universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„LRCP: Evaluation and Application of Prevalent Linear Regression Techniques Used in Channel Pruning To Enable GNN Inference in Resource-Constrained Environments“

verfasst von / submitted by

Lorenz Börtlein B.Sc.

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2023 / Vienna, 2023

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066 921
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Informatik
Betreut von / Supervisor:	Univ. Prof. Dr. Siegfried Benkner
Mitbetreut von / Co-Supervisor:	Dott. mag. Dr. Atakan Aral

Abstract

Graph Neural Networks (GNNs) have been proven to be remarkably effective in solving graph prediction tasks, and help with modeling spatio-temporal data. However, they suffer from great computational intensity and memory requirements, making efficient model compression and acceleration essential. A popular approach is channel pruning, which eliminates redundant feature dimensions inside middle layers of GNNs without sacrificing performance.

This thesis evaluates linear regression techniques used to identify dimensions to remove during channel pruning. Therefore, three popular approaches are being compared in Lasso regression, Ridge regression, and Elastic Net regression by conducting extensive experiments to evaluate their efficacy while retaining the predictive performance of the networks.

In addition, the channel pruning approach with all tested linear regression techniques is also applied to spatio-temporal models of the Torch Spatiotemporal library. They are part of the SWAIN project's analytical tools to survey possible pollution of waterways by examining water samples with resource-constrained devices on the network's edge. These require computational efficiency. The experiments indicated that the proposed channel pruning technique compresses the spatio-temporal GNNs effectively while maintaining competitive performance.

The contributions of this work include a thorough evaluation of linear regression techniques for channel pruning GNNs, to provide insights into their comparative performance. Moreover, the successful application of channel pruning to spatio-temporal models has been demonstrated for the first time. This showcases the potential for deploying spatio-temporal models on resource-constrained edge devices, such as Raspberry Pis or Arduinos.

Keywords— Channel pruning, Graph Neural Networks, Linear regression techniques, Lasso regression, Ridge regression, Elastic Net regression, Spatio-temporal models, Torch Spatio-Temporal Library

Kurzfassung

Graphische neuronale Netze (GNN) haben sich in verschiedenen Bereichen wie der Graphenanalyse und der räumlich-zeitlichen Modellierung als sehr erfolgreich erwiesen. Die zunehmende Komplexität von GNN-Architekturen erfordert jedoch effiziente Modellkomprimierungstechniken, um die Rechenkosten und den Speicherbedarf zu reduzieren. Channel Pruning, eine beliebte Komprimierungstechnik, zielt darauf ab, redundante Kanäle in GNNs zu eliminieren und gleichzeitig deren Leistung zu erhalten. Insbesondere der Einsatz von GNNs auf ressourcenbeschränkten Geräten in der Nähe des Netzwerkrandes ist noch nicht erforscht worden.

Diese Masterarbeit konzentriert sich auf die Evaluierung der Wirksamkeit von linearen Regressionstechniken welche innerhalb von Channel Pruning Techniken für GNNs verwendet werden. Insbesondere untersuchen wir die Leistung von Lasso-Regression, Ridge-Regression und Elastic Net Regression. Durch umfangreiche Experimente zeigen wir, dass Ridge- und Elastic Net-Regression die Lasso-Regression bei der Erhaltung der GNN-Leistung während des Channel Pruning-Prozesses durchweg übertreffen.

Darüber hinaus wenden wir die identifizierten Regressionstechniken auf raum-zeitliche Modelle innerhalb der Torch Spatio-Temporal Library an. Diese Bibliothek bietet einen umfassenden Satz von Werkzeugen und Modellen für die Analyse von raum-zeitlichen Daten. Sie wird im SWAIN-Projekt verwendet, um Wasserproben von möglicherweise verschmutzten Flüssen auf ressourcenbeschränkten Geräten zu untersuchen. Diese erfordern, dass die verwendeten GNNs recheneffizient sind. Dies macht es zu einem idealen Testfeld, um die Effektivität des Channel Prunings in diesem Kontext zu evaluieren. Unsere Experimente zeigen, dass der vorgeschlagene Channel Pruning-Ansatz erfolgreich räumlich-zeitliche GNN-Modelle komprimiert und dabei eine konkurrenzfähige Leistung beibehält.

Zu den Beiträgen dieser Arbeit gehört eine gründliche Evaluierung von linearen Regressionstechniken für das Channel Pruning in GNNs, die Einblicke in ihre vergleichbare Leistung bietet. Darüber hinaus demonstrieren wir die erfolgreiche Anwendung von Channel Pruning auf raum-zeitliche Modelle in der Torch Spatiotemporal Bibliothek und zeigen damit das Potenzial für den Einsatz von raum-zeitlichen Modellen auf ressourcenbeschränkten Geräten am Netzwerkrand.

Contents

List of Tables

List of Figures

List of Algorithms

Listings

1 Introduction

Cloud computing has been remarkably prevalent in the information technology discourse in the past decade. One of the biggest upsides is the exploitation of economies of scale, which minimizes the cost of operation and thus usage. Further, there is higher capital expenditure required to build a computing architecture locally in comparison to utilizing remote computational resources of a cloud provider. This, in return, reduces the capital risk to transform an application's architecture into a more modern and scalable one, when leveraging cloud services. All of these are mainly motivated by the centralization of computation, which enables more fine-grained utilization of the underlying hardware, with no requirement for always-on services and a pay-as-you-go model [8].

However, a centralized approach of computation holds several drawbacks as well, with latency induced by longer network passes, available bandwidth between multiple services, and further restrictions, like comparatively slow data storage options [9]. In most application scenarios, these drawbacks can be considered as negligible. Nevertheless, there are use cases, which require high data sensitivity, time-efficient processing of large amounts of data, or different environmental conditions, which restrict the usage of cloud services. These environmental conditions include remote areas where stable internet connection and high bandwidth cannot be granted. If there is the requirement for prompt processing of the captured data, like object detection in surveillance videos, or interpretation of sensor data, cloud computing often is not the most suitable option [10]. Perceived latencies, by having to bypass several routing points to reach a data center offering said cloud services, portray a considerable hurdle to overcome for offering cloud capabilities to any device. This effect is most noticeable, the further away the issuing device gets, or the less bandwidth it has available. Thus, bringing the actual computation closer to the source, can alleviate the restrictions by reducing latencies and not having the reliance on highly performant networks [11]. Further, this alleviates many privacy concerns created by sending large amounts of data to remote data centers, which can be tempered or captured by undesired observers [12]. For instance, security cameras would profit from only forwarding detections, by processing the video stream on or close to the device without the need of a remote backend. This would alleviate the need of forwarding the entire video stream over the network, which is often done in practice.

Nascent technologies such as applications for mobile computing and the Internet of Things (IoT) are following this idea by dispersing computation and bringing it closer to the end user. This decentralized approach is being realized by the Edge Computing paradigm. It utilizes large computing and storage resources at the Internet's edge to be leveraged by mobile devices or sensors, which are in a closer proximity than a large cloud provider's data center. These IT infrastructural nodes are referred to as cloudlets, micro data centers, or fog nodes. Satyanarayanan et al. [13] propose that due to the recent attention Edge Computing is gaining from both research and industry, it "promises to deliver highly responsive cloud services for mobile computing".

On top of large, powerful IT infrastructures, there is a large availability of embedded systems. In 2019 IBM estimated the current number of edge devices to 15 billion, predicted a growth to 55 billion in 2022, and 150 billion in 2025 [14]. These enable the so-called "extreme edge", where the data processing happens right next to the sensor, on the same device. This architecture is attractive for applications that process raw sensor data, as the necessity of sending data over the network is no longer a requirement, which omits latency, bandwidth and privacy concerns. Especially running machine learning models on the edge is a highly desired task, as executing the

prediction task on the device would solely require the communication of results, if the embedded system is part of a bigger ecosystem [11].

Therefore deploying tasks out of the machine learning workflow to the edge, called Edge Intelligence (EI), is a new and upcoming research topic. It leverages small resource-constrained embedded devices, which mostly run on batteries. Thus, it is required to optimize the task around computation and energy efficiency. As embedded devices are rarely working as a standalone, they are often part of a larger ecosystem of devices. They mostly just play a small part in a larger application's architecture. Thus, special attention needs to be given to the architectural design as well, as it gives room for many possible improvements.

Environmental monitoring for instance often produces large amounts of data by using embedded devices equipped with sensors to survey an area of interest. These are often required to be interconnected with other systems that give their observations necessary context. Therefore, the SWAIN project [15] is concerned with providing a sustainable watershed management system through IoT-Driven Artificial Intelligence (AI). It deploys embedded systems near waterways in a rural area to run prediction tasks, so-called inferences, on sensor data taken from the river's water. It seeks to predict possible pollution above a certain level by micropollutants from industrial waste, such that an early warning can be issued to the responsible authorities. Therefore, it is essential to consider sequences of measurements to determine the current pollution level. The rural deployment environments are highly constrained by the absence of constant power supplies, and low availability of stable network connections. This limits the types of deployment devices, which can be used, as well as the choice of application architectures. Thus, low end resource-constrained devices are being utilized.

For their cause, specialized Graph Neural Networks (GNNs) are being developed, which offer accurate predictions according to the spatio-temporal data measured by the sensors. GNNs are rising in popularity, as graphs possess a high expressive power in many fields. Environmental contexts often include interconnected elements that can be modeled extremely well using graphs, such as waterways, or road networks, due to their inherent real-world structure. Additionally, they translate well to the biological areas of living organisms and medicine as well, as cellular, and chemical structures are graph-like. [16]

Currently, GNNs variants often suffer from too high computational intensity to be used in embedded environments. As the embedded systems are energy, computation and memory restricted, the deployed neural networks need to be optimized accordingly. Therefore, a novel pruning approach is being developed, which reduces the number of feature dimensions considered during prediction tasks, thus lowering the total amount of computation needed, energy consumed and memory required. This pruning technique is one of the first ones focusing on removing hidden feature dimensions in intermediary layers for prediction tasks, which has been shown to have a high speedup factor with low reduction in total accuracy [6]. This former work focused on identifying less relevant features by applying Lasso regression to identify hardly necessary feature dimensions on the output activation. This work aims to extend this work by evaluating the effectiveness of the linear regression used, comparing it to different variants, and transferring the idea to the spatio-temporal realm.

As it is of utmost importance when heavily pruning a network to remove the right feature dimensions to not influence the prediction accuracies in a strong manner and retain accuracies, we will focus on evaluating the effectiveness of different approaches according to their predictive performance.

1.1 Statement of the Problem

Edge Computing is on the rise, and neural networks are highly attractive to be deployed close to the edge. This decreases data offloading, privacy and bandwidth concerns. Therefore, neural networks need to be optimized to be executed in resource-constrained environments. GNNs often require large computational resources, and are difficult to be used in an edge setting, although they possess great predictive power for many classification tasks. Currently, there are few optimization procedures aimed at GNNs, and some existing ones have not matured enough. Besides different algorithmic acceleration methods, pruning is a promising approach, which does not require many preconditions to be applicable for a network. Thus, this work focuses on analyzing existing problems in the channel pruning approach found in [6], which is based on [17], a technique to reduce computational intensity by removing hidden feature dimensions from the network. Especially, in low-powered environments, extensive pruning is required to make GNN inference feasible, thus improving the detection of less relevant feature dimensions will result in better performing networks that are heavily compressed.

Further, the proposed technique in [6] has not been applied to spatio-temporal models, as of our best knowledge. Therefore, this work adapts the pruning approach and shows its effectiveness on these kinds of models as well.

1.2 Research Question

Currently, there is little research done according to accelerating GNN inference by reducing computational complexity. This is most important for resource-constrained environments, where no constant supply of energy nor high computational power can be assumed. Existing research focuses on reducing some computational complexity, but is highly reliant on presumptions. Therefore, a technique needs to be found, which can reliably reduce the computational intensity, as well as memory requirement, for GNN inference, such that it can run in resource-constrained environments.

A promising algorithmic acceleration technique is proposed in [6], which uses channel pruning to reduce the dimensionality of hidden feature vectors of neural networks. It uses Lasso regression to identify less relevant feature dimensions on the output activation. Lasso regression is not known to perform well in high dimensional datasets, with little highly correlated features. Therefore, we want to analyze the quality of the utilized linear regression technique. Further, we want to apply the technique to the spatio-temporal models used in the SWAIN context, to facilitate their usage on low-end devices close to the waterways to be investigated.

The resulting research question of this work reads: "Is Lasso regression outperformed by Ridge or Elastic Net regression, when used to identify less relevant feature dimensions of graph neural networks during channel pruning? And can the technique be applied to spatio-temporal graph neural networks such that they can be used to perform predictions in resource-constrained environments with a tolerable loss of accuracy?"

If the proposed research shows generalizable and successful results, it will enable the realm of Edge Computing and Edge Intelligence to use a variety of GNNs in resource-constrained environments. This would enable the architectural design of sensor networks, and their possibility to perform predictions close to the data source.

1.3 State of Research

Compared to different types of neural networks, GNNs are computationally more expensive, as [1] displays. They naturally come with a high training and inference intensity, but with the upside of delivering the best performance in graph prediction tasks. Inference describes the process of running a prediction task with a neural network. One of the most powerful properties of GNNs is the ability to work with massive graphs of variable sizes. Thereby, they can process large amounts of information during training and inference. Due to the effect of large training data and slow convergence times induced by an overabundance of features and the various considered nodes, a GNN requires comparatively long training times in respect to other approaches. In this regard, a significant amount of research has already been made with approaches that can even reduce the training times of large datasets to a couple of minutes.

Currently, the idea of making GNNs perform well on resource-constrained devices, with their high need for computation and memory, has not been investigated enough. Therefore, a way to reduce computational intensity and therefore accelerate the networks while performing a prediction needs to be found. Besides hardware acceleration, there are different algorithmic acceleration approaches to draw from. One group of them aims to reduce the number of components to consider during inference. This can be done in a number of ways. Currently, there is no clear term for considering less structural components during GNN inference computation. There are different research efforts that aim to achieve similar goals, but with different methodologies or quality criteria in sight.

