained using different station dropout ratios. Fig. 4.9 a-6) show the test loss, for different time series paddings and each row correponds to evaluating the models with a different station dropout ratio. Fig. 4.9 b-6) show the mean results, averaged across the time series paddings, visualizing the results from Tab. 4.6. Fig. 4.9 c) shows the mean results, averaged across the station dropout ratios during evaluation, visualizing the results from Tab. 4.5.

Experiments 2 and 3 did not improve the overall results of experiment 1, indicating that the pretraining - at least in the form proposed here - may not be helpful for this setting. Therefore, only the results from experiment 1 are shown here, focusing on the effect of station dropout. For the results from experiments 2 and 3, see Sec. A.2.1.

Fig. 4.9d) shows that the overall performance was best for models trained with a station dropout ratio of $0.2 - 0.4$. The model trained without station dropout performed only best in some cases when the time series padding was 0s. A low station dropout ratio of 0.1 performed worse. Similarly, a high station dropout ratio of 0.5 performed even worse, possibly because too much of the learning signal is omitted.

Averaging over the time series paddings, Fig. 4.9b1-6) show that the model trained with a station dropout ratio of 0.2 performed best for all six station dropout ratios during evaluation. The difference across the models is generally small for small station dropout ratios but becomes larger for higher values. This indicates that models trained with a medium station dropout ratio are generally better, especially, when there is a lot of missing time series data.

Furthermore, Fig. 4.9c) shows that the model trained without station dropout performs slightly better when the time series padding is 0s. For higher paddings, the models trained with medium station dropout ratios outperform the others. As the time series padding increases, the proportion of the input time series containing an actual earthquake signal decreases, increasing the difficulty of the task. In practice, it means that these models are quicker to make better predictions, when the earthquake signal just arrives at the first stations. For a heavily time-sensitive task such as EEW, this is an important aspect, because the warnings given to the public would be based on better estimations that are faster available.

The trends and patterns described here were consistent with the results from experiments 2 and 3, even though the pretraining did not improve the performance further.

4.6.1. Error for Masked Out Stations

In the previous section, masked and non-masked nodes were evaluated jointly to gain insights into the overall performance of the models under different circumstances. This section further investigates the effect of the masking on the performance. For this, bootstrapping was applied to the masked out nodes only, for the datasets with 10%-50% station dropout ratio. For comparison, bootstrapping was applied to the same nodes from the dataset with 0% station dropout, i.e. where none of the stations were masked. This allows a direct comparison of performance for the nodes with masked or non-masked input time series.

Note that the actual fraction of masked out nodes is always slightly below the value set as a hyperparameter, because the first node to receive the earthquake signal is omitted from the masking to ensure that there is an actual earthquake signal in the input time series (c.f. Sec. 3.6). The actual numbers and percentages of masked nodes for each dataset are given in Fig. 4.10a1-6).

Table 4.7.: MSE loss (Eq. 3.2) of the masked out nodes in the test dataset for models trained using different station dropout ratios without pretraining. The values are averaged across the results from the test dataset evaluated for different time-series paddings. This table corresponds to the results visualized in Fig. 4.10c) and d). The best result for each time-series padding is highlighted in bold.

Station Dropout during Training		$\mathcal{L}_{\mathbf{node}}$ **for different Time-Series Paddings** 0s	1s	2s	3s	4s	5s	$\mu_{\mathcal{L}_{\mathbf{node}}}$
0%	$\mathcal{L}_{\mathrm{m}}$	**0.5030**	0.5097	0.5160	0.5210	0.5361	0.5616	0.5246
	$\mathcal{L}_{\mathrm{m}} - \mathcal{L}_{\mathrm{nm}}$	0.0166	0.0207	0.0278	0.0277	0.0244	0.0235	0.0234
10%	$\mathcal{L}_{\mathrm{m}}$	0.5244	0.5174	0.5251	0.5319	0.5367	0.5578	0.5322
	$\mathcal{L}_{\mathrm{m}} - \mathcal{L}_{\mathrm{nm}}$	0.0223	0.0241	0.0249	0.0247	0.0226	0.0236	0.0237
20%	$\mathcal{L}_{\mathrm{m}}$	0.5119	0.5048	0.5051	0.5156	0.5240	0.5421	**0.5173**
	$\mathcal{L}_{\mathrm{m}} - \mathcal{L}_{\mathrm{nm}}$	0.0086	0.0109	0.0145	0.0142	0.0131	0.0127	0.0124
30%	$\mathcal{L}_{\mathrm{m}}$	0.5246	0.5172	0.5121	0.5147	**0.5207**	**0.5386**	0.5213
	$\mathcal{L}_{\mathrm{m}} - \mathcal{L}_{\mathrm{nm}}$	0.0135	0.0114	0.0141	0.0126	0.0128	0.0128	0.0129
40%	$\mathcal{L}_{\mathrm{m}}$	0.5144	**0.5045**	**0.4999**	**0.5134**	0.5309	0.5548	0.5197
	$\mathcal{L}_{\mathrm{m}} - \mathcal{L}_{\mathrm{nm}}$	0.0127	0.0069	0.0086	0.0126	0.0134	0.0132	0.0112
50%	$\mathcal{L}_{\mathrm{m}}$	0.5182	0.5219	0.5223	0.5297	0.5437	0.5566	0.5321
	$\mathcal{L}_{\mathrm{m}} - \mathcal{L}_{\mathrm{nm}}$	0.0099	0.0164	0.0158	0.0130	0.0154	0.0164	0.0145

Tab. 4.7 shows the loss for the masked out stations $\mathcal{L}_{\mathrm{m}}$ for models trained with different station dropout ratios, for different paddings of the time series. The rows labeled $\mathcal{L}_{\mathrm{m}} - \mathcal{L}_{\mathrm{nm}}$ show the difference in loss between the masked and non-masked input time series for the same nodes, i.e. how much does the performance decrease on average when the input time series is masked. Tab. 4.8 shows the

same but for different station dropout ratios during evaluation instead of different time series paddings.

Table 4.8.: MSE loss (Eq. 3.2) of the masked out nodes in the test dataset for models trained using different station dropout ratios without pretraining. The values are averaged across the results from the test dataset evaluated for different time-series paddings. This table corresponds to the results visualized in Fig. 4.10b1-b6) and d). The best result for each time-series padding is highlighted in bold.

Station Dropout during Training		**Station Dropout during Evaluation** 10%	20%	30%	40%	50%	$\mu_{\mathcal{L}_{\text{node}}}$
0%	$\mathcal{L}_{\text{m}}$	**0.5103**	0.5165	0.5193	0.5337	0.5430	0.5246
	$\mathcal{L}_{\text{m}} - \mathcal{L}_{\text{nm}}$	0.0151	0.0168	0.0228	0.0283	0.0340	0.0234
10%	$\mathcal{L}_{\text{m}}$	0.5154	0.5251	0.5279	0.5429	0.5500	0.5322
	$\mathcal{L}_{\text{m}} - \mathcal{L}_{\text{nm}}$	0.0119	0.0177	0.0241	0.0301	0.0345	0.0237
20%	$\mathcal{L}_{\text{m}}$	0.5129	**0.5119**	**0.5093**	**0.5214**	**0.5309**	**0.5173**
	$\mathcal{L}_{\text{m}} - \mathcal{L}_{\text{nm}}$	0.0098	0.0103	0.0094	0.0132	0.0192	0.0124
30%	$\mathcal{L}_{\text{m}}$	0.5165	0.5137	0.5121	0.5280	0.5363	0.5213
	$\mathcal{L}_{\text{m}} - \mathcal{L}_{\text{nm}}$	0.0096	0.0064	0.0097	0.0163	0.0224	0.0129
40%	$\mathcal{L}_{\text{m}}$	0.5212	0.5123	0.5105	0.5218	0.5325	0.5197
	$\mathcal{L}_{\text{m}} - \mathcal{L}_{\text{nm}}$	0.0098	0.0072	0.0075	0.0116	0.0201	0.0112
50%	$\mathcal{L}_{\text{m}}$	0.5196	0.5255	0.5281	0.5406	0.5465	0.5321
	$\mathcal{L}_{\text{m}} - \mathcal{L}_{\text{nm}}$	0.0088	0.0094	0.0157	0.0177	0.0210	0.0145

Fig. 4.10 visualizes the results similar to Fig. 4.9 but for the masked out nodes of each respective dataset only. The colored markers represent the results when the input is masked, whereas the grey markers in the background are the loss when the time series are not masked out.

As could be expected, the performance was always worse when the input was masked out. However, the differences remain comparatively small. Even though the loss increases as the time series padding increases, the difference in loss between masked and non-masked inputs does not appear to change significantly. In contrast, the difference increases as the station dropout ratio increases, which could be expected, as the number of surrounding non-masked nodes, which the models have to rely on for the masked nodes, decreases. This effect is more pronounced for the models trained with 0% and 10% station dropout ratio. Models trained with a higher station dropout ratio appear to be more stable in performance, as more inputs are masked out. This suggests that the models, especially when trained with station dropout, are generally well capable of inferring necessary information from neighbouring nodes, in case an input is missing.

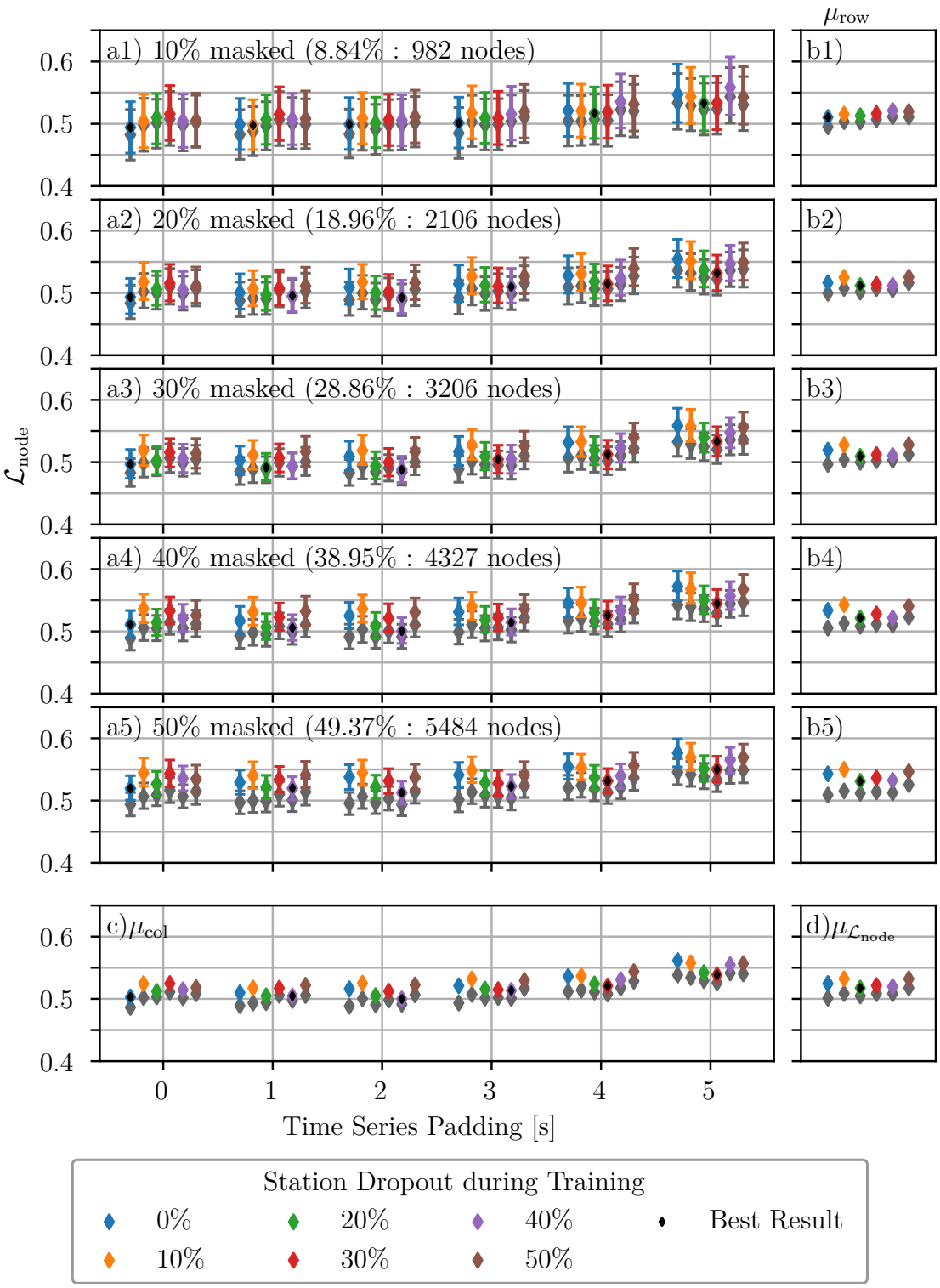


Figure 4.10.: Similar to Fig. 4.9, but the results are filtered to only contain stations (nodes) for which the input was masked out. The grey markers are the results from the dataset with a 0% station dropout ratio (i.e. no masking) for the same stations, that are masked in the other datasets. It is thus a direct comparison of how the performance differs if the input time series is masked or not. Higher station dropout ratios correspond to more masked stations and thus more datapoints.

4.6.2. Earthquake Maps

Fig. 4.11 shows six different earthquake samples from the test dataset. The graph of each sample is plotted with red dots as nodes and thin black lines as edges. The earthquake epicenter is shown as a black star. Coast lines are shown as solid black lines; country borders as dotted lines; water bodies as dotted areas.