Firstly, there is graph sampling, which samples training graphs before training, therefore reducing the number of nodes inside a training graph. Applying this technique comes with a reduced chance of overfitting, faster convergence and more resilience to noise during training. Additionally, the resulting GNN will end up being sparsified as well. Thus, it can be interpreted as a pruned version of a converse network, which would have been trained with a non-sampled version of the training graphs. An example of these approaches is GraphSAGE, proposed in [18], which randomly samples neighborhoods of a starting node to be used in feature training. The result is a sparsified GNN, which reduces the number of nodes to consider during inference, therefore reducing its computational complexity. In comparison to our proposed method, graph sampling mostly aims at different quality criteria like reduction of overfitting, higher accuracies or noise resilience, rather than smaller neural networks with less computational intensities. Additionally, achieved levels of pruning occur rather random, and cannot be guaranteed nor expected to reduce the computational intensity by high enough levels.

Secondly, there are graph sparsification approaches to apply heuristics to sparsify training graphs. DropEdge [19] randomly removes edges of training graphs to solve over-smoothing issues by too similar nodes in proximity. This gives too much weight to similar nodes in associative graphs, where akin nodes are stored next to each other and regularly share an edge. They prove the effectiveness of their technique in Graph Attention Networks (GATs), where a number of attention heads perform the so-called attention mechanism, with relevance coefficients being computed for all neighbors. They show that the standard deviation between attention head coefficients is minimal in most scenarios. Thus, by removing edges randomly, there is a high chance that the dropped edge had a small relevance. This encourages the network to learn a more robust and generalized representation of the targeted graphs. Additionally, they show the effectiveness of this approach in Graph Convolutional Networks (GCNs) as well. As "DropEdge can be considered as a data augmentation technique" [19], it does not focus on computational reduction, even though it complements it as well. By reducing the number of edges, it diminishes the feature aggregations, also called message passing, in GNNs. When comparing different pruning approaches, dropping weights is less effective than removing entire nodes, layers or feature dimensions, as singular

edges are less often considered during computation of feature embeddings. Thus, even though the proposed technique can help during training, it is less effective for optimizing inference tasks.

Another graph sparsification technique is FastGAT (Fast Graph Attention Networks, proposed by Gao et al. in 2019 [20]. Here, the main concept is to accelerate computation of attention coefficients in GATs by using a truncated softmax function. A softmax function is often used to produce the final class labels in a multi classification task, such as labeling a subset of nodes inside a graph. In GATs it is used to normalize the attention coefficients. FastGAT then takes this truncated softmax function to find the top-k most similar nodes to compute attention weights for the next step. This corresponds to a weight pruning technique, in the sense, that fewer connections to different nodes are considered. The method is applied during training and inference. As with many other techniques, this adds an overhead to the prediction step, as the softmax function needs to be truncated according to the applied heuristic. So, the potential speed-up is relative to the size of the graphs used during inference, and the additional induced overhead by the truncation. The technique discussed in this work reduces the size of the network and hidden feature dimension before performing inferences, thus does not induce additional overhead during inference and should be able to outperform these approaches.

Another technique is the unified GNN sparsification UGS, proposed by Chen et al. [21] in 2021, which suggests that GNNs contain identifiable subnetworks, so-called "lottery tickets", which can be trained in isolation and extracted from their larger context. The obtained sparse subnetworks perform similarly to a full network in their classification task. Experimental results have proven that UGS achieves significant parameter reductions with little to no loss in performance. The obtained models portray a good fit for resource-constrained environments, but largely depend on having large training graphs with identifiable lottery tickets to be able to utilize properly. The approach highlighted in the later section is more generic, as it can be applied to any setting with an expected increase in performance and reduction in size of the network, irrespective of preconditions.

Further ways of optimizing GNN inference are graph simplification approaches with LightGCN [22] and UltraGCN [23]. He et al. [22] find that feature transformation, using weight matrices, and nonlinear activation function hardly benefit the collaborative filtering approach used in many GCN architectures. The simplified LightGCN architecture consequently drops these and performs simple matrix multiplications on propagated node representations. This results in a more scalable and efficient architecture, which can be used to train on large graphs with millions of nodes and edges. It has shown good performances on recommendation datasets, and outperforms other popular architectures like GraphSAGE and GAT.

UltraGCN, proposed in 2020 by Zhang et al. [23], is designed to address issues of oversmoothing in GCNs. It builds on top of the work of LightGCN by enhancing their utilized message passing with a simpler structure [1]. Therefore, it uses a pooling operation that aggregates local information of neighbors. This can be done at multiple levels of graph coarsening, such that the model can capture information at different depths of the graph. Additionally, residual connections are used to update the node representations by taking a linear combination of their neighbors, with weights determined by the adjacency matrix. UltraGCN has shown similar or better results to LightGCN and can be regarded as an extension. Both approaches focus on accelerating the training of a network, not the inference tasks, but end up with a smaller and often faster neural network. The performance of it is mostly task related and is not expected to perform well in a spatio-temporal setting.

Knowledge distillation concerns itself with training small student models based on larger teacher models, which should achieve similar performance. This is done by using the prediction labels of a teacher model together with the original training graph to train the student model. TinyGNN, proposed by Yan et al. in 2020 [24], helps with extracting the neighborhood information between

a teacher and a student model by combining a peer aware module and a neighbor distillation strategy. Further, it introduces a novel attention mechanism, which allows the student model to capture long-range dependencies between nodes. These are being extended by a residual connection to the GCN layers. This allows for capturing non-linear relationships between nodes and retention of information from previous layers. Knowledge distillation strategies can result in significantly more efficient and compact GNN models, but are highly dependent on the utilized datasets for training. Additionally, existing implementations are mostly hard-wired to a certain kind of student model, as the information retention is tough to generalize between different styles of GNN models. Consequently, the applicability and extension to different problem domains is a limiting factor for current approaches.

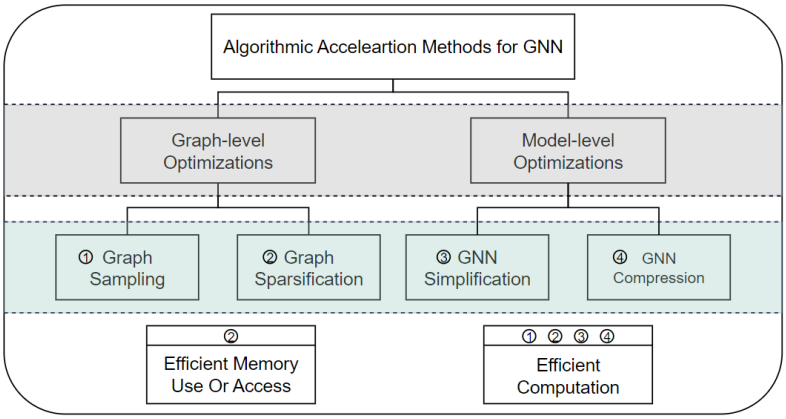


Figure 1.1: Adapted version of the proposed taxonomy for algorithmic GNN acceleration methods by [1].

Lastly, there is "Accelerating Large Scale Real-Time GNN Inference using Channel Pruning" (GCNP), introduced in 2021 by Zhou et al. [6]. They utilize a channel pruning technique to reduce the number of feature dimensions considered during inference. Therefore, they came up with a novel Lasso regression formulation, a linear regression technique, which is often used in machine learning to perform feature selection. It is better known as the L1-loss function and is often applied in scenarios where the amount of training data is low, but the number of feature dimensions is high. It computes relevance coefficients of the feature dimensions, which are reduced up to zero, if their influence on the output activation is slim. This effectively removes the feature dimension from the model, thus sparsifying it. The Lasso regression converges when the changes in coefficients become smaller than a certain threshold. The main critique to this approach are the limitations of the utilized Lasso regression. These get even more severe for a network that requires a high rate of compression. Additionally, existing implementations of the approach need to be cleaned of errors, and transferred to different GNN architecture, to approve the effectiveness of the approach for different use cases.

Therefore, our proposed technique uses Ridge and Elastic Net regression, which are linear regression techniques as well. In comparison, to the reduction of coefficients by absolute values, they add a penalty term to encourage a model to minimize the squared values of the coefficients. This leads to coefficients which are close to zero, but not exactly zero. These techniques use the so-called L2 regularization, which is most useful in settings where many features are not highly correlated. By having small coefficients, we can prune feature dimensions which are either below

Method	Work
Graph Sampling	GraphSAGE[18],
Graph Sparsification	DropEDGE[19], FastGAT[20], UGS[21], GCNP[6]
GNN simplification	UGS[21], LightGCN[22], UltraGCN[23]
GNN compression	TinyGNN[24]

Table 1.1: The related work classified according to the adapted version of the taxonomy proposed in [1].

a certain threshold, or a certain percentage of feature dimensions. This makes our approach more flexible than the former one and can be better utilized in resource-constrained environments. Additionally, the different linear regression techniques promise to perform better when higher levels of compression are required.

To classify existing algorithmic acceleration methods aimed at GNN [1] introduces a taxonomy to differentiate between different approaches, it can be seen in figure 1.1. It hierarchically makes a distinction between graph-level and model-level optimizations. These are then further split up into graph sampling, sparsification and partitioning, and GNN simplification, and compression respectively. The aforementioned related techniques can be found in table 1.1 according to the proposed taxonomy. Our proposed work can be classified to the "Graph Sparsification" as well. Apart from the aforementioned techniques, this taxonomy also mentions "Graph Partitioning", which describes splitting a model between two layers, such that the inference task can be processed on two devices instead of one. As it includes multiple devices, we do not consider it as relevant to the present topic.

1.4 Description of the Remaining Chapters

The following chapter 2 gives a theoretical foundation for the work, as it includes topics about neural networks, Edge Intelligence (EI), pruning, and linear regression. The introduced topics and concepts form the basis for understanding the utilized techniques in the experiments of this work. The methodology as well as the conceptual contributions of this work are explained in chapter 3. The experimental setup can be found in chapter 4, which is used to run the benchmarks that are interpreted and discussed in chapter 5. Lastly, the findings of the work are being summarized in chapter 6.

2 Theoretical Foundation

2.1 Neural Networks

A neural network is a type of machine learning model that is designed to solve tasks that typically require a human brain to process the information, such as speech recognition, decision-making, or object detection. It is composed of a set of interconnected layers containing nodes, so-called "neurons", that process their input by applying simple mathematical operations to them. The results are then forwarded along weighted edges to connected nodes, forming a computational graph. These weights are typically learned by iteratively refining the obtained results a network provides. When a neuron processes information, it forwards its results to all connected nodes with weights greater than zero. This is not done per node, but with matrix multiplications, such that the weights of an individual node are contained inside a weight matrix and the nodes themselves are present inside an adjacency matrix. Before a final prediction is made, the outputs often need to be forwarded through an activation function to produce an interpretable result.

As neural networks normally learn higher level information from a dataset of already labeled data, they are often used for supervised learning tasks. Their inputs are typically some form of numerical data, like the features of a dataset or the pixel values of an image. These are then ingested into the network and the final prediction is being compared with the expected result. This observation is then used to readjust internal weights of the network. This process is repeated until the chances in the results converge or do not change much between the iterations.

The outputs of a neural network depend on the task it is designed to perform. Most of them are trained to solve classification tasks, which should output either a single or multiple labels. Therefore, the readout layer of a network often provides a probability distribution over a set of known classes. There is a wide range of different tasks that may output a language translation, or sentiment analysis of an input sentence as well.

The architecture of a neural network refers to the structure of the network, including the number of layers, the number of neurons in each layer, and the connections between the neurons. There are many different architectures of neural networks, each designed for different tasks or data types they should work with.

2.1.1 Types of Neural Networks

One of the most common classes of neural network architectures are feedforward neural networks. They process information by passing it through the network in one direction, from the input layer to the output layer. Each layer performs a set of mathematical operations on its inputs and passes the result to the next layer. Feedforward neural networks are used for a wide range of tasks such as classification and regression.

One of their representatives are Convolutional Neural Networks (CNNs), which were originally designed for processing images, but can be adapted for other data types as well. They use convolutional layers that apply a set of transformations to the input image to extract higher level information of different spatial scales. Therefore, a local neuron obtains enriched information from its spatially connected nodes that already aggregated their information from their neighbors. The outputs of the convolutional layers are then passed through one or more fully connected

layers to produce the final output. For instance, in image detection, one wants to relate the color of a pixel to its neighbors to find connected objects to run predictions on. This information then gets aggregated and some form of averaging is performed to enrich the pixel with higher level information, like an average color.

Recurrent Neural Networks (RNNs) are a specific type of neural network that are designed to handle sequences of data, such as text or speech, where the order of the processed information is essential. They often use so-called feedback connections to allow information to be retained between steps of the data sequence. Modern versions such as [3] use layers of Long Short-Term Memory (LSTM) cells, displayed in figure 2.1, in their neural units. Their purpose is to remember certain parts of the information of prior seen data. Therefore, they contain different kinds of gates to handle how information should be retained. These gates enrich the input data, guide the remembering process of the memory cell itself and when to forget information, and enrich the output as well. A LSTM cell is in itself a small neural network with encoding, processing and decoding parts.

Conversely, the tsl [25] library utilized in the SWAIN project revolves around handling spatio-temporal data, which inherently benefits from the aforementioned concept of remembering certain information about prior data measurements. Therefore, it offers different variants of GNN, such as gated GNNs and an adapted version of Graph Attention Network (GAT), which will be highlighted in section 2.1.2.2.

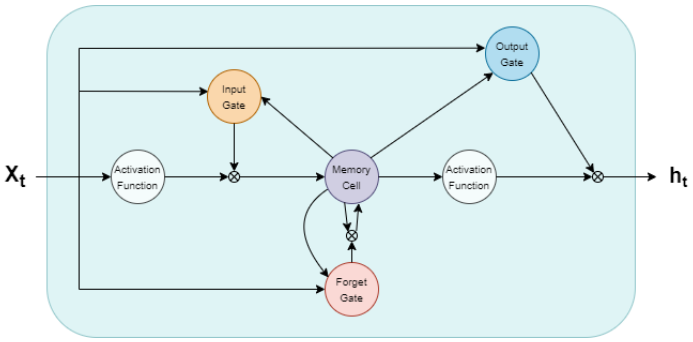


Figure 2.1: Exemplary depiction of a long short-term memory cell with input, output and forget gates, activation functions and a memory cell itself. X_t represents the adjacency matrix at time point t, and h_t the new hidden representation of the input at timestep t. Figure adapted from [3].

Training a neural network involves adjusting the weights of the edges between the neurons such that the network produces the desired output for a provided input. This is typically done via backpropagation, a process that involves computing the gradient of a loss function computing the goodness of a model's fit. The weights inside the neural network are then adjusted such that the loss reduces, therefore representing a better predictive ability for the model.

The loss function measures the difference between the produced output of the neural network and the desired one for a given input. There is a wide variety of loss functions that can be used to achieve distinct goals. Therefore, they use different evaluation criteria inside to determine the current accuracy of a model. For example, a common loss function for classification tasks is the cross-entropy loss, whereas a common loss function for regression tasks is the MSE.

Neural networks are powerful machine learning models that have been successfully applied to a wide range of tasks, including image and speech recognition, natural language processing,

and recommendation systems. However, they can be computationally expensive to train and require large amounts of data to achieve good performance. Additionally, the process of designing and training a neural network can be complex and time-consuming, requiring expertise in both machine learning and computer science.

The following sections introduce different variants of Graph Neural Networks (GNNs) as this work focuses on developing a pruning technique aimed at those.

2.1.2 Graph Neural Networks

Among existing machine learning methods, deep learning has advanced as one of the premier solutions regarding accuracy by leveraging deep representations of the data in the form of Deep Neural Network (DNN). They typically consist of several layers of nodes, or neurons, connected by weights expressing a deep representation of the data.

GNNs are a special type of DNN that focuses on graph prediction tasks. A big upside of using graphs in applications is the high expressive power and generalizability they provide. Thus, GNNs attracted lots of attention due to the higher utilization of graph structures in data mining tasks, like node classification, edge prediction, and graph classification. Although there are different neural networks able to solve these assignments, the node embeddings generated from GNNs outperform them greatly. In comparison to most other DNN variants, some GNNs possess the ability to process data of variable input length. This is not only beneficial as it allows reusing an existing model when the input data changes, but is also helpful because many graph datasets contain graphs of differing lengths.

The goal of running inference tasks on GNNs is to find the most similarities between the input and the trained contextualized representation of the dataset. Thus, it is required to obtain similarity functions, which can handle graph structures. These process different parts of the graph, like single nodes, sub-graphs or even entire ones, to perform their task. Performing a similarity between every node pair is extremely inefficient. Thus, lots of research focuses on finding less computationally complex ways of computing these.

As graphs are irregularly shaped, it is required to compute a node embedding of a graph, which captures the structural and feature information on a single node, sub-graphs, or an entire graph. Once the local information is preserved, several aggregation steps are performed. These are often handled by smaller neural networks, which perform more sophisticated operations. The aggregation steps take in some feature vectors of a graph, aggregate their information and relation, and pass the obtained result to the next layer. Before an output can be generated, the aggregated node embedding will then be passed through an output layer, which performs a final transformation on the result to obtain a classification result. A standardized workflow of a GNN can be seen in figure 2.2.

These are the basic inner-workings of all GNNs, as they draw higher level information from their position in their respective neighborhood. Thus, during one entire propagation phase in a GNN it is required to perform information aggregation on the complete or a subset of neighbors, followed by the node embedding computation, and lastly a propagation stage, where the newly obtained information is passed on to the neighbors. As graphs can grow to a large number of nodes, and this process is often done for most nodes, performing a forward pass on a graph is highly computationally intensive. This leads to most GNNs being restricted to only a few layers inside their structure to reduce the computational effort, as every additional layer will exponentially increase the number of operations. For instance, in the first layer every node considers all neighbors in their 1-step proximity, thus the direct neighbors. The second layer then adds context about all neighbors who are reachable by two steps. It is most likely that the set of 2-step neighbors is greater than the set of all 1-step neighbors. This behavior remains when adding

more layers, which leads to a rapid increase in neighbors to consider, the so-called neighborhood explosion. To tackle this problem, several techniques have been developed to circumvent this issue, by for instance increasing the sparsity in the graphs and thus model, which leads to a far smaller set of nodes to consider [18] [26].

Other work focuses on the reinterpretation of GATs, a variant of GNNs, from the Euclidean to the Hilbert space, which the researchers in [27] show to be more aligned with the internal procedures of GATs. This comes due to the computational tree generated by the attention mechanism used within the approach, fits better to the Hilbert space, than the Euclidean one. This, in return, allows for mathematical reformulation of forward propagation inside the network. They then show that this problem is capable of being optimized and largely reduces the number of considered neighbors. Therefore, applying this technique to GATs might even lead to the elimination of neighborhood explosions completely.

Nevertheless, neighborhood explosion is still a major hurdle for GNNs and often results in current networks being rather shallow.

Many applications that utilize GNNs are latency sensitive at inference, for instance computer vision tasks or fraud and spam detection in natural language processing. Additionally, there is the domain of inferences running on edge devices with limited compute power and memory, like self-driving cars equipped with radars, cameras, and GPS systems that continuously produce data required for time sensitive decision-making. Thus, there is the requirement of accelerating inference times in GNNs by reducing their computational intensity at inference time. As performing the full forward pass during inference leads to high memory usage and latency, batched inference can be used, where only a selection of nodes is targeted instead of an entire graph. Whereas the model training itself often just takes a matter of minutes on graphs with millions of nodes, due to the use of powerful hardware and different optimization techniques like GraphNorm [28] or stochastic node sampling.

GNNs are not only well suited for performing graph related prediction tasks, but are also of interest due to their input being graphs. In the SWAIN project, a sensor network is being placed near the waterways. An intuitive data structure to utilize in this setting is a graph structure that represents the placements of the sensors along the river.

These measurements could then be transmitted using a low-powered local area network (LPWAN) to a series of edge servers to perform the inference tasks, which will then forward the results to a remote cloud instance. This not only allows for consistent communication of measurements, but also alleviates privacy concerns by processing the data at the edge and only forwarding prediction results to a cloud instance.

In the following section, popular variants of GNNs are being highlighted.

2.1.2.1 Graph Convolutional Network

DNNs have grown largely in popularity due to their wide range of applications. One of its most widespread representatives are CNNs. They perform convolutional operations on spatial information of ingested data to extract higher level information. Thus, they perform best in settings where data points are closely related to their spatial neighbors, for instance in image and video analysis.

The basic building block of a CNN is a convolutional layer. In these layers, a set of filters are applied to the input, each filter being a small matrix of weights. Each filter is then convolved across the input, generating a new output feature map, typically according to the direct neighbors of each element. The weights in the filter matrix are learned during the training process through backpropagation of error gradients. These can then be stacked upon each other to consider further information from an original data entry, thus creating a so-called n-hop neighborhood.

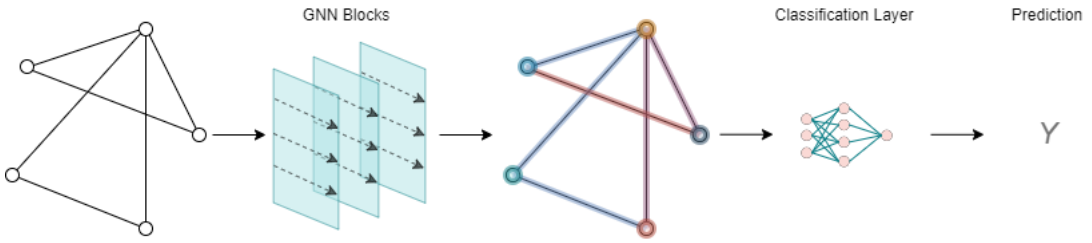


Figure 2.2: An exemplary workflow of standard GNN. First, a graph is inserted into a set of GNN blocks or layers, which transform it into an embedded graph representation carrying enriched spatial information. It is then ingested into a classification layer with different activation functions to form a prediction. This can be in the form of a single classification of a node of interest, a class label for all nodes, or a classification for the entire graph.

As it is computationally costly to perform a forward pass in GNNs, when considering every other node inside the graph, most variants focus on spatially close neighbors to provide essential information. Most graphs are constructed in a way, such that edges express relations between nodes. These either express similarities or dissimilarities, therefore contain crucial information about the node state. Hence, an important assumption about the training graphs must be made, that they are associative, such that relevant similarity or dissimilarity can be determined from the neighbors. This motivated the transfer of performing convolutions on graph nodes inside a GNN as well. These networks are called Graph Convolutional Networks (GCNs).

Corresponding to CNNs, an input layer is used to transform the graph into a hidden representation. Afterwards, every following convolutional layer first averages all the neighbors of a given node, thus computing a node embedding. Then, it multiplies the node embedding with a trainable bias matrix, and ensures non-linear activation by multiplying it with a given coefficient. These layers are typically followed by pooling layers, which downsample the output feature maps, reducing their spatial dimensions while retaining most of the information. Lastly, the output gets passed through an activation function, which typically introduces non-linearity to the network and enables it to transform the results into a complex pattern. This output is then flattened into a 1-dimensional vector and is then passed into a classifier for the final prediction.

2.1.2.2 Graph Attention Network

GATs are one of the most compelling options for solving graph prediction tasks. They excel in selectively focusing on parts of the input data by using the so-called attention mechanism [29]. It allows the network to aggregate information about neighboring nodes in a more controlled manner, as it only considers a number of most important edges according to their learned relevance. The propagation in GATs is subdivided in sequential steps by first transforming the local feature vector in a lower-dimensional representation. Then the attention coefficients are computed according to a node's neighboring nodes and some compatibility function. These weights are then taken to perform feature aggregation on the information of determined relevant neighbors to create a context-aware representation for the central node. Lastly, this representation is used in various downstream tasks, such as node classification or link prediction, by applying a softmax function to it to find the most probable label of the current prediction result.

In the following section, the attention mechanism will be highlighted in more detail and

put into mathematical terms. The goal is to compute an attention vector that represents the relative importance of each element in the input data. First, a linear transformation needs to be applied to the input graph to transform the input features into a higher-level representation. This linear transformation is parameterized by a learnable weight matrix $W \in R^{F \times F}$ with F being the number of feature dimensions. This is then multiplied with the a given input vector $h = \left\{\vec{h_1}, \vec{h_2}, ..., \vec{h_N}\right\}, h_i \in R^F$ [29]. Afterwards, the self attention mechanism can be applied to the result such that $a : R^F \times R^F \rightarrow R$ computes the attention coefficients.

The formula to compute resulting node embeddings is then given by:

$$e_{ij} = a(W_{hi}, W_{hj})[29] \tag{2.1}$$

The final attention vector is then computed by using a softmax function, which normalizes the weights. It represents a probability distribution that sums to one. This is done to normalize the results and put them into an interpretable form. The equation for computing the normalized attention coefficients is therefore given by:

$$\alpha_{ij} = softmax_j(e_{ij}) = \frac{exp(e_{ij})}{\sum_{k \in N_i} exp(e_{ik})}[29] \tag{2.2}$$

This reduces overfitting as well, as high values are being scaled down. Furthermore, it ensures that the model pays attention to all parts of the input, but assigns more weight to the parts that are most relevant to the current prediction. Therefore, it often discards many edges from consideration as well, such that only selected important nodes remain. The total number of nodes to consider is defined by the number of so-called attention heads to be used inside the attention mechanism.

Traditional GNNs perform a fixed, uniform aggregation of information from neighboring nodes. In contrast, GATs allow for each node to learn an adaptive weighting scheme for aggregating information from its neighbors.

The final output of the GAT is a node embedding for each node in the graph. This embedding can be used in a variety of ways, such as node classification, link prediction, or even graph clustering.

One of the main benefits of GATs is that they can handle graphs of varying sizes and structures. Unlike traditional GNNs, which require a fixed-size adjacency matrix as input, GATs can operate on graphs with varying numbers of nodes and edges. Further, GATs can handle directed and weighted edges as well, and even are known to work on graphs with multiple different types of edges inside.

Another benefit of GATs is that they can learn node embeddings that capture both local and global information. The attention mechanism allows each node to selectively attend to its neighbors, which enables the network to learn representations that capture the local structure of the graph. At the same time, the multi-head attention mechanism allows the network to attend to multiple aspects of the graph simultaneously, which enables it to capture more global information. Further, due to its non-uniform neighborhood aggregation mechanism, it can handle graphs of varying sizes and structures.

One drawback is the high computational need for computing extensive feature vectors, which are appended at every iteration and layer depth. Therefore, Rong Yu et al. state in [19] that the variance in computed attention head coefficients along edges is often negligibly small. Thus, it is feasible to remove some at random during training and inference, with little loss of accuracy. This further motivates the pruning before training of the GAT, as most parts are proven to be non-integral for the final output activation. Thus, it should be possible to reach a high level of compression together with a subsequent reduction in computational intensity in GATs.

In the SWAIN project, a variant of GATs is used to estimate influences of rainfall on measurements. Therefore, the project can benefit from being able to deploy the network on low-end devices close to the waterways if the utilized models are optimized well enough.

2.2 Edge Intelligence (EI)

With breakthroughs in deep learning, the boom of IoT devices, and mobile applications, there are now about 55 billion devices [14] generating enormous amounts of data at the network's edge. This provoked a push for AI to the network's edge to unleash the potential of edge big data. After the surge of edge computing, with its decentralization of computation from the network's core to the edge, there is now a range of new and upcoming research domains [4]. One of them is EI, which has garnered lots of interest from both research and industry because of its high applicability, low upfront cost, and possibility to reduce cost, when swapping from a centralized architecture running machine learning tasks [30].

AI and especially the widely popular deep learning, have made significant breakthroughs in numerous fields over the last decade, spanning computer vision, speech recognition, natural language processing and many more. Hence, many human assistance programs, i.e. virtual assistant technologies, recommenders, video surveillance or smart home appliances, have emerged accordingly.

Typically, the data is being captured and progressed on the same device, with just partial communication to a data center. Formerly, big data was mainly born and stored at large-scale data centers. Now, Cisco estimates that nearly 850 ZB will be generated by all people, machines, and things at the network edge by 2021 [31]. In comparison, the estimated total amount of global data center traffic was only 20.6 ZB by 2021 [4]. So, there is an insurmountable problem in bringing the current volume of data from the edge to the cloud to be processed. Similarly, Gartner states that "around 10% of enterprise-generated data is created and processed outside a traditional centralized data center or cloud. By 2025, Gartner predicts this figure will reach 75%" [32]. So, application scenarios that consume data on the edge will only rise in need over the next couple of years as this decentralization trend continues.

Yet, currently there is a lack of performance and energy efficiency when running standard machine learning models on IoT devices, which outweighs the reduction in network latency and lower security concerns, as less raw data needs to be transferred over the network. Because application scenarios greatly vary, there are different technologies that move the computation closer to the edge, ranging from embedded systems the size of a credit card to powerful micro data centers. The gained physical proximity "promises several benefits compared to the traditional cloud-based computing paradigm, including low-latency, energy-efficiency, privacy protection, reduced bandwidth consumption, on-premises and context-awareness" [4].