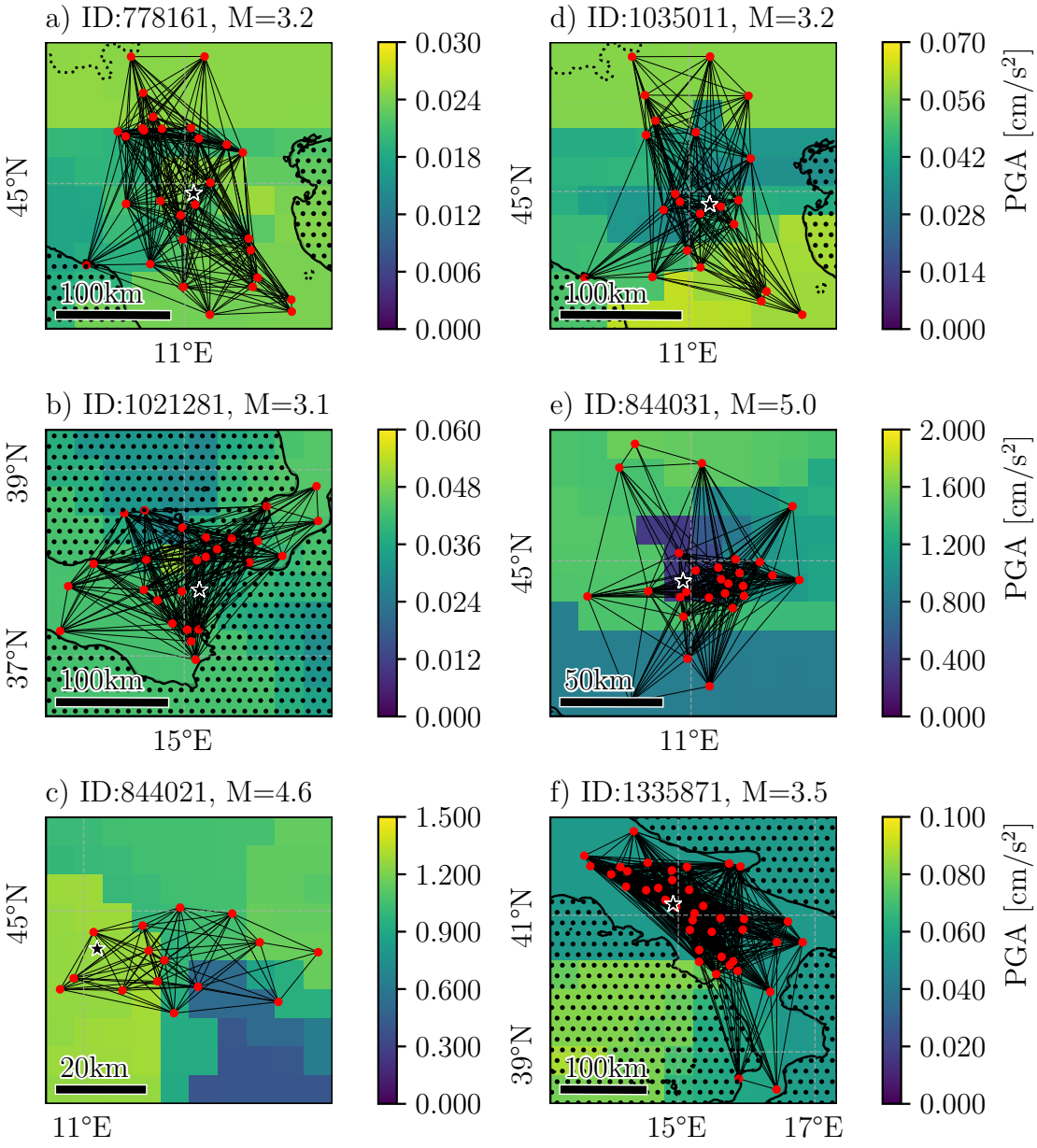


Figure 4.11.: Earthquake Maps (c.f. Sec. 3.7) for different earthquake events from the test dataset, where the predicted PGA is shown as a 10×10 heatmap. The model used to generate the predictions was trained using a station dropout ratio of 20%. Dotted areas are water bodies; thick straight lines are coasts, dotted lines are country borders. The graph of the sample is shown with nodes in red and edges as black lines. The earthquake epicenter is shown as a black star. Map projection: Plate Carrée.

The method described in Sec. 3.7 was used to generate earthquake maps for the samples in Fig. 4.11. Specifically, 100 artificial nodes were added in a 10×10 grid. The model from the previous experiment, trained using a station dropout ratio of 20%, was used to generate the predictions. For each artificial node, all five IMs were predicted and the PGA was then plotted as a heatmap. None of the input time series were masked, except for the artifical node, and the time series padding was set to 0s.

Since there are no ground truth values for the locations of the artificial nodes, the accuracy of the predicions cannot be calculated. If a real node is removed from the graph, and then artificially added back into the graph, it is essentially the same as simply masking the input time series for this node. The change in performance for masked and non-masked input time series was investigated in the previous section. For the maps generated here, stations were also inserted in places where no real station could exist, such as in the Mediterranean Sea. Depending on the structure of the graph, the artificial nodes might also be far outliers in terms of the geographical position. Therefore, this scenario is not fully comparable to the masking investigated in the previous section. Nevertheless, the previous section may give limited insight into the reliability of the predictions generated here.

In some instances, the predicted patterns appear reasonable as the highest predicted intensity was close to the epicenter of the earthquake (Fig. 4.11a-c). In other cases, the epicenter was located in the area of lowest predicted intensity (Fig. 4.11d & e). The maps further illustrate the difficulty of the task and dataset: the graphs do not only differ vastly in structure and number of nodes but also in terms of geographical scale (c.f. Fig. 4.11c & f).

As there is no way of assessing the actual quality of these maps, the results presented here should be seen as a proof-of-concept for the generation of earthquake intensity maps using NNs and dynamic graphs.

4.7. Probabilistic Models, Mixture Models and Deep Ensembles

This section illustrates multiple experiments regarding probabilistic models, mixture models, and deep ensembles built from individual probabilistic models.

The standard model architecture was modified to predict the parameters of a Gaussian distribution, i.e. mean μ_θ and variance σ_θ, for each node and target, as explained in Sec. 3.3.2. To achieve this, a second linear output layer was added after the final GCN layer (c.f. Sec. A.7). The models were then optimized using

the NLL loss from Eq. 3.5; model selection was done using the regular MSE loss from Eq. 3.2 on the validation dataset.

The experiments were conducted as follows:

1. **Probabilistic Models**
   The described probabilistic model (c.f. Sec. A.7) was trained five times with random weight initialization, yielding five slightly different models built on the same architecture. The results for all five models can be found in Tab. A.1 and Fig. A.2.

2. **Mixture Models**
   The mixture models use the individually trained probabilistic models from the first experiment and combine their outputs to parameters of a single Gaussian distribution using a specific weighting scheme (c.f. Sec. A.8), without retraining them.

   - *Uniform Weighting Scheme*
     The outputs by each model were weighted uniformly according to Eq. 3.7-3.8, similar to [Lakshminarayanan et al., 2017].
   - *Variance Weighting Scheme*
     The variances predicted per node and target were used to calculate a weighting scheme across the models in the mixture, where high variances correspond to low weights. This weight was then included in the calculation of the mixture mean and variance as in Eq. 3.9-3.11.

3. **Deep Ensembles**
   The deep ensembles here are ensembles of the probabilistic models from the first experiment, but in contrast to the mixture models, the individual models were trained jointly in the ensemble (c.f. Sec. A.9). The main loss component was the same NLL loss, as for the individual models (c.f. Eq. 3.5), but applied to the combined ensemble output using the variance weighting scheme from Eq. 3.9-3.11. During each evaluation on the validation dataset during training, the mean and standard deviation of the model weights were calculated according to Eq. 3.12-3.13 to monitor model collapse and specialization. To encourage diversity and prevent collapse of individual models in the ensemble, a regularization loss was added:

   - *Mean Weight Regularization*
     Here, the mean weight regularization loss from Eq. 3.16 was used, which penalizes the ensemble when models do not equally contribute to the ensemble output (i.e. have different average weights). For optimization, Eq. 3.17 was used. The ensemble was trained seven times

using different weightings factors $\lambda_{\text{mw}} \in \{0, 1, 10, 10^2, 10^3, 10^4, 10^5\}$ for the regularization loss (c.f. Sec. A.12). The results for all seven training runs can be found Tab. A.2 and Fig. A.3.

- *HSIC and Participation Loss*
  In the second approach, the regularization consisted of two loss terms. The HSIC (Eq. 3.25) was used to encourage diversity in output distributions across the models in the ensemble (c.f. Sec. A.13-A.15). The participation loss (Eq. 3.26) was used to penalize the ensemble if at least one of the models was contributing less than 5% to the ensemble output (c.f. Sec. A.16). For optimization, Eq. 3.27 was used. Again, the ensemble was trained seven times, where both weighting factors λ_{HSIC} and λ_{part} were set to the same value $\lambda_{\text{HSIC}} = \lambda_{\text{part}} \in \{0, 1, 10, 10^2, 10^3, 10^4, 10^5\}$. The results for all seven training runs can be found Tab. A.3 and Fig. A.4.

4.7.1. Experimental Results

Tab. 4.9 and Fig. 4.12 show the results from the previously described experiments. Specifically, it shows the best of the five probabilistic models, the uniformly-weighted mixture model and variance-weighted mixture model, the best deep ensemble model trained using mean weight regularization and the best deep ensemble model trained with HSIC and participation loss. The results for all probabilistic models, as well as all configurations of the deep ensembles, are found in Sec. A.2.2.

The best probabilistic model generally performed worse than the mixture models and deep ensembles, usually by a large margin. The exception is for a time series padding of 5s, where it performed best; looking at the other probabilistic models in Fig. A.2 shows that this appears to be an outlier in terms of performance.

This also shows in the two mixture models, consisting of all five individually trained probabilistic models, which performed worse. Both mixture models were generally much better than the individual probabilistic models they consist of. The difference between the uniformly-weighted and the variance-weighted mixture models was negligible. This may be because the individual models in the mixture predicted rather similar variances across nodes and targets, yielding similar weights. This is the drawback of mixture models in contrast to deep ensembles trained end-to-end, since the individual models in the mixture models are encouraged to be generalists and cannot specialize.

The best deep ensemble model trained with mean weight regularization used a weighting factor of $\lambda_{\text{mw}} = 1$. It generally performed worse than the mixture models. This may be due to the fact, that the models in the ensemble are free

to, but not forced to specialize. At the same time, this freedom may allow partial collapse of the models in the ensemble, if the regularization loss is not effective enough at disencouraging this. This is further investigated in the next section.

The best deep ensemble model trained with HSIC and participation loss used a weighting factor of $\lambda_{\text{HSIC}} = \lambda_{\text{part}} = 10$. It generally outperformed all previously mentioned approaches, except for time series paddings of 2s and 5s, where it was slightly outperformed. It achieved the overall lowest loss within this set of experiments.

Table 4.9.: MSE loss (Eq. 3.2) on the test dataset for [**P**] the best probabilistic model, [**M1**] the uniformly-weighted mixture model, [**M2**] the variance-weighted mixture model, [**E1**] the best deep ensemble trained with mean weight regularization, and [**E2**] the best deep ensemble trained with HSIC and participation loss. $\mathcal{L}_{\text{node}}$ is the average across all target variables, which is the metric used for model selection. M is the metric, U and L are the upper and lower bounds of the 95% confidence interval. The best result for each time-series padding is highlighted in bold.

Model		$\mathcal{L}_{\text{node}}$ **for different Time-Series Paddings**						
		0s	1s	2s	3s	4s	5s	$\mu_{\mathcal{L}_{\text{node}}}$
P	U	0.5509	0.5516	0.5460	0.5632	0.5752	0.5762	0.5465
	M	0.5360	0.5369	0.5327	0.5488	0.5612	**0.5631**	
	L	0.5211	0.5223	0.5192	0.5344	0.5465	0.5489	
M1	U	0.5443	0.5385	0.5333	0.5457	0.5606	0.5824	0.5374
	M	0.5301	0.5247	**0.5204**	0.5328	0.5477	0.5688	
	L	0.5160	0.5110	0.5072	0.5193	0.5337	0.5546	
M2	U	0.5444	0.5391	0.5334	0.5456	0.5607	0.5828	0.5375
	M	0.5302	0.5251	**0.5204**	0.5326	0.5477	0.5692	
	L	0.5161	0.5114	0.5071	0.5192	0.5337	0.5550	
E1	U	0.5523	0.5400	0.5386	0.5493	0.5622	0.5794	0.5401
	M	0.5378	0.5258	0.5257	0.5359	0.5494	0.5661	
	L	0.5233	0.5122	0.5122	0.5220	0.5355	0.5517	
E2	U	0.5432	0.5340	0.5368	0.5429	0.5569	0.5798	**0.5355**
	M	**0.5295**	**0.5202**	0.5235	**0.5299**	**0.5439**	0.5661	
	L	0.5156	0.5066	0.5101	0.5167	0.5302	0.5519	