As standard deep learning models are often too constrained to be well suited for EI, the following sections will introduce concepts and methodologies that concern themselves with these issues. The subsequent sections will then cover current architectures, enabling technologies, systems, and frameworks for training efficient and applicable EI models.

2.2.1 EI: Levels

Zhou et al. state that "EI should not be restricted to running AI models solely on the edge server or device. [...] Running (DNN models) with edge-cloud synergy can reduce both the end-to-end latency and energy consumption" [4]. They rate EI into six levels, as shown in figure 2.2.1. They range from doing inference and training on the device to doing both on the cloud.

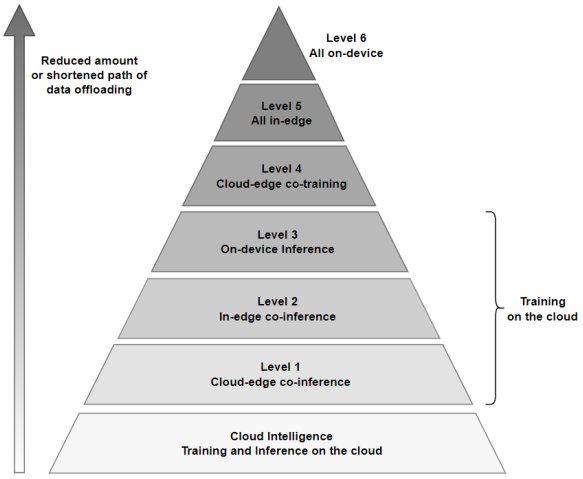


Figure 2.3: Levels of Edge Intelligence. Adapted from [4]

The SWAIN project's setting can be placed on the levels 1-3, where training is being done on the cloud, but inference can be shared between the edge and the cloud.

In general, there are multiple different architectural styles to choose from, which are advantageous for different settings under different preconditions. This work assumes no presence of stable internet connection and no availability of a constant power source. Thus, it is focusing on a setting where inferences are run completely on the end device. So, we attempt to make it feasible that a trained model can be executed on it for a long enough time frame while keeping in mind memory and computational constraints.

2.2.2 EI: Model Training

Model training is an essential task in the machine learning workflow. Not only does the initial model require to be trained with a set of training data, but it often needs to be retrained if a significant deviation between the original training data and current data distributions can be observed. Thus, training, or retraining, machine learning models close to or on the device itself is beneficial.

Hence, facilitating distributed training of DNNs should be aimed for. This can either be done centralized, decentralized or hybrid. When aiming at optimizing model training for edge devices, one needs to be aware of four key performance indicators. Namely: privacy, communication cost, latency, and energy efficiency. There are many compelling approaches, enabling technologies and architectures listed in [4]. As these are not inside the scope of this thesis, they will not be further elaborated.

Due to changes in model structure, architecture and parts, algorithmic acceleration techniques often already have to be considered during model training. Thus, there is an overlap between inference and training optimization techniques. For instance, there are training techniques aimed at producing sparsified networks, thus reducing their size and computational intensity. These techniques are often limiting due to the fact, that they are highly data-dependent and not agnostic to the type of network one wants to produce. Therefore, we focus on an acceleration technique, which is happening post training.

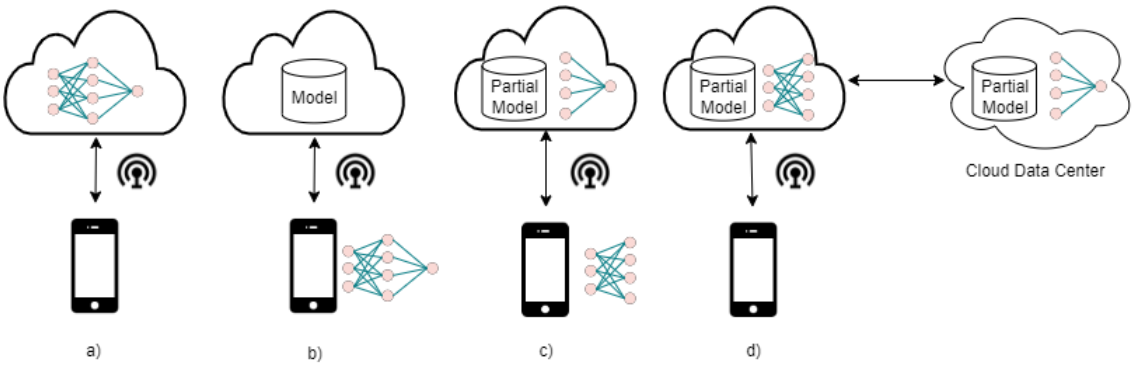


Figure 2.4: Modes of edge inference for differently deployed DNN models. a) shows an edge-based mode, b) and c) an edge-device co-inference mode, and d) an edge-cloud mode [4]. The illustration has been adapted from [4].

2.2.3 EI: Model Inference

To enable high-quality EI service deployment, optimizing model predictions is of most critical importance. The quality of AI models is not only reliant on the accuracy of the prediction, but on the timeliness of delivered results. Consequently, there are several different approaches and architecture to choose from, when deploying a machine learning model in an edge setting [33]. Therefore, Zhou et al. [4] differentiate between four different DNN model inference modes, namely, edge-based, edge-device, and edge-cloud mode. These can be observed in figure 2.4.

Edge-based mode is running the inference completely on an edge server while receiving the input data from a mobile device and returning the prediction results. The main disadvantage of this mode is the reliance on good network bandwidth between the devices.

Device-based mode is running the inference on the device, while maintaining the model on an edge server. Thus, communication during the inference is not required. Although this makes the inference reliable, it consumes high computational resources and energy.

Edge-device mode is based on sharing a partial model between edge computing resources and the issuing mobile device, thus reducing the total required number of computations a device has to handle. Then, an intermediary result is passed to the edge server, which finalizes the prediction result and returns it. The edge server can partition the model in several ways, and make decisions, based on environmental conditions, which parts to communicate with the device. This mode is more flexible than the former two, but also requires high computational power on the edge device, as the first layers of inference are computationally intensive, and a stable connection to the backend.

The last mode is edge-cloud mode, where an edge server is sharing its inference task with a cloud server. A mobile device solely provides the input data for the DNN model and receives the final prediction from the edge server. This mode is preferable when the mobile device is highly resource restricted.

Key performance indicators are latency, accuracy, energy, privacy, communication overhead and memory footprint. These must be traded off in most scenarios to reach a satisfactory model for a given device executing it.

2.2.4 EI: Inference Optimizations

To optimize the model inference in the four modes of EI inference, there are different approaches to take. For instance, there is model compression, which reduces the model complexity to alleviate the resource requirement for an edge device in both storage and computation. This enables on-device inference, by reducing response latencies and privacy concerns. It optimizes latency, energy, privacy, and memory footprint.

Between other techniques, like data quantization and more compact architectures, weight pruning represents the most widely adopted technique [4]. Pruning approaches analyze a pre-trained DNN to remove redundant or insignificant weights. On top, many "dead" neurons can be completely dropped, as they either have no incoming or outgoing edge anymore. There are different techniques as well, which directly focus on neuron pruning, so-called structural pruning techniques.

These approaches trade-off accuracy with network size. This poses the main challenge during optimization. Han et al. [34] attempt to solve this problem by magnitude-based weight pruning. The method is to first remove small weights below a specific magnitude and then reconfigure the model to restore potentially lost accuracy. Afterwards, the same team approached the task with deep compression [35] by combining weight pruning, weight sharing, and Huffman coding. This resulted in a compression of 35-49x. Further, there is VIBNet [36], which advances as the premier structured network pruning techniques. It removes inter-layer redundancies and aggregates the obtained set of connected neurons in a sparsified model.

Energy-constrained devices might not be able to take full advantage of these techniques, as pruning does not necessarily address energy savings. For instance, CNNs typically require most of the computational need for the first convolutional layers instead of the fully connected ones at the end, which hold the most amount of the weights. Nevertheless, model compression by the ways of pruning reduces the number of computational operations, which results in reduced energy consumption as well.

Data quantization is tackling the size reduction by converting the utilized data formats to a lower bit representation, and therefore reducing the memory footprint and potential energy consumption. The selection of data format can be switched between layers. Naturally, switching to a less accurate number representation results in a loss of accuracy. It is important to note that data quantization should only be applied after training, as lower precision does prolong the training time because convergence takes far longer.

The next major category of inference optimization is model partitioning. As shown in the different modes of running inference between a mobile device and the edge, there is the attempt to offload computationally intensive parts to an edge server. Model portioning mainly aims at reduction of latency, energy, and privacy concerns. When partitioning a model between different devices, the Neurosurgeon framework [37] helps with finding proper splitting points to optimize performance, latency, and energy efficiency. It estimates the imposed latency by each layer in the DNN model in a regression-based mode to obtain the optimal splitting points. When applying model splitting for embedded devices in rural settings, one should aim at finding the layers which require the smallest amount of information to be passed in between them to alleviate bandwidth constraints.

Similarly, Ko et al. [38] propose a model partitioning technique to host the obtained parts at the edge. They combine this approach with lossy feature encoding. Thus, the data to be forwarded between the layers at a given splitting point, is being compressed before transmission by using a lossy feature encoding. The JALAD [39] framework takes a similar approach by deploying an integer linear programming optimization problem to minimize model inference latencies [4]. These problems are NP-hard, so Hu et al. [40] approach the topic from a worst-case

performance guarantee standpoint by applying heuristics. Additionally, there are approaches allowing for collaborative DNN model inference by multiple devices utilizing broadcasting in a peer-to-peer network [41]. As a result, depending on a given situation, the problem of model splitting can be approached from different angles, when considering required latency, available devices and required worst-case performance. Regarding the SWAIN project, mainly the JALAD and Neurosurgeon frameworks appear as attractive, as they optimize communication between the devices.

Another model is the early exit strategy. Here a DNN model is being trained with exit gates at certain layers, to allow for the inference process to be stopped, if the current prediction probability at an exit gate is below a threshold. Early exit strategies need to be considered during training, as they are an own structural component in the network. BranchyNet [42] for instance, trains a DNN model by enabling different classification points for the data at intermediary layers. Different approaches, like distributed DNNs [43] use a similar idea in a distributed architecture, where inference on a mobile device, edge server or cloud server all portray one layer. At each exit point of these layers, there is a possibility to exit the inference process. The early exit training is based on BranchyNet. The final predictions need to be aggregated at a central point. Edgent [44] is proposing a joint approach between early exit and model partition by attempting to maximize the accuracy under a given latency constraint. This is done by employing a regression-based layer latency prediction model. Further, Lo et al. [45] extend the BranchyNet model by extending it with an authentic operation unit to determine if an input should be forwarded to an edge server or cloud data center for further investigation. Therefore, different confidence levels of the DNN model's classification classes are compared to defined thresholds.

Edge caching is an attempt to accelerate the inference at the edge as well. It caches prior inference results and reuses them when a cache hit occurs. The architecture consists of a mobile device issuing the request, an edge server caching the results, and a cloud server running the inference model and passing its results to the edge server. This procedure is most useful in latency aware contexts that depend on larger input data, like object detection in an image captured by a camera. Glimpse [46] proposes a technique which offers reuse of stale detection results of objects on current frames that have not changed. It achieves this goal by computing an optical flow between recently processed frames and the current frame. When running latency-sensitive inference tasks on video data, detection boxes around objects will often be off by a larger margin in the image, as the camera can shift between the time it takes for the information to pass through the network. By reducing latencies through caching, Glimpse achieves a precision increase of 1.6-5.5x [46].

Obviously, the caching approach cannot scale to higher numbers of data entries due to a fast-growing demand for memory. Thus, there are several proposals to optimize this approach by applying some heuristics like locality-sensitive hashing in Shadow Puppets [47], or Precog [48], where data is being cached on the edge server and mobile device together with a cached feature extraction model.

Furthermore, there is input filtering, which is most useful in video analytics. Here, nontarget-object frames of the input data are being omitted to reduce redundant computation of irrelevant frames. This improves inference accuracy, decreases latency, and reduces energy costs. NoScope [49] uses lightweight binary classifiers to spot the presence of certain elements in the frame, to determine if the frame is relevant for inference. Then there is FFS-VA [50], a pipelined system for multistage video analytics, the first eliminates background frames to obtain only parts of the video containing detection objects. These are then forwarded through a network to identify target-object frames. These are then passed through a Tiny-YOLO-Voc model to filter out frames that contain fewer objects than a certain threshold.

Hence, these frameworks aim at providing video streams of single cameras that only contain

relevant frames. RexCam [51] extends on the single camera concept by performing cross-camera analytics. Therefore, it maintains a spatio-temporal model that filters single video streams for relevant frames according to earlier observed frames of different cameras. Therefore, it can work with a lower computational intensity, and contextualizes the inference information of single video streams. Thus, it achieves a reduction in computational workload by 4.6x while improving the total prediction accuracy by 27%.

The approach of model selection is to train multiple models of different sizes and accuracies and choose the most efficient one based on environmental conditions, like available bandwidth or energy. This is, for instance, useful in areas, where the edge device is being charged by solar energy and cannot always run the most resource-hungry model. Park et al. [52] propose a so-called big/little DNN model selection framework that revolves around producing two dependent models with differing functionalities. It therefore combines a small model which is trained on the shape of the input data and determines if a classification task should be performed. If it's worthwhile, a big model exists to run the inference task.

Further, there are different other approaches that try to optimize prediction accuracy as well as inference latency. For instance, in object detection there are different model types that reach the lowest inference latency and highest accuracy for different kinds of images, e.g. MobileNet, ResNet, and Inception [4]. Hence, Taylor et al. [53] suggest a model selection framework to choose the optimal model according to a provided input. Lastly, there is the approach of Stamoulis et al. [54] which focuses on energy saving, by trading off accuracy for energy efficiency. This is achieved by running a hyperparameter optimization problem that considers the accuracy and communication constraints of a certain device. It is then solved by a Bayesian optimization to achieve an improvement on energy consumption by up to 6x.

Lastly, there are optimization techniques in place for multi tenant applications and application-specific optimizations. As this work is focusing on embedded systems and other resource restricted execution environments, these two approaches are not as relevant as the previously stated ones. Multitenancy is not of interest in these applications, as the default state of the device should be an energy saving mode, apart from the time when inference is required. Application-specific optimizations often concern themselves with either video data, which rely on devices that have high computation and energy resources, or hardware accelerators that do not portray an application design approach per se.

There are many different directions to take when attempting to optimize the model inference on edge devices. Most of them can and should be combined to reach a satisfactory level of performance in accuracy, latency, energy, and memory footprint depending on the use case. Thus, it is essential to carefully engage with the application's needs, the environments it will be deployed in and the capabilities the devices provide.

In this work, we will focus on furthering the insights into pruning techniques aimed at a specific form of DNN, namely graph neural networks, which have not been investigated thoroughly enough.

2.2.5 TinyML

With the rapid growth of availability of low-end embedded devices and the advancements in optimizing machine learning algorithms for less powerful devices, the AI community now shifts its attention to the realm of IoT. Thus, there is a need for a new paradigm concerned with exactly these types of devices running machine learning models. It's called tiny machine learning (TinyML) and aims to "provide low latency, effective bandwidth utilization, strengthen data safety, enhance privacy, and reduce cost." [5]

Often, in an IoT ecosystem, a device integrated with sensors outsources the generated data to

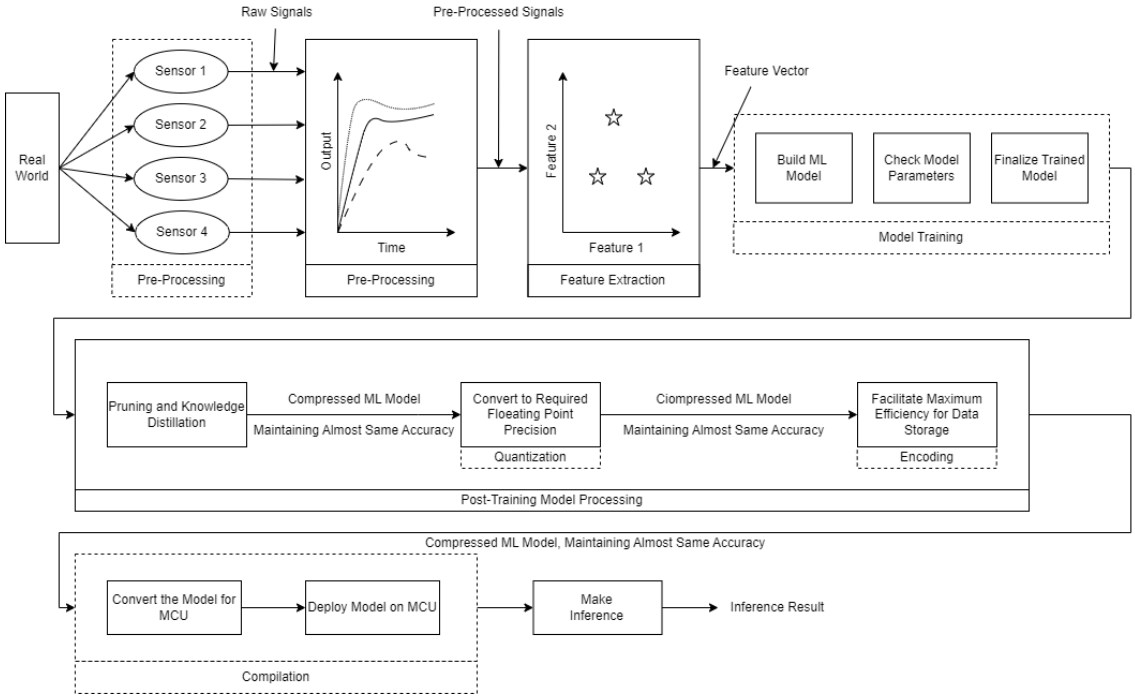


Figure 2.5: The deep compression workflow for neural networks running on MCU proposed in [5].

the cloud. The cloud service then aggregates the data and provides the required machine learning service. Thus, TinyML is especially useful in environments where no reliable connectivity, with high enough bandwidth, to a data center can be assumed. Consequently, there is incentive for an efficient hardware-software co-design. Additionally, moving the computation closer to the source of the data, carries a lot of potential for latency sensitive operations, especially when keeping in mind the rise of data being generated at the edge is predicted to out scale the total data movement to data centers by the factor of 22x [31].

TinyML focuses on executing optimized machine learning models on ultra-low-power (<1mW) Microcontroller Unit (MCU) with minimal power consumption. The main benefits of TinyML are enabling IoT devices with analysis of local measured data, independence of available network bandwidth, increase in security and privacy as fewer data is transmitted, reduction of latency, increase in reliability, increase in energy efficiency, possibility for data filtration, and reduction of cost.

As in EI, TinyML does not focus on model training, as it is an extremely energy consuming task. Additionally, the limited storage available on these devices poses a challenge for model training because frequent updates in weights and structure are inherent in machine learning model training. To obtain sufficiently performing neural networks as a suitable choice in an edge setting, the authors in [5] propose a deep compression workflow to facilitate these objectives. It is visualized in figure 2.5.

To obtain an adequately small neural network after training, several compression techniques should be utilized in knowledge distillation, pruning, quantization, encoding and compilation.

The combination of all of them is called deep compression.

Knowledge distillation concerns itself with transferring knowledge from a large teacher model to a smaller student model [55]. Therefore, a "softmax" function is used in the output layer of a neural network to obtain a probability distribution according to the input. This method retains not only the final label, but the similarity trait of the input to other classes as well. This creates so-called "soft targets" which are assigned a new parameter called "temperature". When the temperature is high, the spread of the prediction is high. Thus, to keep most of the information, a teacher model profits from a high temperature obtained from the modified softmax output. Afterwards, a smaller student model is trained with the pre-computed soft targets at the same temperature, which are the predictions made by the teacher model. The temperature then portrays a new hyperparameter, which needs to be optimized during model training, at the cost of computational time. Training the student model with the softmax input to a hard label, results in a smaller model, which imitates the prediction capabilities of a large model in terms of accuracy, as the loss function can be estimated using the Kullback-Leibler divergence [56].

Pruning is an iterative process that locates weights below a certain threshold to remove them and identify "dead" neurons without outgoing or incoming weights. Although this method ensures a sparse network, the obtained model often does not achieve sufficient accuracy. Thus, retraining is used to restore most of the lost accuracy even if 90% of weights are being pruned. Han et al. [57] show that pruning can reduce the network's size by up to 13 times without loss of accuracy. To further enable training of pruned models, Guo et al. [58] propose a dynamic network surgery by combining pruning and splicing. Splicing is a mechanism that enables the recovery of already removed connections if regarded as beneficial. This technique achieved a compression factor of 108, which is far superior to typical pruning approaches.

Quantization is downsizing the model by changing the number of bits utilized to a smaller representation. This often needs to be done to match the architecture of the MCU [59]. Depending on the newly chosen bit-width, e.g. 8-bit Int8 quantization, where 32/64-bit floats are mapped to 8-bit integers, can make a network smaller by the factor of 31 without losing accuracy [35].

Encoding attempts to further reduce the size of an obtained model, by applying an encoding scheme like the Huffmann encoding to it. It encodes more frequent weights with fewer bits and infrequent ones with longer ones. This can result in a reduction of 49 times, as shown in [35].

The final step of size reduction in machine learning models is compilation. Here the approach is to convert the model into a language that is readily understandable for the MCU, like embedded C. [60] Therefore, interpreters are utilized to convert a machine learning script into a more efficient representation, for instance C++ output files, when utilizing TensorFlow Light, which can then be burned onto the memory of a MCU.

As the importance of TinyML is rising, so is deep compression. Sadly, for the moment the developer is burdened with having to retrain the model several times until a satisfactory performing model is obtained, as the compressed model is desired to obtain similar accuracies as the original one. Thus, there is a requirement for a deep compression framework, which automatically handles these optimization tasks by providing a benchmark mechanism.

A similar deep compression workflow should be targeted when optimizing the GNN inference model used for the SWAIN project. For the moment, this is not possible as there are too few techniques, so far, which can handle the different variants of GNNs used in production. Therefore, this work aims at providing a pruning technique suitable for GNNs. An additional problem is the inherent nature of often shallow GNNs trained with large graphs in even larger datasets. This often results in a highly dense network with a small number of layers, which is not a suitable setting for knowledge distillation, and some pruning approaches.

2.3 Pruning

In section 2.2.4 we already briefly touched on the algorithmic optimization technique of neural network pruning. It is used in machine learning to simplify and reduce the size of complex models by removing less relevant structural components. It is therefore commonly used to increase the performance, efficiency and interpretability of models. This simplification corresponds to a sparsification of the network.

2.3.1 Model Sparsification

There exist many different methods, which aim at obtaining a sparsified model from a dense one. Many of these do not belong to the group pruning approaches. Nevertheless, they aim to reach the same objectives with different techniques. One of their many points of differentiation is, when they are applied during the lifecycle of creating a machine learning model.

There is no accepted classification for different model sparsification techniques. Therefore, we differentiate them according to the phase they are applied during model creation. These can be before training by preprocessing the dataset, during training by applying heuristics to the training data or conceptually restructuring the network, and lastly after training, with posterior investigation of structural components for their relevance on the output activation.

As our focus is on GNN sparsification, the following paragraphs will focus on highlighting selected techniques, which fall into this classification scheme.

The sparsification of networks is especially important to reduce the effect of overfitting in too dense feature spaces. Overfitting happens, when a trained model considers the variance in the training dataset too highly. Thus, it performs well on training data, but less good on unseen data.

This is not only the case in standard datasets, but especially also useful for graph datasets. Irrespective of the associativity of a graph, it contains plenty of edges between nodes to express some form of relation. This relation can be of any kind and often does not solely indicate highly correlated feature dimensions. Therefore, overfitting to spatially close neighbors that share an edge tends to be common. Further, training of GNNs is highly computationally expensive, as all connected neighbors need to be considered during the calculation of feature embeddings. Mainly this stems from two large matrix multiplications to first obtain an aggregated feature matrix by multiplying an adjacency matrix with a feature matrix, which then often gets multiplied by a learned weight matrix. Lastly, the obtained results usually gets passed through an activation function to obtain an interpretable and non-linear result.

The general equation of matrix multiplication looks as follows. Let A be an $m \times n$ matrix with elements a_{ij}, and B be an $n \times y$ matrix with elements b_{ij}. The matrix multiplication of A and B is denoted by AB, and is defined as and $m \times p$ matrix given by:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} * b_{kj} \tag{2.3}$$

Note that for the product AB to be defined, the number of columns of A must be equal to the number of rows of B. One can observe that this equation is being dominated by the size of n. Reducing the dimensionality or setting certain values to zero, such that they fall out of the scope of consideration, can drastically decrease the number of Multiply–accumulate (MAC) operations to obtain a result. In fact, removing elements from a highly dense matrix exponentially increases the performance of matrix multiplication.

Therefore, sparsifying the feature matrix used inside the node embedding computation step is highly beneficial, as it not only reduces the computational intensity in general, but additionally

reduces overfitting. Next to sparsifying the feature matrix, the learned weight matrix needs to be readjusted as well to fit the new dimensions of the feature matrix.

The first stage where sparsification of the model can be forced is by already sparsifying the training graphs. This can be done in a variety of ways. GraphSAGE [18] for instance, performs graph sampling, which essentially evaluates edges inside the graph for their significance and then selects a subset of nodes, a so-called neighborhood. The information of the sampled neighborhood will then be used for the feature aggregation steps. On top of essentially sparsifying the resulting model, it portrays an inductive approach by non-deterministically performing the sampling step, thus generating different training data from one base training graph. GraphSAINT [26] adds to this technique by sampling the training graph into batches of subgraphs to train the model. This way, it ensures a fixed number of well-connected nodes between every layer. Further, the approach offers techniques for adaptive sampling and node feature normalization to increase the performance even more. It is most useful for large-scale graph datasets, i.e. the Reddit [18] or the Protein-Protein Interactions (PPI) [61] dataset.

Next up there are sparsification approaches, which solely happen during training, by applying heuristics to the current model state. For instance, researchers in DropEDGE [19] concern themselves with GATs. Here, a number of attention heads compute attention coefficients corresponding to the neighbors of a certain node. These then get appended to every additional instance of the attention mechanism inside a given GAT model. Thus, at the end, every feature will have long feature vectors corresponding to the relevance of each neighboring node. When examining the computed coefficients, they found out that in most cases the variance between the coefficients is minimal. Thus, most coefficients are extremely similar. Therefore, they transform this into the heuristic that randomly dropping edges from consideration during training steps will not result in worse performing networks. This dropping of nodes can be regarded as adding noise to the training graph, which corresponds to the randomized production of unseen data. This is a common technique in machine learning in general. By applying their heuristic, they effectively counteract over-smoothing issues, which occur when training the model on a too small set of training data, as the model will expect unseen data to be too similar to the training data.

Lastly, there are model sparsification techniques, which happen after finalizing the training of the network. The channel pruning technique of [17] is a representative for that. Here, He et al. propose a novel Lasso regression to identify less relevant parts of a neural network to be subsequently removed. This is being done by computing relevance coefficients of hidden feature dimensions from the input to the output layer. Therefore, the L1-norm of the proposed Lasso regression is used and identified feature dimensions are removed. The authors in [6] then transfer this technique to GNNs trained on large-scale graphs. After pruning, the network will go through a smaller training stage, to retrain potentially lost accuracies, without adding additional feature dimensions to the network's nodes. Zhou et al. [6] prove the effectiveness of the approach by experimentally showing an average speedup of 3.27x for full inference, and 6.67x average speedup for batched inference, both with a negligible loss in F1-Micro of 0.002 and 0.003 respectively. In this context, full inference targets all nodes inside the graph, whereas batched inference targets only a subset of nodes during prediction tasks.

It is important to mention, that in the GNN context, most sparsification techniques rather aim at improving predictive performances by alleviating over-smoothing, overfitting or the effect of neighborhood explosion during training. They do not explicitly aim at reducing the computational complexity required to perform a forward pass inside the final model to make a prediction. In general, there appears to be a gap in research of algorithmic acceleration methods specifically aimed at GNNs.