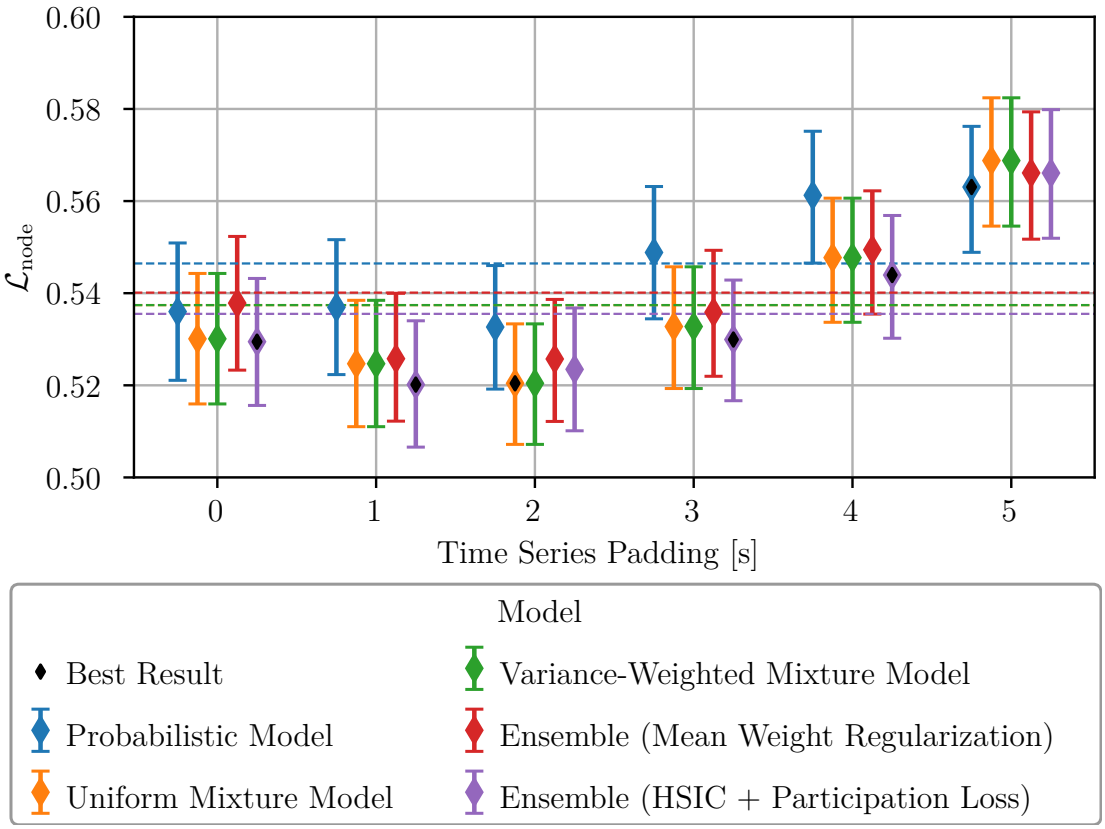


Figure 4.12.: Visualization of Tab. 4.9. The markers show $\mathcal{L}_{\text{node}}$ of the test dataset for different time series paddings. The caps mark the corresponding 95% confidence interval.

4.7.2. Specialization of Deep Ensembles

Fig. 4.13 shows the mean $\mu_{w_{\theta_m},t}$ and standard deviation $\sigma_{w_{\theta_m},t}$ of the weights of each model in the ensemble, trained with mean weight regularization ($\lambda_{\text{mw}} = 1$), across the nodes of the validation dataset during training as calculated with Eq. 3.12-3.13 for the different target variables t. The lines represent the mean $\mu_{w_{\theta_m},t}$; the shaded areas represent $\mu_{w_{\theta_m},t} \pm \sigma_{w_{\theta_m},t}$. Each line corresponds to one model in the ensemble. Fig. 4.13e shows the average across all target variables.

Initially, all models contribute relatively equally to the output of the ensemble. However, the contribution of models 3 and 5 quickly degrades as their mean weight approaches 0, while the contribution of the remaining 3 models increases. Finally, models 3 and 5 collapse since their contribution becomes negligible as their predicted variances are becoming too high. As a consequence, their contribution to the ensemble loss becomes negligible as well and they do not receive significant updates of their model parameters anymore. In the end, 40% of the ensemble ressources are practically wasted. This shows that the regularization loss in its form in Eq. 3.16 appears to be too weak of a constraint to reliably force the ensemble to make use of each individual model.

Apart from this issue, the remaining 3 models appear to specialize during the training. On one hand, their contribution to the ensemble output varies across the different target variables; e.g. model 2 contributes most to PGA, PGV and $SA_{0.3}$, whereas model 1 contributes most to $SA_{1.0}$ and $SA_{3.0}$. On the other hand, the standard deviation of the weights across the nodes increases significantly during the training. In the beginning, each model contributes rather equally to the output for all nodes. Later on, the contribution varies heavily across the nodes. This could indicate that the models are indeed specializing on certain types of nodes or samples.

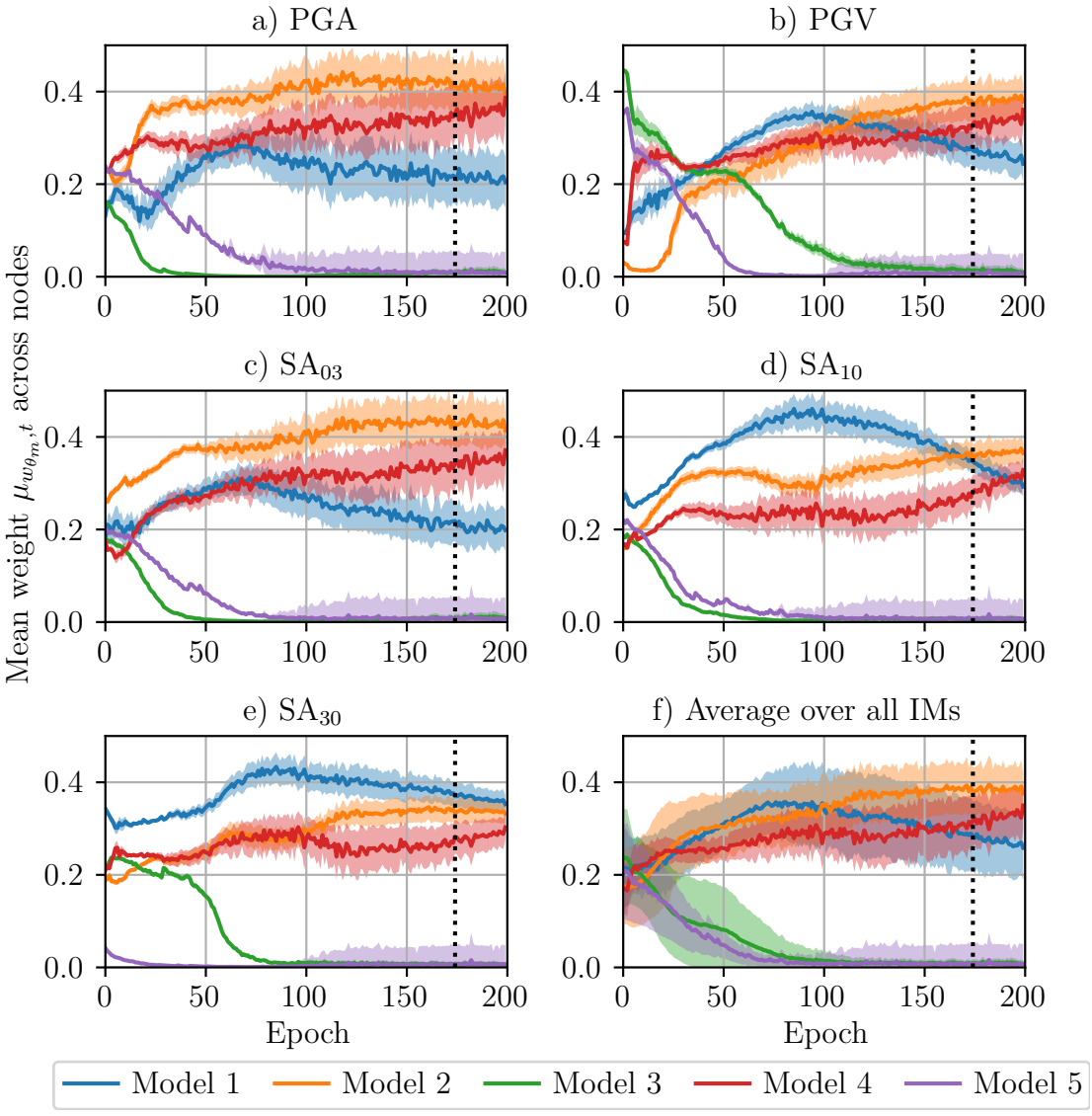


Figure 4.13.: Mean weight $\mu_{w_{\theta_m},t}$ across all nodes in the validation set for each target and model in the ensemble, trained with mean weight regularization, during the training. The shaded areas represent one standard deviation above and below the mean. The dotted line marks the best performing epoch.

Fig. 4.14 shows the equivalent of Fig. 4.13, but for the ensemble trained using HSIC and participation loss ($\lambda_{\text{HSIC}} = \lambda_{\text{part}} = 10$) as regularization. The horizon-

tal dashed line in Fig. 4.14f) marks the participation threshold $\gamma = 0.05$ from the participation loss in Eq. 3.26. If the mean weight of any model in the ensemble falls below this threshold, the regularization counteracts this.

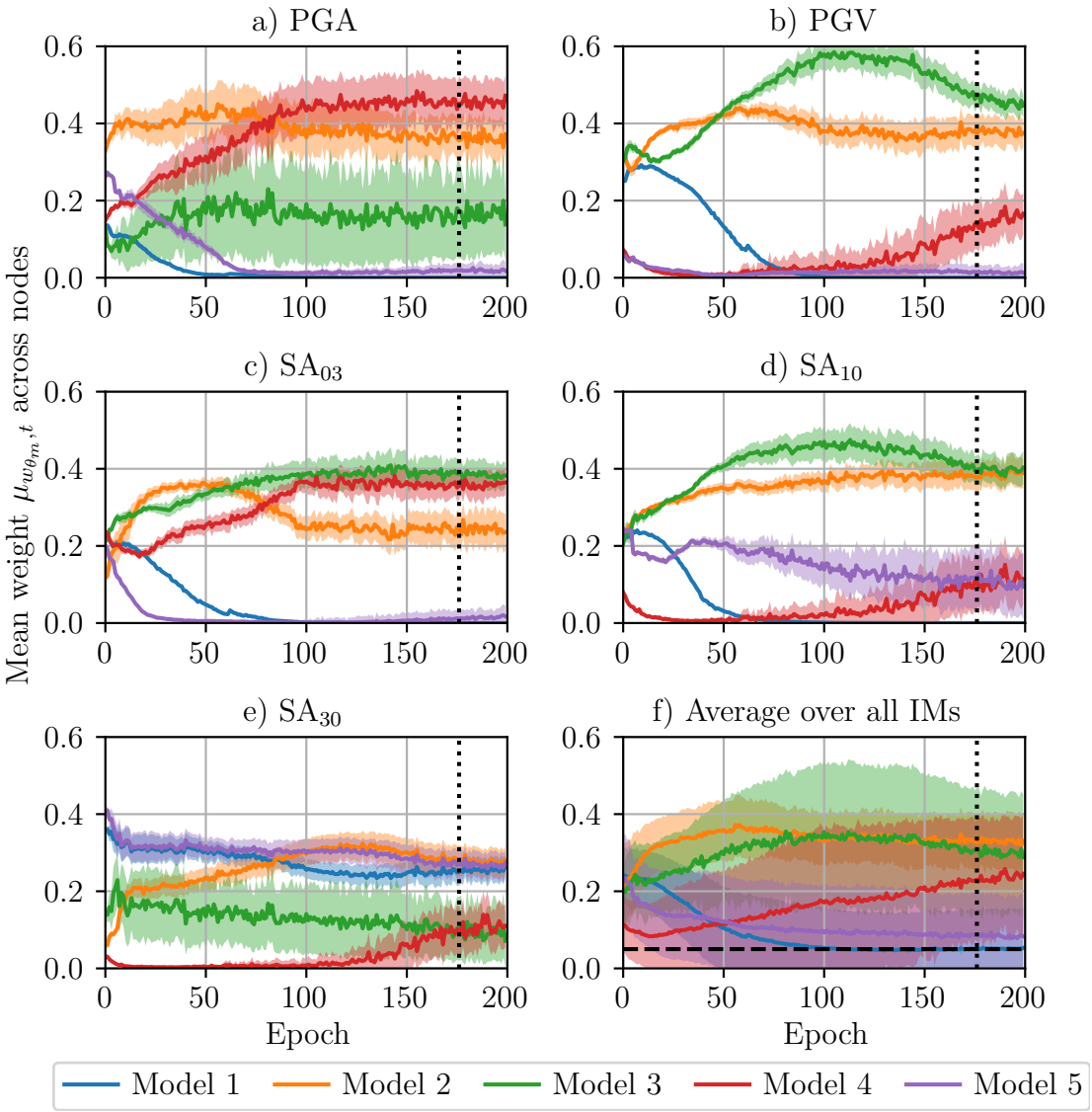


Figure 4.14.: Mean weight $\mu_{w_{\theta_m},t}$ across all nodes in the validation set for each target and model in the ensemble, trained with HSIC and participation loss ($\lambda_{\text{HSIC}} = \lambda_{\text{part}} = 10$) during the training. The shaded areas represent one standard deviation above and below the mean. The dotted line marks the best performing epoch. The horizontal dashed line in f) marks the threshold for the participation loss.

As can be seen, the participation loss was 0 during the first half of the training and only became active as model 1 approached the participation threshold. In the second half of the training, model 1 stayed on the threshold, showing that the participation loss appears to be efficient in preventing the collapse of models in the ensemble. At the same time, it does not force equal contributions of the

models, unlike the mean weight regularization, allowing the models to specialize in differently sized structured subsets of the data.

Models 1 and 5 neglected most of the targets, but focused on the SA_{30} and SA_{10} instead. The other models also had different contributions across the targets showing that there was some specialization in terms of target variables.