The discussed and adapted technique in this work aligns with pruning, the last group of sparsification techniques, and aims at achieving high sparsification, such that computational

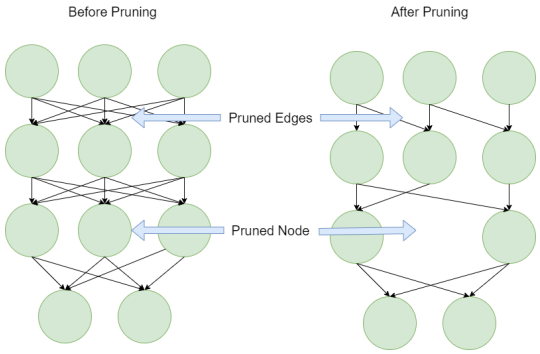


Figure 2.6: Visualized example of a neural network before and after pruning away certain structural components.

intensive GNNs can be executed on resource-constrained devices used in the SWAIN project's setting.

2.3.2 Definition

Pruning has been practiced since 1990, when a team of researchers led by Yann LeCun pruned the connections in a neural network trained to recognize handwritten digits [62]. They found that by selectively removing connections with small weights, they could substantially reduce the size of the network without significantly affecting its performance. The field of neural networks largely benefited from this, as it strongly suggested that pruning could be used to produce smaller, more efficient networks with a negligible loss in accuracy. Since then, pruning has become a common technique in neural network design and optimization.

The general procedure can be seen in figure 2.6. In this example, edges are being removed until dead neurons can be identified and dropped from the network. This can be seen in the lower part of the right-hand side of the figure.

When aiming at achieving a sparsified neural network, one of the biggest upsides of using pruning is that the elements to be removed can be hand-picked. Comparatively, other techniques mostly achieve sparsification by applying heuristics. This gives more control to the developer, who can fine tune the desired level of compression by altering the number of elements to remove.

Pruning can be categorized into different types, depending on the criteria used to decide which weights, connections, or feature dimensions to remove. The most wide-spread ones are:

- magnitude-based pruning, where the smallest weights according to a defined threshold are removed
- sensitivity-based pruning, which removes weights that have the least impact on the output
- structural pruning, which removes entire layers or neurons based on a determined relevance score

The proposed method of this work belongs to the category of sensitivity-based pruning. Therefore, the relevance of feature dimensions considered inside a weight matrix are being examined according to their relevance on the output activation. The layers of a model are then pruned back to front, to obtain a sparsified network with new internal hidden feature vector dimensions.

2.3.3 Channel Pruning using Linear Regression

Currently, there are little pruning techniques explicitly aimed at GNNs. This is because they are known to be computationally much more complex than other deep learning models and therefore often need to be executed in powerful environments. Therefore, most attention is being put on improving prediction results and decreasing training times. On top, the high computational intensity motivates most real-world model architectures to just include a few layers. Thus, there tends to be less opportunity, and therefore less incentive, for improvement in comparison to deeper model structures.

There are generalizable approaches, which do not consider architectural properties. For instance, simple magnitude based-pruning approaches often fail to achieve sufficient performance by only arbitrarily removing weights without consideration of the influence of the removal. Another approach focuses on inter-layer optimizations [63], which often fall flat in GNNs, as the input layers are often responsible for the highest number of MAC operations [6]. Thus, optimization gains often depend on how many layers are included in the model and how large the hidden feature vectors are between them.

In general, computations of node embeddings are responsible for the largest portion of MAC operations during forward propagation of GNNs. Therefore, optimization should focus on reducing the computational intensity inside the layers.

The typical choice in model for EI tasks is a lightweight one, that performs well with a small set of parameters and weights. For instance, DarkNet [64] or YoloV5 [65] are popular representatives for objective detection of camera streams due to a fast forward propagation at inference, which is essential for latency-sensitive applications. This mainly comes from the use of heavily optimized components resulting inside the models, thereby reducing the computational intensity at inference.

A similar approach is being followed by channel pruning, which is a sensitivity-based pruning method that aims at reducing the amount of feature dimension, so-called channels, inside a model's hidden feature vectors. Therefore, it aims to reduce the number of considered parameters during inference. It was first proposed in [17] by He et al. to enhance computer vision and pattern recognition tasks. There, the feature dimensions between the hidden layers of utilized CNNs were being examined according to their relevance for the final prediction. So, they employed a novel Lasso regression formulation for this cause, which effectively performed feature selection on computed vectors in the middle layers by penalizing them by the absolute value of the L1-norm of some error term. It was later adapted by Zhou et al. in the paper "Accelerating Large Scale Real-Time GNN Inference using Channel Pruning" [6] to accelerate large-scale real-time GNN. Currently, it portrays the only pruning technique specifically aimed at GNNs, as of our best knowledge.

The channel pruning technique of [6] applies the layer-wise reduction of hidden feature dimensions on GNNs, therefore reducing the size of the input and output dimensions of a layer. Hence, this performs inter-layer optimization as well. To achieve their goal, they formulate an optimization problem using Lasso regression to compute a relevance vector of the respective hidden feature dimensions. To apply the developed method, relevant layers containing weight matrices are being processed from the output layer to the input layer. The newly computed input dimensions are carried over as output dimensions for the previous layers. An illustration of the effect of channel pruning on one GNN layer can be seen in figure 2.7, where the blue and red areas denote pruned feature dimensions inside the weight matrix and computed hidden feature vectors.

The pruning approach focuses on altering the dimensions of the weight matrix by identifying less relevant feature dimensions to remove. As shown in the illustration, by resizing the weight matrix, the size of the computed hidden feature vector changes as well. This comes due to the inherent nature of matrix multiplication. When multiplying two matrices with the respective

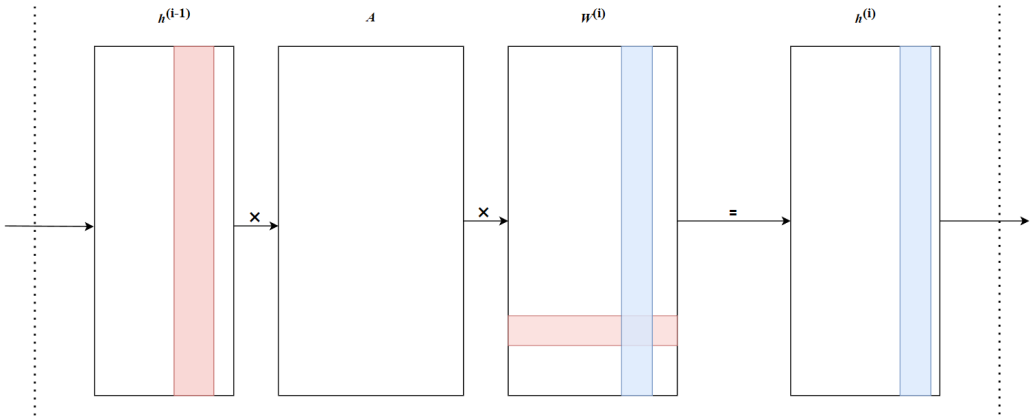


Figure 2.7: Illustration of a channel pruned GNN layer's components. The × symbol denotes a sparse matrix multiplication and the = sign denotes the final result after both sparse matrix multiplications. The hs are input and output hidden feature vectors, A the adjacency matrix, and $W^{(i)}$ the weight matrix inside layer i [6]. The red areas show the pruned channels from a previous layer, whereas the blue areas display the pruned channels inside the current layer. The illustration has been generalized and adapted from [6].

dimensions of (a, b) and (c, d) the resulting matrix will have the dimensions (a, d). As shown in the illustration, the three elements to multiply are the hidden feature vector of the prior layer h^{i-1}, the adjacency matrix A, and the weight matrix $W(i)$. The product of the adjacency matrix and the hidden feature vector results in the so-called feature matrix, which will then be multiplied with the weight matrix. The feature matrix has the row dimension of the hidden feature vector and the column dimension of the adjacency matrix. Therefore, the result from multiplying it with the weight matrix will have the row dimension of the hidden feature vector, and the column dimension of the weight matrix. Therefore, the result is determined by the input dimensions and the column dimensions of the weight matrix. Therefore, reducing the amount of feature dimensions inside the weight matrix simultaneously reduces the size of the resulting hidden feature vector as well. For the matrix multiplication between the elements to exist, the weight matrix dimensions have to match the output dimensions of the previous and the input dimensions of the following to shape the size of the hidden feature vectors that gets passed between the layers.

The proposed method of [6] is applied to two large-scale real-world datasets: Reddit [18] and Yelp [18]. The results show that channel pruned GNNs achieve comparable accuracy to their original counterparts while reducing the inference time by up to 40%. The authors also compare their method with other state-of-the-art techniques and show that their approach outperforms them in terms of both accuracy and latency during inference. Further, they analyze the impact of channel pruning on different aspects of the GNN's computation, such as the number of parameters and the computation time of each layer. They also provide insights into the characteristics of the pruned GNNs and show that they have a smaller number of channels but a similar receptive field to the original GNNs. Their channel pruning approach represents a novel and effective technique for accelerating large-scale GNNs for latency-sensitive applications. This method is not

only effective to reduce the computational intensity during inference tasks, but it accelerates the prediction as well without compromising accuracy.

The effectiveness of channel pruning can also be observed, when investigating the computational complexity and memory consumption of a forward pass inside a GNN. When performing a full forward pass of the entire graph during inference, a node-wise aggregation has to be done for every node. When denoting the input and output feature dimensions of the weight matrices W_k^i by $f_k^{in(i)}$ and $f_k^{out(i)}$, the number of nodes inside the graph as $|V|$, and the average node degree of the whole graph as d according to [6]. Then the average complexity per node $C_{full}^{(i)}$ and memory consumption $M_{full}^{(i)}$ at full inference can be determined by:

$$C_{full}^{(i)} = O(dmin(f_1^{in(i)}, f_1^{out(i)}) + \sum_{k=0}^{1} f_k^{in(i)}, f_k^{out(i)}[6] \tag{2.4}$$

$$M_{full}^{(i)} = |V|(f_0^{in(i)} + f_0^{out(i)} + f_1^{out(i)}) + \sum_{k=0}^{1} f_k^{in(i)}, f_k^{out(i)}[6] \tag{2.5}$$

The value of k references possible branches inside a single layer or structural element. These branches are often utilized in GNNs to divide the computation into chunks to be handled by specialized parts inside the building block of the network. Since every layer computes the output features from every node, each layer's computation and memory usage are distributed equally.

The total amount of memory consumption and computational complexity is highly reliant on the underlying model structure and architecture. Most variants run their inferences in a batched manner, consider fewer neighbors, or aggregate their feature embeddings in different ways. Assuming a model with the GraphSAGE architecture [18], where nodes are being aggregated from an L-hop neighborhood, the computational complexity is dominated by the last layer, as the number of nodes to consider increases exponentially the higher the L-hop distance grows inside a deeper network.

A main critique point for their work is the utilization of Lasso regression to identify feature dimensions to remove. It has known limitations in highly correlated settings, and tends to perform badly in scenarios where a small number of samples are available with a high amount of feature dimensions. These limitations are both present during the application of channel pruning, as the influence on the output activation is determined by a single batch of training data, and hidden feature dimensions tend to be highly correlated, as they usually aggregate heavily contextualized information from multiple input feature dimensions.

In this work, we want to help bridge the gap between GNNs and latency-sensitive applications in resource-constrained environments. Therefore, the existing channel pruning approach of [6] is being extended and optimized to especially perform better for high levels of compression. Thus, it will be investigated according to the effectiveness of the utilized Lasso regression, by comparing it to Ridge and Elastic Net regression when used during pruning different variants of GNNs.

Additionally, it is worth pointing out that the researchers in [6] focused on accelerating GNN inference for latency sensitive applications, like camera surveillance. We approach this topic from a different angle by aiming at reducing the computational intensity and memory requirements during prediction tasks. Therefore, we set our main focus on the trade-off between accuracy loss and achieved level of compression. The produced models will also benefit from faster inferences as well, such that the devices can turn themselves off, as soon as the inference task is done, thus conserving energy.

The channel pruning approach is also being extended to handle spatio-temporal models of the Torch Spatiotemporal (tsl) library [66] used on resource-constrained devices in the SWAIN

project's setting. Optimizing this technique is not only beneficial for low-end environments, but is in general beneficial for GNNs, as their performance is being retained, while reducing memory requirements and computational intensities. On top, pruning networks can reduce the effect of overfitting, as the assumed variance in the dataset will be lower as a result of removing highly correlated hidden feature dimensions.

2.3.4 Pruning in the Torch Spatiotemporal Framework

Pruning a GNN is no straightforward task, as the utilized building blocks are often heterogeneous, contain entire neural networks, or even consist of nested elements with components using neural networks to solve a sub-problem.

This work focuses on analyzing different linear regression models in a simplified setting, by applying it to different GraphSAGE [18] models trained with the GraphSAINT [26] approach, which are composed of higher order aggregators and graph attention layers. Therefore, the pruning process can be streamlined as the layers can be assumed to be of one of two kinds.

To be able to prune a model trained in the tsl framework, one first needs to understand the underlying building blocks a developer has at hand to build a spatio-temporal GNN. Therefore, they provide a range of different structural components to generate a GNN, which can handle spatio-temporal data. They provide a wide range of tools spanning datasets, data structures, inference engines, metrics and neural network components. A detailed overview can be found in [25], but in the following paragraphs a brief introduction of the most relevant components to this work is given.

The three most important submodules to utilize are layers, blocks and models, which are ordered by increasing levels of abstraction. A neural layer is the most basic building block of spatio-temporal models. It can internally perform basic encoding and decoding steps by normalizing input data, and performing an activation function to determine their outputs, but mostly handles straightforward tasks. For instance, the "DiffConv" layer implements a diffusion convolutional operation from [67]. Blocks perform more complex input transformations by chaining several operations to each other. These are being divided into encoders, to provide a contextualized representation of the input space, and decoders, which provide an interpretable form of a given data representation. Hence, a "SpatioTemporalConvNet" consists of several spatial and temporal convolutional operations, to enrich a graph embedding with a temporal component, relative to its spatial information. It can be gated, padded, have different lengths of temporal windows and contain certain dilation, such that the input data has some added noise to it to make the network perform better on unseen data. This showcases the wide range of already available configurability the blocks offer. A major upside of using blocks is that complex operations can be easily applied and experimented with, without the necessity to first conceptualize the operation chain, and develop it inside the framework. Similarly, models take up entire prediction tasks, which can be used as is or be utilized inside a model architecture or as a standalone network as well. It consists of several layers and/or blocks, such that a spatio-temporal graph signal can be processed, and a desired output can be produced. For instance, a forecasted node feature after a given amount of time steps can be predicted.

As realizing a spatio-temporal GNN is a complex endeavor, having ready-made building blocks to reside to is a big upside, although some may need to be adjusted, parameterized, and sometimes even added entirely. Therefore, it does not save a developer from conceptualizing and fine-tuning a network, but largely influences the development time of a network.

Applying the conceptualized pruning technique to said models is a complex task due to their nested nature. There is not only a need to analyze the network prior for potentially existing weight matrices to optimize, there are additional components, which are reliant on the input and

output dimensions of pruned parts and not be adjusted. For instance, there are norming steps or activation functions, which needs to be accounted for.

Further, due to the use of spatio-temporal convolutional elements, there are often multidimensional weight matrices with two additional dimensions for the stride and time window to consider. Therefore, the channel pruning approach needs to be adjusted accordingly, such that partial weight matrices are pruned on their own and reassembled afterwards.

Therefore, developing a generalized framework to accommodate all possible model architectures fell out of the scope of this thesis, so we focused on showing the effectiveness of the approach on selected elements used inside a bigger model.

2.4 Linear Regression

Machine learning practitioners aim at either drawing certain higher level information from a given dataset or training an instance which can predict attributes of a new unseen data entry. There are millions of datasets to draw from, which often contain a high number of so-called samples or data entries. The individual characteristics of each sample are called features, so the number of distinct attributes are called feature dimensions. As a simple example, let's assume a dataset which contains samples containing the height and width of a certain type of tree. These collectively define the input space of the dataset.

By using it, a model shall be trained to handle unseen data entries and predict certain learned behaviors. Therefore, a prediction about the tree height should be made according to its width. However, an important assumption must be made, which is crucial for the success of machine learning techniques: the features of a data point are correlated. As correlation is a complicated subject, some approaches simplify the setting and assume a linear relationship between feature dimensions. These are so-called linear regression models.

2.4.1 Linear Regression Models

Exemplary features of the dataset can be seen inside the scatter plot of figure 2.8. It displays known entities of the dataset. In machine learning the task often is to predict attributes of unseen data entries, according to a provided set of known feature dimensions. Therefore, the objective is to find a function, in the basic form of $f(x) = m * x + b$, which minimizes the error between predicted and the actual values. This objective function can be denoted as a basic linear function, by projecting the value of x to the space of f. In the given setting, we aim at projecting the width according to the height of a tree, while maintaining a small margin of error. The value for the intercept b is the value of $f(x)$ when the independent variable x is 0. The slope m represents the change in the dependent variable for a one-unit increase in the independent variable height, which $f(x)$ aims to predict. Therefore, residuals are computed, which are then taken over into a next optimization iteration. As residuals are just individual deviations, one wants to put them more in generalizable and interpretable terms to penalize the current prediction of gradient m and intercept b. In practice, certain so-called loss functions are used, which utilize penalization terms representing errors in the prediction. These are added to the regression model such that it can be iteratively refined to obtain a better predictive performance. In practice, this is being done by adding the loss function as an error term ϵ to the objective function. Thus, it reads as $f(x) = m * x + b + \epsilon$. A loss function can be calculated in a variety of ways by differently weighting residuals. In the chosen example, $f(x)$ is a linear function, and therefore denotes a linear regression model. Hence, an underlying assumption of linear dependency in the data is made. Further, this restricts ϵ to be greater than zero, as perfect linear dependency can rarely be observed in a non-artificial dataset.

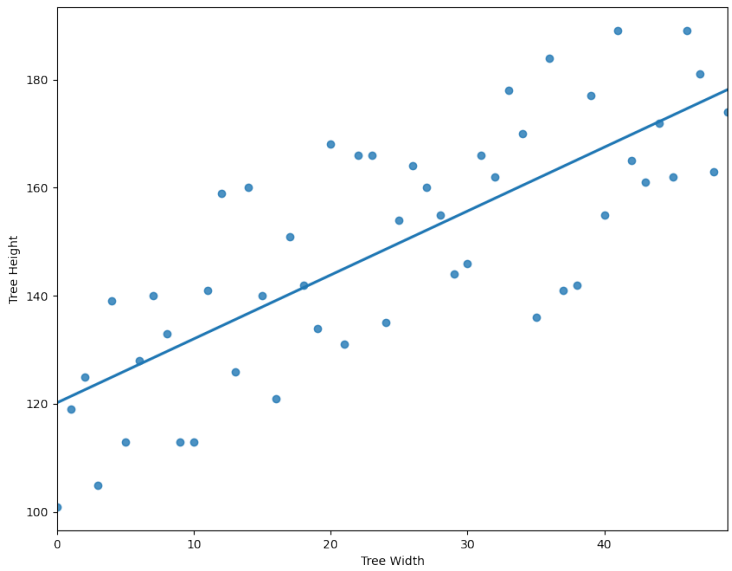


Figure 2.8: Scatter plot of tree height and width with an exemplary linear regression line. The underlying dataset was artificially created.

Linear regression models are used in statistical analysis between a dependent variable and one or more independent variables [68][69][70]. In the present example, these are the width and the height. The goal of linear regression is to find the best fitting line or plane inside the feature space that minimizes the distance between the predicted values and the actual values of the dependent variable. In our simple example with only two variables, the linear regression would compute a line, which displays the mapping of x, for instance width, to $f(x)$ values, height, a so-called projection. The graph in figure 2.8 shows an example of a linear regression line. To gradually optimize the objective function, $f(x)$ one wants to reduce the value of ϵ. In case there are more features to consider, the formula can be extended for each additional independent variable, by adding another gradient multiplied by the newly introduced variable, which adds another dimension to the projection space. In our example, by adding a second variable representing the root depth with y to $f(x)$ becomes: $f(x, y) = b + m1 * x + m2 * y + \epsilon$.

2.4.1.1 Variance and Bias

In standard linear regression, the objective is to minimize the sum of squared errors between the predicted and the actual values of the dependent variable, also known as the least-squares approach [71]. However, when the number of independent variables is large or the variables are highly correlated, this can lead to overfitting, which means that the model is too complex and performs poorly on unseen data. For instance, in graphs this problem is common, as many knowledge graphs are modeled to be associative with edges between nodes who share high similarities [72]. Further, numerous datasets contain redundant attributes, which will worsen the resulting model, when not handled properly.

Figure 2.9 shows three different regression lines in a dataset with high variance. The goal is to represent the data in the most suitable way possible. It is typical practice that outliers should not have high influence on the general prediction result, as they do not correctly represent the majority of data points. They are causing high variance in the regression model, which should not be considered closely. On the left-hand side of figure 2.9 we can see a regression line, which considers too much of the variance inside the dataset, thus overfits on the outliers. The result is a model, which can accurately predict the training data, but likely performs less good on novel unseen data. This is referred to as overfitting and is an undesired trait in predictive models. The middle graph shows a regression line, which under-fits on the dataset, as it does not properly represent the variance in the dataset and contains a high bias towards unseen data points. It therefore assumes most of the magnitude of the training features are caused by errors in the data. The final linear regression line on the right-hand side displays a good balance between considering variance in the dataset and bias towards outliers in it, thus portraying a good fit.

Linear regression generally has a larger bias towards the dataset than many other regression models, as it assumes linear dependency between the data points, thus a linear projection between the dependent and independent variables will be made. In most datasets this cannot be assumed, so researchers often perform feature selection to weigh linear dependent features higher than ones that are not similarly correlated to obtain better results. This is mostly done iteratively, and most often leads to the elimination of entire feature dimensions.

In general, the goal is to choose an approach with correctly optimized hyperparameters to find the correct balance between assumed bias and variance in the dataset, such that prediction accuracy stays high on unseen data entries.

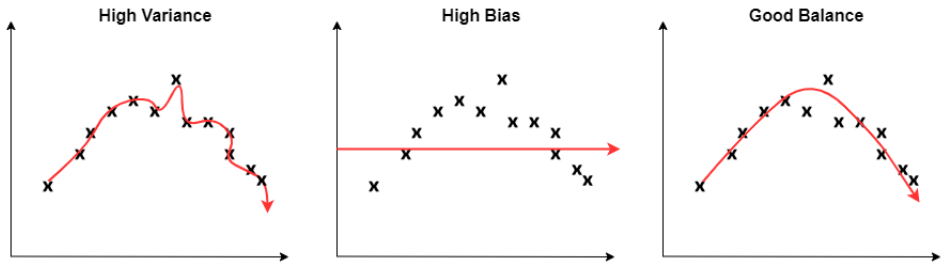


Figure 2.9: Scatter plots with regression lines of varying levels of bias and variance.

2.4.1.2 Penalization

As most datasets contain feature dimensions that do not relate to the output as much as others, the goal in linear regression models is to give the feature dimensions different relevance in the loss function. This is often done by assigning a certain weight to each dimension, which is progressively improved between every iteration.

Between every optimization step, the condition of the model is evaluated by assessing a quality criterion on the current state. One of the most wide-spread options to utilize is the so-called Mean Squared Error. It computes a coefficient according to the mean of the squared errors between the predicted variables and their expected values. Mathematically, MSE is defined as:

$$MSE = (1/n) * \sum((y_i - y_hat_i)^2) \tag{2.6}$$

where n is the number of data points in the dataset, y_i is the actual value of the dependent

variable for the ith data point, and y_hat_i is the predicted value of the dependent variable for the ith data point.

It penalizes larger errors more than smaller ones, as the residuals are being squared, which magnifies the effect of greater errors. The lower the value of the MSE, the better is the predictive performance of the current model. There are different metrics to assess how close to an optimum an objective function has been solved as well. Some popular ones are highlighted in 3.1.

So, a quality criterion of the current prediction is computed, which is then used as a bias term inside the loss function ϵ. The process of determining relevance coefficients is then repeated until the total loss of the objective function converges or the change between iterations becomes negligibly small. Finally, the coefficients of the best performing iteration are selected.

Lastly, it is worth mentioning that there are non-linear regression models as well, which perform better in settings, where feature dimensions are not linearly correlated, i.e. Random Forest regression [73] or Support Vector regression [74]. Thus, the proper choice of regression model is highly dependent on the dataset. In real scenarios, where datasets carry vast amounts of feature dimensions and sometimes no highly correlated ones. Therefore, researchers should try out multiple model variants and compare their performance using different quality criteria.

For this work, we focus on techniques aimed at datasets with multiple linearly dependent feature dimensions, as it is often the case in associative graphs and inside hidden feature vectors of neural networks.

In the following paragraphs we will focus on selected linear regression techniques, which have been widely applied in different kinds of machine learning models due to their simplicity, general applicability and overall effectiveness. These are Lasso regression, Ridge regression and Elastic Net. The former two are also known as L1-loss and L2-loss, the latter is a combination of the aforementioned ones.

2.4.2 Lasso Regression

Lasso (short for "Least Absolute Shrinkage and Selection Operator") [75] regression is a type of linear regression that uses L1 regularization to prevent overfitting and to perform feature selection. It was first proposed in 1986 by Santosa et al. in geophysics literature [68], to estimate the subsurface structure of the earth based on seismic measurements. Therefore, they utilized the then already established L1-penalty for fitting the regression function and computation of the coefficients. Later, it was further refined and popularized by statistician Robert Tobshirani [75] in 1996.

Lasso regression is the most popular choice, if one desires to eliminate features entirely from a regression model. It computes a relevance coefficient vector by determining the squared residuals of the model and adding a penalty term to them. The penalty is the sum of all absolute values of the independent variables' squared residuals multiplied by a configurable hyperparameter λ. By iteratively penalizing elements inside the set of features by their absolute error, some coefficients will shrink towards or even exact zero, which can result in a simplified model with fewer feature dimensions, thus performing feature selection.

The basic formulation of most linear regression models is referred to as "least squares" or "Ordinary Least Squares (OLS)" is to solve:

$$\min_{\beta_0,\beta} \left\{ \sum_{i=1}^{N} (y_i - \beta_0 - x_i^T \beta)^2 \right\} \text{ subject to } \sum_{j=1}^{p} |\beta_j| \leq t [76] \tag{2.7}$$

Here, a considered sample is composed of N cases, which each consists of p covariates and a prediction. Then, y_i is the prediction and $x_i := (x_1, x_2, \ldots, x_p)_i^T$ the covariate vector for the

ith case. β_0 is a constant coefficient and $\beta := (\beta_1, \beta_2, \ldots, \beta_p)$ is the coefficient vector, and t is a preconfigured free parameter that determines the degree of regularization, the so-called threshold.

The Lasso regression extends this with the L1-norm that serves as an additional penalty according to the absolute magnitude of the computed relevance coefficients β. Its strength of influence is controlled by the parameter λ.

Thus, the full objective function of Lasso regression is to minimize:

$$\min_{\beta \in R^p} \left\{ \frac{1}{N} \|y - X\beta\|_2^2 + \lambda \|\beta\|_1 \right\} [75] \tag{2.8}$$

where $|\beta|$ is the absolute value of the regression coefficients.

The tuning parameter λ allows for controlling the considered significance of the coefficients and thus the degree of penalization. A larger λ value corresponds to a more aggressive penalization and higher likelihood to remove feature dimensions from consideration. This increases the sparsity and decreases the size of the model. A smaller value of λ will lead to a solution more similar to the OLS.

This procedure is known as L1 regularization [77], and encourages sparsity in the data subset concerning prediction influence on a dependent variable. This is most useful in situations where there are numerous interrelated features, which don't add a lot of new information to a prediction, but rather overfit another property. For instance, one can imagine a dataset with length and width of an item that shall be used to predict the surface area. If the length was present inside the dataset as an absolute value and in a segmented form, for instance split up into five different features, a machine learning model would benefit from removing the segmented length entries to perform better predictions.

Lasso regression is one of the most popular regression techniques and is widely utilized in many different fields, including machine learning, economics, and biology.

The main drawback of Lasso regression is that some features converge to zero far sooner than others, as it is an iterative process. This behavior can be observed in figure 2.11a. Thus, during further optimization steps some feature dimensions will not be put into consideration anymore. This can lead to suboptimal solutions, as the removal of less relevant feature dimensions influences the context of all other feature dimensions. Therefore, a positive influence of the presence of a feature is not possible after its relevance converges to zero. This occurs most often if there is a group of highly correlated variables, where Lasso regression tends to select just one of them [78]. Thus, it is important to note that Lasso regression is not suitable for all situations. Hence, there are different regularization methods such as Ridge regression and Elastic Net that might be more appropriate choices for some data sets. Additionally, the selection of the tuning parameter λ can be challenging, as cross-validation techniques may need to be used to select an appropriate value.