Although differing across targets, the standard deviation of the weights, as indicated by the shaded areas, appeared to be much higher on average compared to the approach with mean weight regularization. This may indicate a specialization in terms of certain subsets of nodes, which could likely be the effect of the HSIC.

Another thing worth noting is, that the models (specifically model 4) were able to start contributing again for some targets, where their contribution had already been negligible (PGV, SA_{10}, and SA_{30}). This did not occur in the other experiment with mean weight regularization.

5. Discussion

5.1. Recap of Major Findings

This section gives a brief summary of major findings from the previous chapter:

1. The models appear to be highly sensitive to the learning rate, as they only converged for a very small lr of 0.0001 but experienced mode collapse for lrs 0.001 or higher; i.e. identical outputs across all nodes and samples.

2. Static features and how they are processed appear to have a strong effect on performance. Introducing an MLP for static features achieved the biggest improvement to the standard model across all experiments. This suggests that even simple architectural changes can have a substantial effect on performance.

3. Pretraining, using the prediction of the PGA for a certain time window as a pretext task, did not improve performance.

4. Training with station dropout made the models better at handling missing input time series and improved their predictions given shorter signals.

5. Creating earthquake intesity maps by inserting artificial nodes into the graph is possible, but fully assessing their reliability is difficult due to missing ground truth data.

6. Mixture models and deep ensembles outperform individual models. HSIC and participation loss appear to lead to specialization of individual models, while preventing collapse.

7. Using dynamic and highly heterogeneous graphs with sparse inputs represents a realistic but especially challenging setting for EEW.

8. There is a trade-off between a models performance on a narrow dataset and its ability to generalize.

5.2. Architectural and Optimization Challenges

5.2.1. Learning Rate and Optimization Dynamics

Sec. 4.1 showed that the training success was highly dependent on the used lr. For medium and high lrs, the models produced degenerate ouputs and experienced a mode collapse by mapping all inputs to the same output. Mode collapse occurs when the model is only able to fit a minority of modes in the data distribution while ignoring the rest.

This phenomenon is well known in other areas of DL, particularly in image generation using Generative Adversarial Networks (GANs), where the optimization landscape is especially complex. There have been numerous attempts to solve this issue [Durall et al., 2020] through more sophisticated optimization techniques. Whether such strategies are applicable in this context remains an open question and could be worth investigating.

In this work, the Adam optimizer was used, which adapts the parameter updates based on statistical properties of the previous gradients. Specifically, Adam computes an exponential moving average of the gradients and their squared values (to estimate the variance) for each individual parameter and uses them to rescale the raw gradients before multiplying the lr and updating the parameters. This way the step size is adapted for each parameter individually: in regions where the gradients fluctuate strongly (high variance), the updates are scaled down; in smoother regions (low variance), the step size remains larger. However, the overall scale of the weight updates is still controlled by the base lr η, which is a hyperparameter provided by the user.

An lr scheduler modifies this base lr over time, independently of Adam's internal gradient scaling. Typically, the lr is decreased over time during training to encourage convergence by making weight updates more conservative as the model approaches a local minimum. Popular strategies include cosine annealing [Loshchilov and Hutter, 2017], where the lr gradually decays, or ReduceLROnPlateau [Wu and Liu, 2023], which reduces the lr if the training or validation loss has not improved for a specified number of epochs. Such schedulers, as well as other optimizer variants, could potentially improve the training stability and convergence in this work.

5.2.2. Architectural Exploration

Sec. 4.4 showed that static features (station position and altitude) had a significant impact on the model performance. Replacing simple concatenation of the raw static features with an MLP that preprocesses them before concatenation

led to a substantial reduction in test loss. The model trained with the MLP for static features yielded the overall best result across all experiments.

This work is based on the architecture by [Bloemheuvel et al., 2023] and focuses on investigating different training methods, with only minimal changes to the original architecture. However, the results from Sec. 4.4 suggest that further architectural exploration could significantly improve model performance.

Possible directions could be the use of alternative NN layers, structural modifications to existing building blocks, or both. The models in this project used two simple 1D CNN layers as encoder for the time series. Alternatives to these include:

1. **Dilated CNNs**
   Dilated convolutions are frequently used in time series modelling, e.g. for autoregressive models like Temporal Convolutional Networks (TCNs) [Lea et al., 2016]. The convolutional filters in a CNN have a limited receptive field and are thus good at capturing local patterns but worse on a global scale. The dilation allows the CNN to capture patterns on a broader scale without increasing the kernel size or number of layers, because the receptive field increases exponentially with each layer due to the dilation.

2. **LSTMs** or **Gated Recurrent Units (GRUs)**
   LSTMs and GRUs are variants of Recurrent Neural Networks (RNNs) that have been used for sequence modelling successfully for a long time [Chung et al., 2014]. However, they come with their own set of disadvantages, especially due to their sequential nature, which prevents parallelization and can lead to vanishing or exploding gradients [Bai et al., 2018].

3. **Transformers**
   Transformers [Vaswani et al., 2023] are sequence models based on self-attention. Besides their high expressiveness, their advantage is their global context as they can adhere to the entire time series at once. Their disadvantage is their higher computational cost, and that they typically require a lot of data or suitable pretraining. This makes them less promising for small datasets like the GI dataset.

4. **Structured State Space Models (S4s)**
   A new, but promising architecture for sequence modelling is S4, which appears to be faster than transformers, has a long memory and can handle irregularly sampled time series data [Gu et al., 2022]. However, they are more complex, still under active research, and - as of now - have no native implementation in Pytorch.

Similarly to the time series encoder, the layers of the GNN could be replaced by more expressive ones, e.g. from simple GCN layers to:

1. **Graph Attention Networks (GATs)**
   While the edge weights in a GCN are static and defined through a normalized adjacency matrix, the edge weights in a GAT are learned through an attention mechanism with multiple heads [Veličković et al., 2018]. The attention mechanism calculates a similarity score between each node and its neighbors based on the node features. Due to the sparseness of the input time series, it might be better to base this primarily on the static node features.

2. **Graph SAmple and aggreGatE (GraphSAGE)**
   GraphSAGE randomly draws a fixed numbers of neighboring nodes and uses a parametrized aggregator function (e.g. an MLP), which is more expressive than a GCN [Hamilton et al., 2018]. GraphSAGE is inductive and thus performs better with unseen nodes and graphs. Compared to GCNs the smoothing of information is reduced and there are more possibilities for customizing the layer. However, due to the random sampling, it is less deterministic and stable, more complex, and might miss informative neighboring nodes.

3. **Graph Isomorphism Networks (GINs)**
   GIN is a GNN architecture that was built to improve the structural expressiveness, i.e. the models ability to distinguish different graph structures based on their connectivity and node features [Xu et al., 2019]. GIN aggregates all neighboring nodes via a sum (as opposed to a mean) and applies an MLP. It was designed to be as powerful as the Weisfeiler-Lehmann test, which determines whether two graphs are isomorphic, i.e. structurally identical. Previous architectures may not have been able to do this, i.e. they may map two structurally different graphs to the same embedding. GIN was designed to always map non-isomorphic graphs to different embeddings. GIN is thus stronger in structural expressiveness, which may be an advantage in an application like EEW.

Instead of replacing NN layers by other architectures and variants, one could also make structural changes using the existing building blocks, which could potentially improve the models:

1. **Deeper Architectures and Residual Connections**
   In the models used here, there are only two layers in the CNN and the GCN, as well as a single linear layer in the model head for target prediction.

One could easily either increase the hidden dimensions of these layers, or increase the number of layers. A greater depth could increase the model capacity to learn hierarchical or more abstract features. At the same time, deeper models may lead to over-smoothing in the GCN, and both CNN and GCN components could suffer from vanishing gradients. This could be addressed through residual connections similar to ResNets [He et al., 2016], BN layers or dropout layers. Furthermore, the linear layer in the head could be replaced by a more expressive MLP.

2. **Multi-Scale Feature Aggregation**
   Another option would be to extract features at different scales, for example by applying CNN layers with different kernel sizes, strides, and dilations in parallel. Alternatively, one could make multiple parallel paths in the network, applying different numbers of layers, possibly with different hyperparameters, and then joining their results. Shallower branches could extract low-level features, while deeper branches could extract high-level features, similar to the idea of Inception models [Szegedy et al., 2015]

3. **Time-and-Space Architecture**
   The models used here are TTS models, meaning they first process the temporal component of the data (CNN), and only afterwards process the spatial component (GCN). Considering that most nodes in this task have time series containing only background noise, when the earthquake has not reached them yet, the CNN's potential might be mostly wasted. Instead, one could use a T&S model, where both are processed at the same time or in an alternating way. These have a higher computational complexity than TTS [Cini et al., 2023], but may increase performance. The benefit would be that the MP of the GNN allows information to spread across nodes, before applying (part of) the CNN layers, which would allow them to operate significantly more on real signals instead of just noise. Due to this, T&S models could potentially be much more desirable in a task like EEW where only a fraction of input time series carries an actual signal.

4. **Auxiliary Tasks**
   It may be helpful to train the model on auxiliary tasks by adding more heads to the output of the GNN, e.g. the prediction of the arrival time of the P wave on node-level. Even if the task is not primarily of interest, it could guide the model towards learning more useful features in the intermediate layers.

5.3. Graph Representations and Challenges

5.3.1. Node Feature Design

The input node features for the GNN consisted of flattened time-series representations by the CNN, concatenated with the static features, which were station position and altitude. The goal was to make the model as generalizable as possible, such that it could also be used for locations not contained in the training dataset.

Therefore, the absolute node coordinates were replaced by relative ones, calculated w.r.t. some randomly sampled reference point. This hides the real position of the network on the Earth. Normalizing the relative coordinates furthermore hides the scale of the network.

The altitude of the stations was similarly normalized. The normalized altitude of an individual node always depends on the altitudes of the other nodes in the sample. This prevents the model from consistently identifying a specific station across different samples based on its altitude. Thus, there is no way for the model to identify different geographical regions in the data.

Even though geological data is not explicitly provided, the model could still infer regional geological properties indirectly, e.g. by recognizing characteristic earthquake patterns in specific areas. This implicit learning of geological conditions is prevented by the relative coordinates and normalization.

5.3.2. Edge Construction

The graphs are dynamic across samples, varying significantly in the number of nodes, number of edges, and spatial extent. But after the graph of a sample is created it is static. The edge weights were calculated using the geodesic distances, which were normalized, transformed according to Eq. 3.1, and filtered using a threshold for the graph diameter. The normalization again hides the scale of the graph from the model.

This method closely follows the approach used by [Bloemheuvel et al., 2023], but there may be considerable room for improvements in terms of model performance. The geodesic distance might not be the (only) key criterion as to how long a seismic wave travels between two points and how intense it is in terms of IMs. Some ideas that could potentially improve performance are:

1. **Learned Edge Weights**
   Instead of static edge weights, they could be learned, e.g. via another GNN architecture like GATs, as explained in the previous section. For this, static features of the nodes could be used or possibly features of the original static edges, such as geodesic distances.

2. **Addition of Geological Priors**
   One could also add geological properties of nodes as a static node features and compute edge weights based on similarity of these properties. Alternatively, one could incorporate pairwise geological relationships, e.g. aggregated material properties along the theoretical wave path between stations.

3. **Physics-informed Graphs**
   Instead of focusing on geographical and geological data, one could also use physical models, e.g. to approximate travel-time based on a velocity model, or the theoretical energy decay of a seismic wave between two points.

4. **Hybrid or Multi-Graph Combinations**
   The aforementioned methods could also be combined, either in a single graph, or by processing multiple graphs, representing different kinds of relationships (distances, geological properties, feature similarity), in parallel. While none of these connectivity strategies may be sufficient alone, they could complement one another when integrated into a hybrid or multi-graph framework. The models would thus not be as reliant on a single method to work well for each sample.

5. **Edge Filtering Criteria**
   In this approach, edges were sorted by their weight, based on geodesic distance, and removed until it would increase the graph diameter above a defined threshold. More adaptive or data-dependent strategies for edge filtering might improve results, but require careful design to avoid breaking graph connectivity, and appear challenging due to the high heterogeneity of the data.

5.3.3. Specialization vs. Generalization Trade-Off

[Bloemheuvel et al., 2023] applied their model to two datasets (CI & CW) from Italy, both containing the same 39 stations in each sample. The 39 nodes were hardcoded into the model architecture, particularly the regression head, making the model incompatible with dynamic graphs. Both networks were relatively small in terms of node count. Because the graphs were static, the authors could experimentally select a weight threshold to filter out low-weighted edges and minimize loss. In the CI dataset especially, earthquakes and stations were concentrated in a relatively small geographic area, increasing the likelihood that multiple stations would register the signal within the first 10s. Additionally, absolute latitude and longitude coordinates were used as static node features. This setup results in a model that is highly specialized to both the fixed graph topology and the specific regional conditions of the datasets.

In contrast, the GI dataset used in this work contains dynamic graphs that vary substantially in the number of nodes, edges, and geographical scale. Samples span the entire Italian region, with some epicenters located far offshore in the Mediterranean Sea or even in neighboring countires such as Croatia. The model does not have access to absolute position or physical scale due to the use of relative coordinates and normalization of coordinates, geodesic distances, and altitudes. The only potential cue for the physical scale of the graph comes from the relative timing of P wave arrivals across nodes. However, in many samples, few or even just one of the nodes register a signal within the 10s window, making the signal-based scale inference highly unreliable. These constraints also make tasks such as the epicenter prediction particularly difficult, if not infeasible.

Both approaches are valid, but they lie at opposite ends of a spectrum: [Bloemheuvel et al., 2023]'s model is tightly specialized to a static graph and region, whereas the models used in this work are trained for broad generalizability. The choice between generalization and specialization represents a fundamental trade-off, with each offering distinct advantages and disadvantages.

A specialized model can learn much about the region it is trained on, potentially capturing local geological conditions or typical seismic behaviors. However, such a model risks severe overfitting and may perform poorly - or even fail completely - when applied to other regions or graphs which are out of its training distribution.

Conversely, a generalist model is far more flexible: it can be applied to unseen regions and varying graph structures. But this comes at a cost: the model cannot learn region-specific patterns or adapt to local geological characteristics, not even implicitly, since most of this information is intentionally obscured through normalization and coordinate transformations.

Ideally, one would want a model that retains the flexibility of a generalist while still being able to incorporate regional context where available. This could be partially achieved by integrating geological priors - such as crustal properties - into edge construction, or by providing normalization factors (e.g. for coordinates or distances) as explicit static features.

Another idea would be to train a generalist model with layers that are conditionally adapted based on specific inputs. For example BN layers which could adapt to certain properties of the local geology. Alternatively, adapter layers could be added, which allow a frozen generalist model to be fine-tuned to specific regions using limited data, similar to adaptation methods such as Low Rank Adaption (LoRA) [Hu et al., 2021] in NLP.

Broadly speaking, reconciling generalization with specialization may require more sophisticated approaches that combine flexibility with region-specific adaptation.

5.4. Pretraining and Representation Learning

The goal of pretraining was twofold: first, to effectively increase the usable dataset size by leveraging more of the time series signals; and second, to initialize the finetuning on the actual task with meaningful representations that already capture relevant patterns in the data. The pretext task was designed to be close to the actual downstream task. Specifically, the model was trained to predict the PGA - one of the downstream regression targets - using random time windows from the input time series that were excluded during finetuning.

However, as mentioned in Sec. 4.6, the pretraining did not improve the model performance - regardless of whether the model base was frozen during the first 10 epochs of finetuning. It is difficult to determine why the models did not benefit from the pretraining. The pretrained models did not converge significantly faster during finetuning, nor did they appear to start finetuning with a lower validation loss. This suggests that the representations learned were not useful for the finetuning task. It may be that the prediction of PGA within a small 20s time window is too noisy or that the models could not handle the significantly higher amount of stations in the finetuning, which only recorded background noise within the initial 10s window.

There are numerous approaches for self-supervised pretraining, many of which can be applied on arbitrary model architectures:

1. **Masked Time-Series Modelling**
   Parts of the input time series could be masked out and the missing values could be predicted. This requires no labels like the PGA but could still teach the encoder about structures of seismic time series. Similar approaches are successful in Masked Autoencoders [He et al., 2021b] or NLP [Devlin et al., 2019b].

2. **Deep Metric Learning**
   Alternatively, one could pretrain the CNN using methods such as Contrastive Predictive Coding (CPC) [van den Oord et al., 2019], e.g. to distinguish whether two time series originate from the same earthquake.

3. **Non-contrastive Methods**
   There are also various non-contrastive methods (requiring no negative samples) such as self-distillation. For example, in Simple Siamese Representation Learning (SimSiam) [Chen and He, 2020] an input is augmented in two different ways, passed through asymmetrical encoder branches, and their outputs are encouraged to match.

Whether these methods are well suited to this dataset and task, and whether they ultimately improve model performance remains an open question. It may

also be that the benefits of pretraining are more evident when using different architectures; e.g. when using more data-hungry layers like transformers.

5.5. Deep Ensembles and Diversity Promotion

The mixture models in this work were composed of five independently trained probabilistic NNs. Their Combination leverages the diversity from different weight initalizations and stochasticity during training. Both mixture models outperformed the individual models they were built from.

However, combining independently trained models with identical architectures does not, by itself, encourage diversity. Applying a weighting scheme based on each model's predicted variances resulted in only marginal differences compared to the uniformly weighted mixture model. This means all individually trained models converged to similar representations. Therefore, two strategies were explored for jointly training the models in the ensemble to encourage specialization, both in terms of node or graph types and target variables.

The first approach, based on mean-weight regularization, only penalized the ensemble if models contributed unevenly. While this setup allowed specialization, it did not actively promote it, nor did it prevent model collapse in the ensemble. The second approach used the HSIC to force specialization across models by encouraging their predicted distributions to be independent. The participation loss additionally penalized if individual models contributed less than 5% to the overall output, which effectively prevented model collapse. This method achieved the best result across all experiments, second only to the model using an MLP for preprocessing static data.

In this work, HSIC was applied to the flattened vector of node-target pairs, promoting diversity across models in terms of both node types and targets. Alternatively, one could use the unflattened matrix of predicted variances to encourage model specialization in terms of nodes only. This would force models to put more emphasis on learning different types of nodes or graphs. Conversely, one could force the models to focus on different targets, but this could be achieved trivially by just predicting a single different IM per model. Specializing on certain structured subsets of nodes appears as a more interesting direction.

Notably, optimizing the ensemble involves a balancing between the main loss - i.e. node-level IM regression - with the regularization losses that promote diversity and prevent model collapse. This is determined by the weighting factors λ_{HSIC} and λ_{part}. In these experiments, both λ_{HSIC} and λ_{part}, were set to equal values and varied by one order of magnitude per experiment. This might not be the optimal solution and could be investigated further.

5.6. Training Limitations and Result Robustness

5.6.1. Hyperparameter Search Bottleneck

All models were trained on the SLURM cluster of the Carl von Ossietzky University Oldenburg using a single NVIDIA L40 GPU with 48 GB of GPU memory, 8 CPU cores, and 64 GB of RAM. Despite this high-performance hardware, a single training run lasted approximately 2.5 days, which consequently posed a major bottleneck for systematic hyperparameter exploration, as well as repeated training runs. Therefore, only a limited set of hyperparameter configurations could be evaluated, often only coarsely or in isolation.

Due to these time constraints and hardware limitations, an extensive hyperparameter search, e.g. through grid search or Bayesian optimization [Snoek et al., 2012], was not possible. Most decisions were thus based on intuition, prior works, or small-scale experiments. Hence, a more extensive tuning could potentially significantly improve performance for some training setups or architectural choices.

Furthermore, hyperparameters might interact in non-trivial ways; e.g. the best HSIC weight could depend on ensemble size. Since most experiments and hyperparameter tunings were done in isolation, it is not guaranteed that the used configurations are anywhere near-optimal. This should be kept in mind when considering the reported perfomance metrics.

5.6.2. Stochasticity and Uncertainty

Another challenge was the inherent stochasticity of DL training. Each training run involved randomness in weight initialization, time series paddings, dropped out stations, batching, and possibly numerical precision. Except for the five probabilistic models, from which the mixture models were built, each configuration was only trained once, meaning the variance across runs remains unknown.

This is problematic in different ways: first, whether the performance differences between models are statistically significant remains unclear; and second, there may be outliers - either unusually good or bad models due to random factors - which influences the results. An example can be seen for the 2nd probabilistic model in Fig. A.2 which significantly outperformed the other - identically trained - models, and even the mixture models and deep ensembles in Fig. 4.12 for a high time series padding.

The bootstrapping helps to get an intuition of the statistical uncertainty w.r.t. individual samples, but the systematic error due to random factors during training remains largely unknown. Thus, the robustness of the presented results cannot be strictly assessed.

Nevertheless, there were consistent trends present, such as mixture models and ensembles outperforming individual models, or models trained with station dropout performing better on shorter signals. This suggests that the most important findings are still meaningful. However, future work should ideally investigate the systematic error more thoroughly.

5.7. Combining Promising Techniques

Similarly to how hyperparameters were primarily tuned in isolation, most methods explored in this work, were investigated individually. The objective was not necessarily to create the best-performing model possible, but rather to understand how individual techniques, configurations and design choices affect the performance relative to the baseline.

However, many of these methods could also be combined. For example, one could finetune a pretrained deep ensemble with HSIC and participation loss, while applying station dropout and adding an MLP for preprocessing of static features. The benefits from these different methods may compound or interact synergistically, potentially leading to substantially better results - much like how the mixture models outperformed their individual components.

6. Conclusion

This work explored NNs for graph-structured time series at the example of seismic data for EEW. The focus was on the generalizability of the approaches and on understanding the effect of specific design choices, such as architecture, inputs, or optimization methods.

The models were highly sensitive to the lr, where models experienced complete mode collapse under poor settings. Static features were crucial, but only if preprocessed, e.g. through an MLP. Furthermore, applying station dropout not only enhanced the models capability of dealing with missing inputs, but also improved performance on shorter signals. Pretraining did not improve results for the given architecture and configuration. Mixture models and deep ensembles outperformed individual models. HSIC and participation loss actively encouraged specialization of models in a jointly trained ensemble while preventing model collapse. Using normalized, relative coordinates and edge weights introduced a trade-off: while improving generalization, it prevents implicit learning of geological and spatial priors and increases difficulty of spatial auxiliary tasks like epicenter prediction.

The work demonstrated how GNNs can be used for highly heterogeneous, dynamic graphs with sparse, limited time series from seismic data. It provided insights into the challenges and associated trade-offs of data preprocessing and graph construction. It revealed how small changes in architecture, such as the addition of an MLP for static data, can yield substantial benefits. Furthermore, it explored ways of pretraining for EEW, and strategies of improving model robustness to missing input data and shorter signals. Artificial nodes were used to generate earthquake intensity maps using DL, serving as a proof of concept; station dropout was used to estimate the error in these maps. Finally, this work investigated novel ways of jointly training probabilistic models in deep ensembles in an end-to-end fashion, while encouraging diversification of expertise in the individual models without causing collapse.

The explored methods could be relevant for real-time EEW with partial, noisy, and regionally diverse data. The experimental results suggest that thoughtful design choices can improve predictive capabilities and robustness of models. Carefully designed and optimized generalist models could be applied across different

geographic areas. Their predictions could be improved drastically by regional finetuning or adding geological priors during graph construction.

The best-performing methods could be combined to create stronger, more robust models. Exploring different architectures, the addition of richer features and regional priors without breaking generalizability, and improvements to pretraining could be focus points of future work. Hyperparameter tuning should be improved with more extensive exploration and result robustness should be enhanced through repeated training runs.

This work revealed some of the major challenges that accompany a paradigm shift from highly specialized, region-specific models to more powerful foundational models capable of generalization. Ultimately, advancing such EEW systems, like other applications, will require a combination of domain knowledge, appropriate data processing, efficient training strategies, and architectures that find a good balance between generalization and regional adaptation.

Bibliography

Bai, S., Kolter, J. Z., and Koltun, V. (2018). An empirical evaluation of generic convolutional and recurrent networks for sequence modeling.

Beyreuther, M., Barsch, R., Krischer, L., Megies, T., Behr, Y., and Wassermann, J. (2010). ObsPy: A Python Toolbox for Seismology. *Seismological Research Letters*, 81(3):530–533.

Bloemheuvel, S. (2021). Dataset for: Multivariate Time Series Regression with Graph Neural Networks.

Bloemheuvel, S., van den Hoogen, J., Jozinović, D., Michelini, A., and Atzmueller, M. (2023). Graph neural networks for multivariate time series regression with application to seismic data. *International Journal of Data Science and Analytics*, 16(3):317–332.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners.

Chen, T., Kornblith, S., Norouzi, M., and Hinton, G. (2020). A simple framework for contrastive learning of visual representations.

Chen, X. and He, K. (2020). Exploring simple siamese representation learning.

Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling.

Cini, A., Marisca, I., Zambon, D., and Alippi, C. (2023). Graph deep learning for time series forecasting.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019a). Bert: Pre-training of deep bidirectional transformers for language understanding.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019b). Bert: Pre-training of deep bidirectional transformers for language understanding.

Dumoulin, V. and Visin, F. (2018). A guide to convolution arithmetic for deep learning. *arXiv e-prints*.

Durall, R., Chatzimichailidis, A., Labus, P., and Keuper, J. (2020). Combating mode collapse in gan training: An empirical analysis using hessian eigenvalues.

Efron, B. (1992). *Bootstrap Methods: Another Look at the Jackknife*, pages 569–593. Springer New York, New York, NY.

Elson, P., de Andrade, E. S., Lucas, G., May, R., Hattersley, R., Campbell, E., Comer, R., Dawson, A., Little, B., Raynaud, S., scmc72, Snow, A. D., lgolston, Blay, B., Killick, P., lbdreyer, Peglar, P., Wilson, N., Andrew, Szymaniak, J., Berchet, A., Bosley, C., Davis, L., Filipe, Krasting, J., Bradbury, M., stephen-worsley, and Kirkham, D. (2024). SciTools/cartopy: REL: v0.24.1.

Fey, M. and Lenssen, J. E. (2019). Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

Geopy Contributors (2008). geopy: Geocoding library for python.

Gillies, S., van der Wel, C., Van den Bossche, J., Taves, M. W., Arnott, J., Ward, B. C., and others (2025). Shapely.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. (2017). Neural message passing for Quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 1263–1272. JMLR.org.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

Gretton, A., Bousquet, O., Smola, A., and Schölkopf, B. (2005). Measuring Statistical Dependence with Hilbert-Schmidt Norms. In Jain, S., Simon, H. U., and Tomita, E., editors, *Algorithmic Learning Theory*, pages 63–77, Berlin, Heidelberg. Springer Berlin Heidelberg.

Grill, J.-B., Strub, F., Altché, F., Tallec, C., Richemond, P. H., Buchatskaya, E., Doersch, C., Pires, B. A., Guo, Z. D., Azar, M. G., Piot, B., Kavukcuoglu, K.,

Munos, R., and Valko, M. (2020). Bootstrap your own latent: A new approach to self-supervised learning.

Gu, A., Goel, K., and Ré, C. (2022). Efficiently modeling long sequences with structured state spaces.

Hagberg, A., Swart, P., and S Chult, D. (2008). Exploring network structure, dynamics, and function using NetworkX. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

Hamilton, W. L., Ying, R., and Leskovec, J. (2018). Inductive representation learning on large graphs.

Han, X., Zhang, Z., Ding, N., Gu, Y., Liu, X., Huo, Y., Qiu, J., Yao, Y., Zhang, A., Zhang, L., Han, W., Huang, M., Jin, Q., Lan, Y., Liu, Y., Liu, Z., Lu, Z., Qiu, X., Song, R., Tang, J., Wen, J.-R., Yuan, J., Zhao, W. X., and Zhu, J. (2021). Pre-trained models: Past, present and future. *AI Open*, 2:225–250.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825):357–362.

He, K., Chen, X., Xie, S., Li, Y., Dollár, P., and Girshick, R. (2021a). Masked autoencoders are scalable vision learners.

He, K., Chen, X., Xie, S., Li, Y., Dollár, P., and Girshick, R. (2021b). Masked autoencoders are scalable vision learners.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. (2021). Lora: Low-rank adaptation of large language models.

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3):90–95.

INGV (2005). Rete sismica nazionale (rsn).

Jost, Z. (2020). Message passing as matrix multiplication. `https://github.com/zjost/blog_code/blob/master/gcn_numpy/message_passing.ipynb`. Accessed: 2024-11-07.

Jozinović, D., Lomax, A., Štajduhar, I., and Michelini, A. (2020). Rapid prediction of earthquake ground shaking intensity using raw waveform data and a convolutional neural network. *Geophysical Journal International*, 222(2):1379–1389.

Jozinović, D., Lomax, A., Štajduhar, I., and Michelini, A. (2021). Dataset - seismic data from central-western Italy used in the paper on rapid prediction of ground motion using a Convolutional Neural Network.

Kingma, D. P. and Ba, J. (2017). Adam: A method for stochastic optimization.

Kipf, T. N. and Welling, M. (2017). Semi-Supervised Classification with Graph Convolutional Networks.

Krizhevsky, A., Sutskever, I., and Hinton, G. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Neural Information Processing Systems*, 25.

Lakshminarayanan, B., Pritzel, A., and Blundell, C. (2017). Simple and scalable predictive uncertainty estimation using deep ensembles.

Lam, R., Sanchez-Gonzalez, A., Willson, M., Wirnsberger, P., Fortunato, M., Alet, F., Ravuri, S., Ewalds, T., Eaton-Rosen, Z., Hu, W., Merose, A., Hoyer, S., Holland, G., Vinyals, O., Stott, J., Pritzel, A., Mohamed, S., and Battaglia, P. (2023). Graphcast: Learning skillful medium-range global weather forecasting.

Lea, C., Vidal, R., Reiter, A., and Hager, G. D. (2016). Temporal convolutional networks: A unified approach to action segmentation.

Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

Li, G., Müller, M., Thabet, A., and Ghanem, B. (2019). DeepGCNs: Can GCNs Go As Deep As CNNs? In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 9266–9275.

Li, Q., Han, Z., and Wu, X.-m. (2018). Deeper Insights Into Graph Convolutional Networks for Semi-Supervised Learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1).

Loshchilov, I. and Hutter, F. (2017). Sgdr: Stochastic gradient descent with warm restarts.

McKinney, W. (2010). Data Structures for Statistical Computing in Python. In van der Walt, S. and Millman, J., editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56.

Michelini, A., Cianetti, S., Gaviano, S., Giunchi, C., Jozinović, D., and Lauciani, V. (2021). INSTANCE – the Italian seismic dataset for machine learning. *Earth System Science Data*, 13(12):5509–5544.

OpenAI (2025). ChatGPT. `https://chatgpt.com/`. [Large Language Model].

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Seo, Y., Defferrard, M., Vandergheynst, P., and Bresson, X. (2018). Structured Sequence Modeling with Graph Convolutional Recurrent Networks. In Cheng, L., Leung, A. C. S., and Ozawa, S., editors, *Neural Information Processing*, pages 362–373, Cham. Springer International Publishing.

Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms.

Song, L., Smola, A., Gretton, A., Bedo, J., and Borgwardt, K. (2012). Feature selection via dependence maximization. *Journal of Machine Learning Research*, 13:1393 – 1434. Cited by: 328.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2015). Rethinking the inception architecture for computer vision.

van den Oord, A., Li, Y., and Vinyals, O. (2019). Representation learning with contrastive predictive coding.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2023). Attention is all you need.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. (2018). Graph attention networks.

von Luxburg, U. (2020). Statistical Machine Learning Part 19 - The reproducing kernel Hilbert space. `https://www.youtube.com/watch?v=EoM_DF3VAO8`. Lecture, University of Tübingen, published on YouTube, accessed 2025-04-09.

Wang, T., Dai, X., and Liu, Y. (2021). Learning with Hilbert–Schmidt independence criterion: A review and new perspectives. *Know.-Based Syst.*, 234(C).

Worden, C., Thompson, E., and Wald, D. (2020). Shakemap manual online: technical manual, user's guide, and software guide. U. S. Geological Survey.

Wu, Y. and Liu, L. (2023). Selecting and Composing Learning Rate Policies for Deep Neural Networks. *ACM Trans. Intell. Syst. Technol.*, 14(2).

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. (2019). How powerful are graph neural networks?

Zender, J. M. (2025). Neural Networks for Graph-Structured Time-Series with Application to Seismic Data. `https://github.com/JZen97/masterthesis`.

Zheng, Y., Liu, Q., Chen, E., Ge, Y., and Zhao, J. L. (2014). Time Series Classification Using Multi-Channels Deep Convolutional Neural Networks. In Li, F., Li, G., Hwang, S.-w., Yao, B., and Zhang, Z., editors, *Web-Age Information Management*, pages 298–310, Cham. Springer International Publishing.

Zippenfenig, P. (2023). Open-Meteo.com Weather API.

A. Appendix

A.1. Python Code

A.1.1. Models

Lst. A.1: Baseline Model, c.f. Fig. 3.5.

```
import torch
import torch.nn as nn
import torch_geometric

class BaselineModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()
        # Time Series Encoder
        self.conv1 = nn.Conv1d(in_channels=3, out_channels=32,
            kernel_size=125, stride=2)
        self.conv2 = nn.Conv1d(in_channels=32, out_channels=64,
            kernel_size=125, stride=2)
        # Graph Layers
        self.gcn1 = torch_geometric.nn.GCNConv(10051, 64,
            add_self_loops=True, normalize=True, bias=False)
        self.gcn2 = torch_geometric.nn.GCNConv(64, 64,
            add_self_loops=True, normalize=True, bias=False)
        # Output Layers
        self.fc_ims = nn.Linear(64, 5) # PGA, PGV, SA03, SA10,
            SA30
        self.fc_epi = nn.Linear(64, 2) # epicenter position

    def forward(self, x1_ts, x2_static, edge_index, edge_weight,
        batch):
        x = self.relu(self.conv1(x1_ts)) # out: [bs, 32, 438]
        x = self.relu(self.conv2(x)) # out: [bs, 64, 157]
        # flatten the cnn output
        x = torch.flatten(x, start_dim=1) # out: [bs, 10048]
        # Concatenate output with static features
        x = torch.concat([x, x2_static], dim=1) # out: [bs,
            10051]
```

```
        # Apply graph layers
        x = self.relu(self.gcn1(x, edge_index=edge_index,
            edge_weight=edge_weight)) # out: [bs, 64]
        x = self.tanh(self.gcn2(x, edge_index=edge_index,
            edge_weight=edge_weight)) # out: [bs, 64]
        # Node level predicitons (Intensity Measurements IMs)
        ims = self.fc_ims(x) # out: [bs, 5]
        # Graph level predictions (Epicenter)
        x_pooled = torch_geometric.nn.global_mean_pool(x, batch)
        epi = self.fc_epi(x_pooled) # out [bs, 2]
        return ims, epi
```

Lst. A.2: Standard Model, predicting only IMs.

```
class StandardModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()
        self.conv1 = nn.Conv1d(in_channels=3, out_channels=32,
            kernel_size=125, stride=2)
        self.conv2 = nn.Conv1d(in_channels=32, out_channels=64,
            kernel_size=125, stride=2)
        self.gcn1 = torch_geometric.nn.GCNConv(10051, 64,
            add_self_loops=True, normalize=True, bias=False)
        self.gcn2 = torch_geometric.nn.GCNConv(64, 64,
            add_self_loops=True, normalize=True, bias=False)
        self.fc_ims = nn.Linear(64, 5)

    def forward(self, x1_ts, x2_static, edge_index, edge_weight):
        x = self.relu(self.conv1(x1_ts))
        x = self.relu(self.conv2(x))
        x = torch.flatten(x, start_dim=1)
        x = torch.concat([x, x2_static], dim=1)
        x = self.relu(self.gcn1(x, edge_index=edge_index,
            edge_weight=edge_weight))
        x = self.tanh(self.gcn2(x, edge_index=edge_index,
            edge_weight=edge_weight))
        ims = self.fc_ims(x)
        return ims
```

Lst. A.3: Standard Model, without static data.

```
class StandardModel_no_static_feats(nn.Module):
    def __init__(self):
        super().__init__()
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()
```

```
        self.conv1 = nn.Conv1d(in_channels=3, out_channels=32,
            kernel_size=125, stride=2)
        self.conv2 = nn.Conv1d(in_channels=32, out_channels=64,
            kernel_size=125, stride=2)
        self.gcn1 = torch_geometric.nn.GCNConv(10048, 64,
            add_self_loops=True, normalize=True, bias=False)
        self.gcn2 = torch_geometric.nn.GCNConv(64, 64,
            add_self_loops=True, normalize=True, bias=False)
        self.fc_ims = nn.Linear(64, 5)

    def forward(self, x1_ts, edge_index, edge_weight):
        x = self.relu(self.conv1(x1_ts))
        x = self.relu(self.conv2(x))
        x = torch.flatten(x, start_dim=1)
        x = self.relu(self.gcn1(x, edge_index=edge_index,
            edge_weight=edge_weight))
        x = self.tanh(self.gcn2(x, edge_index=edge_index,
            edge_weight=edge_weight))
        ims = self.fc_ims(x)
        return ims
```

Lst. A.4: Standard Model, with an MLP for static data.

```
class StandardModel_MLP_static_feats(nn.Module):
    def __init__(self):
        super().__init__()
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()
        self.conv1 = nn.Conv1d(in_channels=3, out_channels=32,
            kernel_size=125, stride=2)
        self.conv2 = nn.Conv1d(in_channels=32, out_channels=64,
            kernel_size=125, stride=2)
        # Static Feature MLP
        self.dropout = nn.Dropout(0.3)
        self.fc1 = nn.Linear(3, 16)
        self.bn1 = nn.BatchNorm1d(16)
        self.fc2 = nn.Linear(16, 32)
        self.bn2 = nn.BatchNorm1d(32)
        self.fc3 = nn.Linear(32, 64)
        self.mlp = nn.Sequential(
            self.fc1, self.relu, self.bn1, self.dropout, self.fc2
                , self.relu, self.bn2, self.dropout, self.fc3,
                self.relu
        )
        self.gcn1 = torch_geometric.nn.GCNConv(10112, 64,
            add_self_loops=True, normalize=True, bias=False)
        self.gcn2 = torch_geometric.nn.GCNConv(64, 64,
            add_self_loops=True, normalize=True, bias=False)
```

```
        self.fc_ims = nn.Linear(64, 5)

    def forward(self, x1_ts, x2_static, edge_index, edge_weight):
        x = self.relu(self.conv1(x1_ts))
        x = self.relu(self.conv2(x))
        x = torch.flatten(x, start_dim=1)
        static_feats = self.mlp(x2_static)
        x = torch.concat([x, static_feats], dim=1)
        x = self.relu(self.gcn1(x, edge_index=edge_index,
            edge_weight=edge_weight))
        x = self.tanh(self.gcn2(x, edge_index=edge_index,
            edge_weight=edge_weight))
        ims = self.fc_ims(x)
        return ims
```

Lst. A.5: Standard Model, with raw time-series tokens in first GCN layer.

```
class StandardModel_raw_tokens(nn.Module):
    def __init__(self):
        super().__init__()
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()
        self.conv1 = nn.Conv1d(in_channels=3, out_channels=32,
            kernel_size=125, stride=2)
        self.conv2 = nn.Conv1d(in_channels=32, out_channels=64,
            kernel_size=125, stride=2)
        self.gcn1 = torch_geometric.nn.GCNConv(10351, 64,
            add_self_loops=True, normalize=True, bias=False)
        self.gcn2 = torch_geometric.nn.GCNConv(64, 64,
            add_self_loops=True, normalize=True, bias=False)
        self.fc_ims = nn.Linear(64, 5)

    def forward(self, x1_ts, x2_static, edge_index, edge_weight):
        x = self.relu(self.conv1(x1_ts))
        x = self.relu(self.conv2(x))
        x = torch.flatten(x, start_dim=1)
        x1_ts_flat_last_100t = torch.flatten(x1_ts[:, :, 900:],
            start_dim=1)
        x = torch.concat([x, x1_ts_flat_last_100t], dim=1)
        x = torch.concat([x, x2_static], dim=1)
        x = self.relu(self.gcn1(x, edge_index=edge_index,
            edge_weight=edge_weight))
        x = self.tanh(self.gcn2(x, edge_index=edge_index,
            edge_weight=edge_weight))
        ims = self.fc_ims(x)
        return ims
```

Lst. A.6: Standard Model, modified for pretraining using the PGA.

```
class StandardModel_Pretraining_PGA(nn.Module):
    def __init__(self):
        super().__init__()
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()
        self.conv1 = nn.Conv1d(in_channels=3, out_channels=32,
            kernel_size=125, stride=2)
        self.conv2 = nn.Conv1d(in_channels=32, out_channels=64,
            kernel_size=125, stride=2)
        self.gcn1 = torch_geometric.nn.GCNConv(10051, 64,
            add_self_loops=True, normalize=True, bias=False)
        self.gcn2 = torch_geometric.nn.GCNConv(64, 64,
            add_self_loops=True, normalize=True, bias=False)
        self.fc_pga = nn.Linear(64, 1)

    def forward(self, x1_ts, x2_static, edge_index, edge_weight,
        batch):
        x = self.relu(self.conv1(x1_ts))
        x = self.relu(self.conv2(x))
        x = torch.flatten(x, start_dim=1)
        x = torch.concat([x, x2_static], dim=1)
        x = self.relu(self.gcn1(x, edge_index=edge_index,
            edge_weight=edge_weight))
        x = self.tanh(self.gcn2(x, edge_index=edge_index,
            edge_weight=edge_weight))
        pga = self.fc_pga(x)
        return pga
```

Lst. A.7: Probabilistic Model, predicting parameters of a Gaussian distribution.

```
class ProbabilisticModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()
        self.conv1 = nn.Conv1d(in_channels=3, out_channels=32,
            kernel_size=125, stride=2)
        self.conv2 = nn.Conv1d(in_channels=32, out_channels=64,
            kernel_size=125, stride=2)
        self.gcn1 = torch_geometric.nn.GCNConv(10051, 64,
            add_self_loops=True, normalize=True, bias=False)
        self.gcn2 = torch_geometric.nn.GCNConv(64, 64,
            add_self_loops=True, normalize=True, bias=False)
        self.fc_mean = nn.Linear(64, 5)
        self.fc_variance = nn.Linear(64, 5)

    def forward(self, x1_ts, x2_static, edge_index, edge_weight):
        x = self.relu(self.conv1(x1_ts))
```

```
        x = self.relu(self.conv2(x))
        x = torch.flatten(x, start_dim=1)
        x = torch.concat([x, x2_static], dim=1)
        x = self.relu(self.gcn1(x, edge_index=edge_index,
            edge_weight=edge_weight))
        x = self.tanh(self.gcn2(x, edge_index=edge_index,
            edge_weight=edge_weight))
        mean = self.fc_mean(x)
        variance = self.fc_variance(x)
        # Enforce positivity of the variances. The 1e-6 is added
            for numerical stability
        variance = torch.log(1 + torch.exp(variance)) + 1e-6
        return mean, variance
```

Lst. A.8: Mixture model, containing individually trained probabilistic models as in Lst. A.7. Parameters of the output distributions are calculated using the desired weighting scheme.

```
class MixtureModel(nn.Module):
    def __init__(self, state_dict_paths, weighting_scheme='
        uniform'):
        super().__init__()
        self.num_models = len(state_dict_paths)
        self.models = nn.ModuleList([ProbabilisticModel() for _
            in range(len(state_dict_paths))])
        self.weighting_scheme = weighting_scheme
        # load model weights
        for model, path in zip(self.models, state_dict_paths):
            checkpoint = torch.load(path, map_location=device)
            state_dict = checkpoint['model_state_dict']
            model.load_state_dict(state_dict, strict=True)

    def forward(self, *inputs):
        outputs = [model(*inputs) for model in self.models]
        # `outputs` will be a list of tuples: [(mean1, var1), (
            mean2, var2), ...]
        means, variances = zip(*outputs)
        means, variances = torch.stack(means, dim=2), torch.stack
            (variances, dim=2)
        if self.weighting_scheme == 'uniform':
            weights = torch.ones_like(variances, device=device) /
                self.num_models
        elif self.weighting_scheme == 'variance':
            weights = F.softmax(-variances, dim=2)
        else:
            raise ValueError(f'Unknown weighting scheme: {self.
                weighting_scheme}')
```

```
        # compute joint weighted mean and variance. Sum across model dimension: [nodes, targets, models] -> [nodes, targets]
        mean = torch.sum(weights * means, dim=2)
        variance = torch.sum(weights * (variances + means**2), dim=2) - mean**2
        return mean, variance
```

Lst. A.9: Deep Ensemble, consisting of probabilistic models as in Lst. A.7.

```
class DeepEnsemble(nn.Module):
    def __init__(self, num_models):
        super().__init__()
        self.num_models = num_models
        self.models = nn.ModuleList([ProbabilisticModel() for _ in range(num_models)])

    def forward(self, *inputs):
        outputs = [model(*inputs) for model in self.models]
        # `outputs` will be a list of tuples: [(mean1, var1), (mean2, var2), ...]
        means, variances = zip(*outputs)
        means, variances = torch.stack(means, dim=2), torch.stack(variances, dim=2)
        # output has shape [nodes x targets x models]
        return means, variances
```

A.1.2. Optimization and Loss Functions

Lst. A.10: Computation of the NLL loss for probabilistic models as in Lst. A.7 according to Eq. 3.5.

```
def NLLLoss(mean, variance, targets):
    loss = torch.sum(torch.log(variance) / 2 + ((targets - mean) ** 2) / (2 * variance))
    return loss
```

Lst. A.11: Computation of the weights and ensemble mean and variances as in Eq. 3.10 and Eq. 3.11

```
means, variances = model(x1_ts, x2_static, edge_index, edge_weight) # forward pass
# Calculate model weights for each node and target
weights = F.softmax(-variances, dim=2)
# compute joint weighted mean and variance
ensemble_mean = torch.sum(weights * means, dim=2) # sum across model dimension: [nodes, targets, models] -> [nodes, targets]
```

```
ensemble_variance = torch.sum(weights * (variances + means**2),
    dim=2) - mean**2
```

Lst. A.12: Computation of the mean weight regularization loss for deep ensembles, according to Eq. 3.16

```
def MeanWeightRegularization(weights):
    # weights shape: [nodes, targets, models]
    num_nodes = weights.size(0)
    num_targets = weights.size(1)
    # compute sum of weights across models and normalize across
        nodes and targets
    total_weights = weights.sum(dim=(0,1)) / (num_nodes *
        num_targets)  # [models]
    # compute mean of total weights across models
    mean_weight = total_weights.mean() # scalar
    # Regularization loss: mean squared deviation from mean
        weight
    reg_loss = ((total_weights - mean_weight) ** 2).mean()
    return reg_loss
```

Lst. A.13: Kernel matrix computation for HSIC loss using an RBF kernel.

```
def get_kernel_matrix(data, gamma=1.0):
    # data: torch.tensor([n_samples, n_features])
    norms = (data**2).sum(1).reshape(-1, 1)
    dists = norms - 2 * data @ data.T + norms.T
    return torch.exp(-dists / (2*gamma**2))
```

Lst. A.14: Compuation of the unbiased HSIC loss as in Eq. 3.23 between two models in the ensemble.

```
def get_hsic_loss_unbiased(X, Y, gamma=1.0):
    n = X.size(0)
    # kernelize model outputs
    K = get_kernel_matrix(X, gamma)
    L = get_kernel_matrix(Y, gamma)
    # set diagonals to zero
    K = K - torch.diag_embed(torch.diagonal(K))
    L = L - torch.diag_embed(torch.diagonal(L))
    ones = torch.ones(n,1, device=X.device)
    term_1 = torch.trace(K @ L)
    term_2 = (ones.T @ K @ ones * ones.T @ L @ ones) / ((n-1)*(n
        -2))
    term_3 = (-2 / (n-2)) * (ones.T @ K @ L @ ones)
    hsic = (term_1 + term_2 + term_3) / (n*(n-3))
    return hsic
```

Lst. A.15: Compuation of the unbiased HSIC loss as in Eq. 3.25 between all combinations of models in the ensemble.

```
def compute_ensemble_hsic_loss(variances, gamma=1.0, unbiased=
    True, normalize=True):
    # Get number of models
    num_models = variances.shape[-1]
    # Flatten node-target pairs to 2D: [nodes * targets, models]
    flat_vars = variances.view(-1, num_models)  # shape: [N*T, M]
    # Accumulate HSIC across model pairs
    total_hsic = torch.tensor([0.0], device=variances.device)
    for i, j in itertools.combinations(range(num_models), 2):
        Xi = flat_vars[:, i].unsqueeze(1)  # shape [N*T, 1]
        Xj = flat_vars[:, j].unsqueeze(1)  # shape [N*T, 1]
        total_hsic += get_hsic_loss_unbiased(Xi, Xj, gamma=gamma)
            .squeeze()
    if normalize:
        total_hsic /= num_models * (num_models - 1) / 2
    return total_hsic
```

Lst. A.16: Compuation of the participation loss for the ensemble as in Eq. 3.26.

```
def get_participation_loss(weights, threshold=0.05):
    avg_weights = weights.mean(dim=(0,1)) # shape: [models]
    loss = torch.relu(threshold - avg_weights).mean()
    return loss
```

A.2. Additional Results

A.2.1. Pretraining

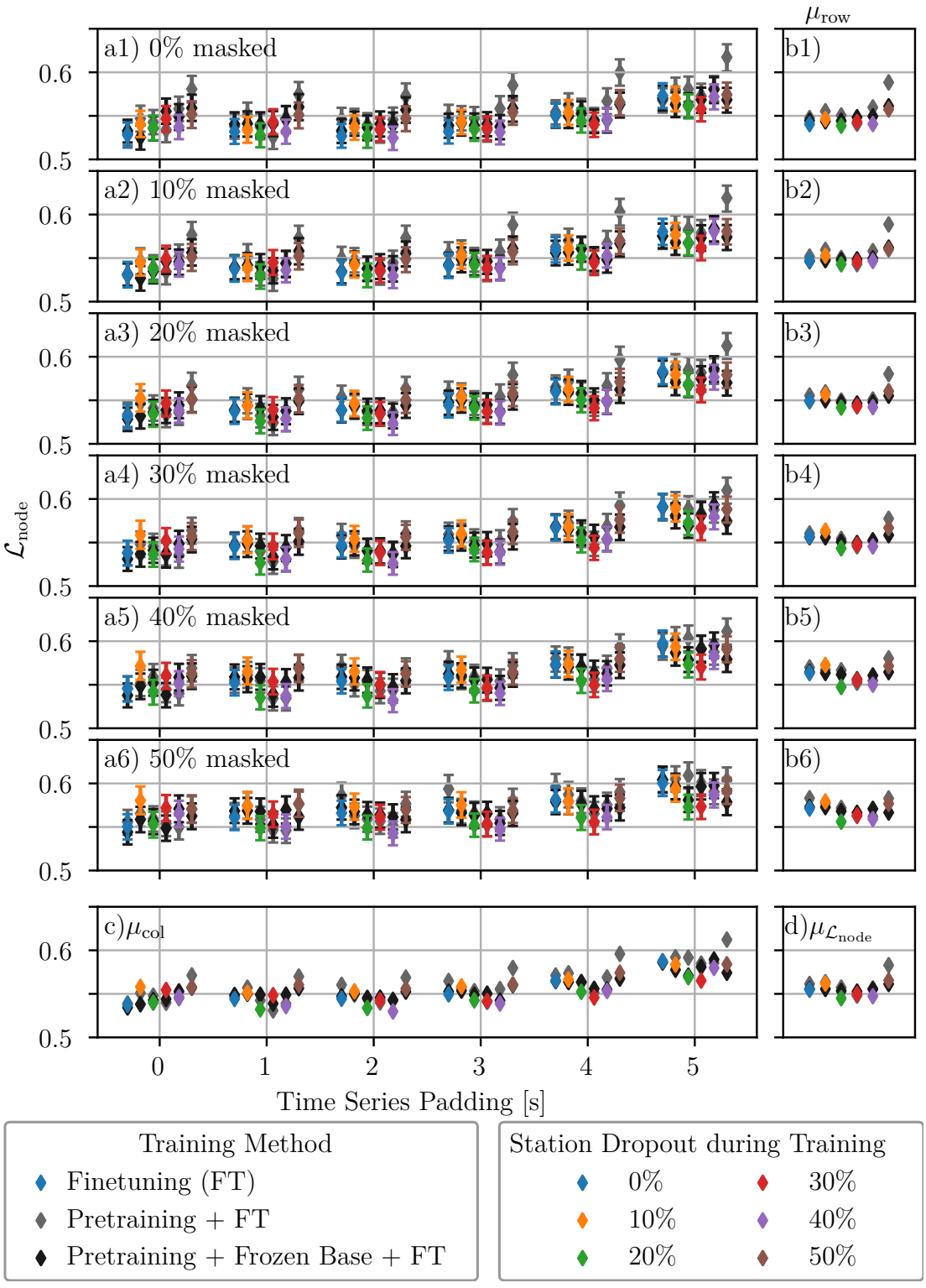


Figure A.1.: Corresponding to Fig. 4.9, including the results from simple finetuning, pretraining and finetuning, and pretraining with finetuning where the base is frozen in the first 10 epochs.

A.2.2. Probabilistic Models, Mixture Models and Deep Ensembles

Table A.1.: MSE loss (Eq. 3.2) on the test dataset for five randomly initialized **probabilistic models** trained on the NLL loss (c.f. Eq. 3.5). $\mathcal{L}_{\text{node}}$ is the average across all target variables, which is the metric used for model selection. M is the metric, U and L are the upper and lower bounds of the 95% confidence interval. The best result for each time-series padding is highlighted in bold.

Model ID		$\mathcal{L}_{\text{node}}$ for different Time-Series Paddings						
		0s	1s	2s	3s	4s	5s	$\mu_{\mathcal{L}_{\text{node}}}$
1	U	0.5713	0.5653	0.5530	0.5595	0.5682	0.5909	0.5541
	M	0.5563	0.5511	0.5399	0.5461	**0.5547**	0.5768	
	L	0.5409	0.5365	0.5260	0.5319	0.5404	0.5623	
2	U	0.5509	0.5516	0.5460	0.5632	0.5752	0.5762	**0.5465**
	M	0.5360	0.5369	0.5327	0.5488	0.5612	**0.5631**	
	L	0.5211	0.5223	0.5192	0.5344	0.5465	0.5489	
3	U	0.5565	0.5451	0.5399	0.5600	0.5894	0.6066	0.5523
	M	0.5424	0.5315	**0.5268**	0.5462	0.5751	0.5918	
	L	0.5275	0.5178	0.5135	0.5328	0.5601	0.5761	
4	U	0.5651	0.5468	0.5458	0.5630	0.5802	0.6080	0.5547
	M	0.5512	0.5335	0.5326	0.5495	0.5673	0.5940	
	L	0.5371	0.5196	0.5196	0.5354	0.5528	0.5787	
5	U	0.5460	0.5434	0.5432	0.5504	0.5687	0.6129	0.5472
	M	**0.5323**	**0.5301**	0.5300	**0.5372**	0.5554	0.5980	
	L	0.5181	0.5167	0.5161	0.5238	0.5415	0.5829	

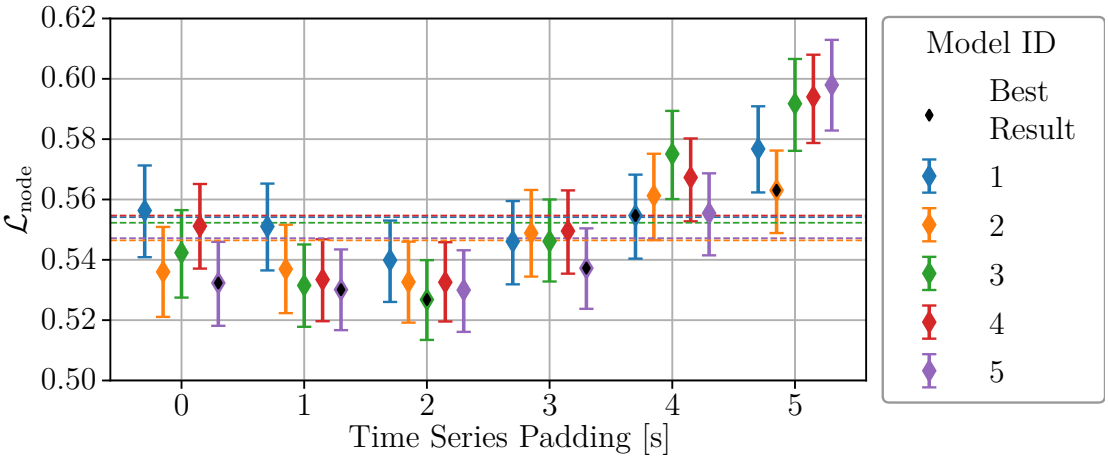


Figure A.2.: Visualization of Tab. A.1 (**probabilistic models**). The markers show $\mathcal{L}_{\text{node}}$ of the test dataset for different time series paddings. The caps mark the corresponding 95% confidence interval.

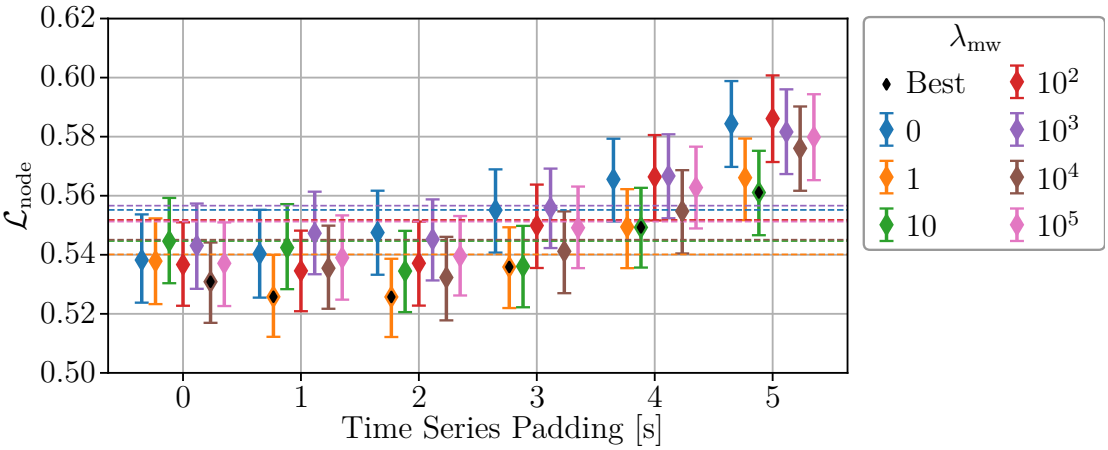


Figure A.3.: Visualization of Tab. A.2 (**deep ensemble with mean weight regularization**). The markers show $\mathcal{L}_{\text{node}}$ of the test dataset for different time series paddings. The caps mark the corresponding 95% confidence interval.

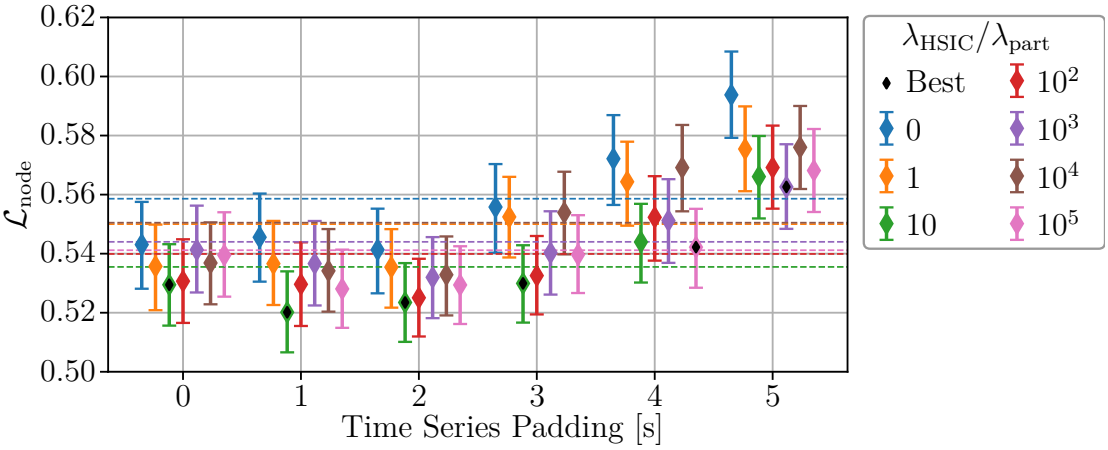


Figure A.4.: Visualization of Tab. A.3 (**deep ensemble with HSIC and participation loss**). The markers show $\mathcal{L}_{\text{node}}$ of the test dataset for different time series paddings. The caps mark the corresponding 95% confidence interval.

Table A.2.: MSE loss (Eq. 3.2) on the test dataset for the **deep ensemble** model (c.f. Sec. A.9) trained with the **mean weight regularization loss**, according to Eq. 3.17. The model was trained using different weights λ_{HSIC} and λ_{part} for the regularization terms. $\mathcal{L}_{\text{node}}$ is the average across all target variables, which is the metric used for model selection. M is the metric, U and L are the upper and lower bounds of the 95% confidence interval. The best result for each time-series padding is highlighted in bold.

λ_{mw}		$\mathcal{L}_{\text{node}}$ **for different Time-Series Paddings**						
		0s	1s	2s	3s	4s	5s	$\mu_{\mathcal{L}_{\text{node}}}$
0	U	0.5537	0.5552	0.5617	0.5689	0.5793	0.5988	0.5552
	M	0.5382	0.5403	0.5475	0.5551	0.5656	0.5844	
	L	0.5238	0.5255	0.5332	0.5407	0.5513	0.5698	
1	U	0.5523	0.5400	0.5386	0.5493	0.5622	0.5794	**0.5401**
	M	0.5378	**0.5258**	**0.5257**	**0.5359**	0.5494	0.5661	
	L	0.5233	0.5122	0.5122	0.5220	0.5355	0.5517	
10	U	0.5593	0.5571	0.5481	0.5498	0.5627	0.5752	0.5447
	M	0.5446	0.5424	0.5345	0.5361	**0.5493**	**0.5611**	
	L	0.5304	0.5284	0.5206	0.5222	0.5357	0.5466	
10^2	U	0.5510	0.5482	0.5512	0.5638	0.5805	0.6007	0.5518
	M	0.5367	0.5346	0.5371	0.5499	0.5664	0.5861	
	L	0.5227	0.5209	0.5228	0.5355	0.5516	0.5714	
10^3	U	0.5573	0.5614	0.5588	0.5692	0.5808	0.5960	0.5566
	M	0.5431	0.5473	0.5452	0.5559	0.5667	0.5816	
	L	0.5285	0.5334	0.5313	0.5423	0.5524	0.5673	
10^4	U	0.5442	0.5499	0.5461	0.5547	0.5687	0.5902	0.5451
	M	**0.5309**	0.5354	0.5323	0.5412	0.5548	0.5760	
	L	0.5169	0.5217	0.5178	0.5270	0.5404	0.5617	
10^5	U	0.5510	0.5533	0.5531	0.5631	0.5766	0.5944	0.5512
	M	0.5371	0.5390	0.5396	0.5492	0.5628	0.5799	
	L	0.5226	0.5248	0.5262	0.5355	0.5489	0.5653	

Table A.3.: MSE loss (Eq. 3.2) on the test dataset for the **deep ensemble** model (c.f. Sec. A.9) trained with the **unbiased HSIC and participation loss**, according to Eq. 3.27. The model was trained using different weights λ_{HSIC} and λ_{part} for the regularization terms. $\mathcal{L}_{\text{node}}$ is the average across all target variables, which is the metric used for model selection. M is the metric, U and L are the upper and lower bounds of the 95% confidence interval. The best result for each time-series padding is highlighted in bold.

$\lambda_{\text{HSIC}}/\lambda_{\text{part}}$		$\mathcal{L}_{\text{node}}$ **for different Time-Series Paddings**						
		0s	1s	2s	3s	4s	5s	$\mu_{\mathcal{L}_{\text{node}}}$
0	U	0.5575	0.5603	0.5552	0.5703	0.5869	0.6085	0.5586
	M	0.5431	0.5456	0.5413	0.5558	0.5722	0.5938	
	L	0.5281	0.5305	0.5266	0.5403	0.5565	0.5792	
1	U	0.5498	0.5511	0.5483	0.5660	0.5779	0.5899	0.5500
	M	0.5357	0.5366	0.5354	0.5524	0.5643	0.5754	
	L	0.5209	0.5226	0.5217	0.5387	0.5495	0.5611	
10	U	0.5432	0.5340	0.5368	0.5429	0.5569	0.5798	**0.5355**
	M	**0.5295**	**0.5202**	**0.5235**	**0.5299**	0.5439	0.5661	
	L	0.5156	0.5066	0.5101	0.5167	0.5302	0.5519	
10^2	U	0.5449	0.5438	0.5383	0.5460	0.5662	0.5834	0.5399
	M	0.5307	0.5296	0.5251	0.5326	0.5523	0.5692	
	L	0.5165	0.5155	0.5120	0.5194	0.5376	0.5552	
10^3	U	0.5563	0.5511	0.5456	0.5544	0.5652	0.5771	0.5440
	M	0.5413	0.5366	0.5320	0.5403	0.5512	**0.5626**	
	L	0.5269	0.5225	0.5182	0.5261	0.5369	0.5484	
10^4	U	0.5506	0.5483	0.5458	0.5677	0.5836	0.5900	0.5505
	M	0.5368	0.5342	0.5329	0.5539	0.5691	0.5760	
	L	0.5229	0.5204	0.5191	0.5398	0.5543	0.5619	
10^5	U	0.5540	0.5414	0.5425	0.5530	0.5552	0.5822	0.5412
	M	0.5395	0.5281	0.5294	0.5397	**0.5422**	0.5681	
	L	0.5254	0.5149	0.5162	0.5267	0.5285	0.5541	

Hinweis zum Einsatz von Generativer KI

Im Rahmen dieser Arbeit wurde ChatGPT [OpenAI, 2025] unterstützend eingesetzt. Die Nutzung umfasste:

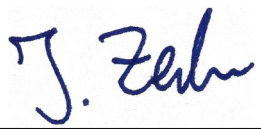
- Verbesserung und Verfeinerung sprachlicher Formulierungen
- Unterstützung beim Debugging von Code (Python, LaTeX)
- Vorschläge zur Gliederung und Strukturierung
- Generierung von Boilerplate-Code (z.B. Abbildungsumgebungen in LaTeX und matplotlib)

Alle inhaltlichen Entscheidungen, Argumentationen, Bewertungen und wissenschaftlichen Analysen wurden eigenständig getroffen. Die Nutzung erfolgte im Sinne unterstützender Werkzeuge und unter Wahrung guter wissenschaftlicher Praxis gemäß den Leitlinien der Carl von Ossietzky Universität Oldenburg.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen verwendet, sowie keine unzulässige fremde Hilfe in Anspruch genommen habe. Zudem wurde die Arbeit nicht unter unkenntlichem Einsatz generativer KI erbracht. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlichen Arbeitens und Veröffentlichens, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg vorgelegt sind, befolgt habe.

Weiterhin erkläre ich, dass die Arbeit in gleicher und ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.



Oldenburg, 07.05.2025