The underlying studies of this work [6] and [17] utilize a Lasso formulation to identify less relevant feature dimensions on the output activation. It is used to identify hidden feature dimensions which have less influence on the output activation than others. This is done with a small subset of the data and corresponds to a setting where the number of feature dimensions is high, but the number of samples is low. Freijeiro-González et al. show in [78], that Lasso does not consistently select the correct feature dimensions in these situations, as it is susceptible to choose noisy feature dimensions and performs poorly in recovering relevant covariates. Further, they highlight deficient behavior in adapted versions of Lasso as well, i.e. relaxed Lasso or square-root Lasso.

2.5 Ridge Regression

Ridge regression is a regularized linear regression technique used to solve multicollinearity problems in datasets. It is also known as Tikhonov regularization, named after its inventor Andrey Tikhonov. It was first proposed in [69] by Hoerl and Kennard in 1970 and was later refined.

When independent variables inside a regression model have a high degree of correlation with one another, they are referred to as being multicollinear. This leads to overfitting and over-smoothing on feature dimensions. Thereby, computed similarity coefficients between attributes tend to be over-interpret, while other ones containing potentially essential information for the accuracies of predictions are neglected [77]. This effect is especially notable in datasets with a low number of samples but a high number of feature dimensions. Thus, the regression coefficients are consequently estimated with instability and imprecision, which can result in predictions that are incorrect. By adding a penalty to the objective function that is minimized by the squared OLS estimation of the regression coefficients, Ridge regression provides a solution to this issue. Therefore, it uses the L2 regularization.

The objective of Ridge regression is to minimize:

$$\min_{\beta \in R^p} \left\{ \frac{1}{N} \|y - X\beta\|_2^2 + \lambda \|\beta\|_2^2 \right\} [69] \tag{2.9}$$

with β being the feature dimension coefficient vector computed by:

$$\hat{\beta}_j = (1 + N\lambda)^{-1} * \hat{\beta}_j^{\text{OLS}} [69] \tag{2.10}$$

with N as the number of cases, X as the covariate matrix, y as the prediction outcome and λ as a regularization parameter. The sum of squared residuals, which determines the difference between expected and actual values of the dependent variable, is the first term in the objective function. The penalty term, which is the second term, is created by multiplying the total squared regression coefficients by the regularization parameter λ.

The regression coefficients' magnitude is constrained by the penalty term, which causes them to converge towards zero, but not zero. The model becomes more noise-resistant and less sensitive to minor changes in the data as a result. All the variables in the model still contribute to the prediction, even though the penalty term does not need any of the coefficients to be exactly zero.

In comparison, Lasso regression uses the absolute values from the L1-norm to shrink the regression coefficients towards zero. Unlike Ridge regression, it therefore automatically obtains sparse solutions without the need of removing small coefficients. The formulation of the objective function of Lasso regression can be found in equation 2.8.

Even though Lasso can be used for feature selection, which makes models simpler and more interpretable by reducing overfitting, it is more sensitive to outliers and unstable when there are strong correlations between independent prediction variables. Thus, it would more likely lead to a higher assumed variance and a lower bias in the data. As pruning aims to remove redundant information from a neural network, one can assume the data to regularize to be highly correlated, and thus Ridge regression should outperform it in situations, where a high magnitude of compression with a negligible loss in accuracy is required.

2.6 Elastic Net Regression

Elastic Net, first proposed in [79] by Zou et al. in 2004, is a linear regression technique that builds on top of Lasso and Ridge regression. Therefore, it combines the regularization methods of L1 and L2 to overcome their limitations. L1 regularization automatically performs feature

selection by shrinking coefficients to zero, but may produce unstable results in highly correlated environments and tends to perform when only few high dimensional samples are available. Whereas L2 regularization often creates highly complex models with low interpretability by not performing feature selection and tends to be sensible to outliers.

Elastic Net overcomes these limitations by combining both regularization methods. It adds a penalty term to the loss function that is a weighted sum of the L1 and L2 norms of the coefficient vector, with a tuning parameter, α, that controls the balance between the two [77]. Alternatively, two individual tuning parameters can be defined for both regularization methods, although this tends to influence the interpretability of the results.

Elastic Net is particularly useful when dealing with high-dimensional data, such as gene expression or image data, where the number of features is much larger than the number of observations [77]. A main weak point of Lasso regression. In such cases, performing good feature selection is crucial to avoid overfitting and improve the interpretability of the model. Elastic Net can also handle correlated predictors, which often occur in real-world datasets.

The objective function of Elastic Net is given by:

$$\min_{\beta \in R^p} \left\{ \frac{1}{N} \|y - X\beta\|_2^2 + \lambda_1 \|\beta\|_1 + \lambda_2 \|\beta\|_2^2 \right\} \text{[79]} \tag{2.11}$$

where the first part of the equation represents the OLS between the predicted and actual values, the L1-norm is the sum of the absolute values of the coefficients, and L2-norm is the sum of the squared values of the coefficients.

The tuning parameters, λ_1 and λ_2 are often exchanged with $1 - \alpha$ and α, such that it controls the balance between the two norms. When α is close to zero, the penalty is dominated by the L2-norm, and the resulting model will be similar to Ridge regression. When α is close to one, the penalty is dominated by the L1-norm, and the resulting model will be similar to Lasso regression. Intermediate values of α lead to a combination of both L1 and L2 regularization.

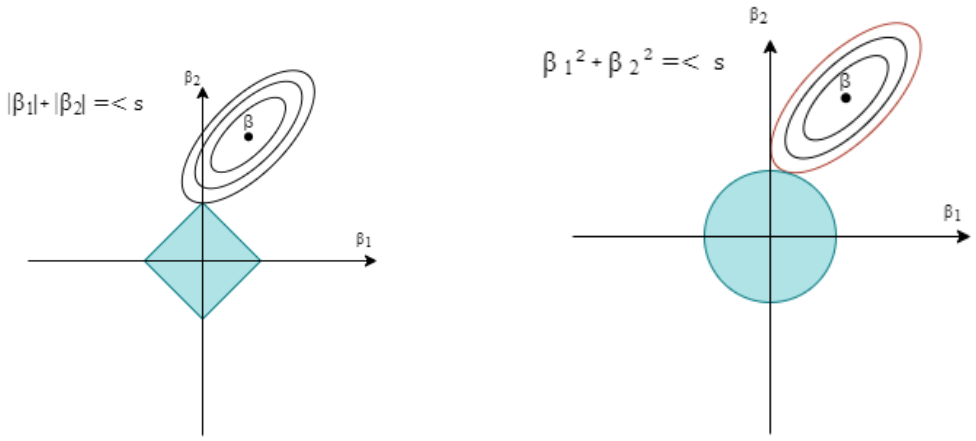
By adjusting the utilized values for α valuable insights about the dataset can be made. It reveals multicollinearity or an overabundance of redundant feature dimensions according to the chosen hyperparameter.

One advantage of Elastic Net over Lasso regression is that it can select more than n variables when n is the number of observations. This is because Elastic Net groups correlated variables together, while Lasso regression only selects one of them. In addition, Elastic Net can handle situations where the number of predictors is much larger than the number of observations, as it produces sparse solutions with a few non-zero coefficients [70]. It therefore combines the advantages of L1 and L2 regularization and can handle high-dimensional data with multicollinear predictors while performing feature selection. It reduces the chance of overfitting and can work with noisy datasets that contain many outliers.

Therefore, Elastic Net deems to be highly suitable in the setting of performing channel pruning in GNNs, as it can perform stable feature selection in high-dimensional settings with limited observations.

2.7 Geometric Interpretation of Lasso and Ridge regression

When geometrically comparing the objective function for Lasso and Ridge regression, one obtains two functions that rely on the computed coefficients of the feature dimensions. Figures 2.10a and 2.10b show the elliptical contours of the respective cost functions in the two-dimensional space. The different radius of the displayed ellipses correspond to the chosen hyperparameters λ_1 and λ_2 correspond to the magnitude of the L1-norm and L2-norm. The lower the value, the



(a) Geometric interpretation of Lasso regression in 2D-Space. Adapted from [77].

(b) Geometric interpretation of Ridge regression in 2D-Space. Adapted from [77].

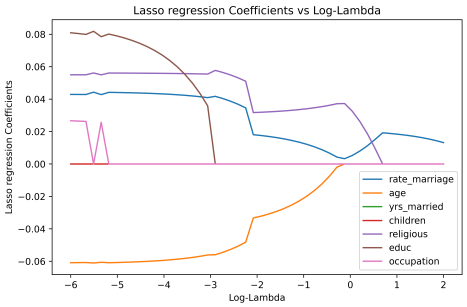
closer the predicted value for $\hat{\beta}$ gets to a standard linear regression model of OLS. Thus, the shaded diamond and circled area would increase [77]. The figures also indicate that the constraint regions of the L1-norm have its corners laying on the axes, thus a convex object that lies tangent to the boundary is likely to encounter a corner, thus setting the components of β for this feature dimensions to zero. Whereas, an n-sphere object boundary points are not positioned at a specific place and cannot be distinguished from all the other points. Thus, an encountered boundary point, by a computed ellipsoid of the loss function is no more likely to lie on exactly zero, than on any other point of the sphere.

This reiterates the behavior of Lasso and Ridge regression when sparsifying the feature space of a regression model. It shows that Lasso regression has the potential to reduce the relevance of feature dimensions to zero, if the cost function touches the constraint area at a boundary point, as they are placed on the axes. Whereas, Ridge regression, and conversely Elastic Net regression, are unlikely to have a tangent point with the cost function on exactly the intersection of the shaded areas and the axes. Therefore, they tend to get close to zero but rarely reach zero for the computed relevance coefficients.

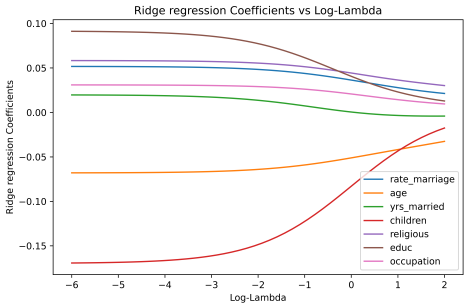
2.8 Comparison of Lasso, Ridge, and Elastic Net regression with a Use Case

Lasso performs good in situations where overfitting appears to be problematic, but bad in situations where a high level of sparsity is required. In these settings, one needs to pay more close attention to lost accuracy when omitting feature dimensions. The figures in 2.11 shows the computed relevance coefficients of Lasso, Ridge and Elastic Net regression in the affairs dataset [7], when utilizing varying numbers for the different hyperparameters of λ, determined by a logarithmic scale. The plots and linear regression models are being generated by the statsmodel framework [80].

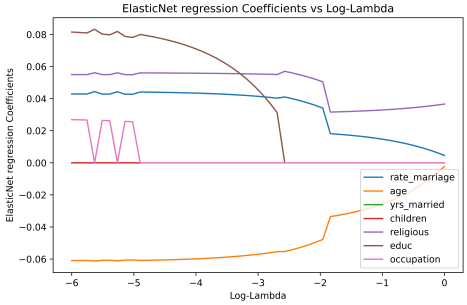
Therefore, it compares different values for the hyperparameters λ_1 and λ_2. There are seven different feature dimensions, which are being surveyed. On the left-hand side of the figure, you can see that Lasso regression only considers two feature dimensions from a λ_1 value of -0.122 and



(a) Relevance coefficients computed with Lasso regression.



(b) Relevance coefficients computed with Ridge regression.



(c) Relevance coefficients computed with Elastic Net regression.

Figure 2.11: Computed relevance coefficients of various linear regressions on the affairs dataset [7] with varying values for the penalization factors λ_1 and λ_2.

higher. Conversely, it omits the consideration of most of the feature dimensions entirely. More than half of the channels are being dropped from consideration after a λ_1 of -2.9 by reducing their relevance coefficient to zero.

The computed coefficients by Ridge regression are being displayed in the middle of the figure. Here, varying values of λ_2 lead to smaller and smaller coefficients of the same feature dimensions. However, it is noticeable that even with increasing values of λ_2 the relevance of feature dimensions still changes and does not converge to a certain value. Therefore, it shows the ability to rediscover relevant feature dimensions by increasing their relevance in later iterations. This shows that Ridge regression can potentially better identify a subset of relevant feature dimensions, compared to Lasso regression.

Graphs often have contained high amounts of feature dimensions, which are enriched by the context of their topological structure. Thus, most often a high amount of hidden feature dimensions are considered inside a model. Therefore, there is the opportunity for dropping a lot of redundant information and thereby reducing model complexity and size. Lasso regression achieves this automatically by reducing the coefficients by the absolute value of an error term until they reach zero. Ridge regression only diminishes the relevance of feature dimensions, but can be used to identify less and less relevant feature dimensions. This approach grants more stability, as the

likelihood of omitting channels, which might gain relevance, or are heavily influenced by outliers in later iterations, is lower.

Lastly, on the right-hand side, the computed relevance coefficients of Elastic Net are displayed. It is clearly noticeable that it combines the strengths of both approaches, by reducing the coefficients of only some feature dimensions to zero and asserting more relevance to dimensions that would have been removed by Lasso regression. Still, it leads to the consideration of only three feature dimensions from a value of -2.5 and higher, signifying that the results will suffer from less overfitting and multicollinearity simultaneously.

The experiments indicated that Lasso regression was outperformed by Ridge regression and Elastic Net in the affairs dataset. As this observation is heavily data-dependent, no definite conclusions can be drawn, but as they are highly related in nature, one can certainly assume different strengths and weaknesses between the approaches. These support our claim of Ridge regression or Elastic Net being more suitable for identifying feature dimensions if high levels of sparsity are required.

The use case was inspired by [77].

3 Methodology

The approach of this work will be realized by choosing a linear regression model to be used during pruning a model after training. It is supposed to remove dimensions of hidden feature vectors according to a configurable pruning budget. The pruned model will then be retrained to regain lost predictive abilities.

The first step is to experimentally evaluate the effectiveness of different linear regression techniques to identify the less relevant feature dimensions, to then remove these according to a given quota. This is done by extending the existing channel pruning technique of [6] to use different linear regression models to identify channels to remove, other than Lasso regression. Therefore, GraphSAGE [18] models are trained with the [26] technique, which are already partially sparsified due to their training approach, but still benefit from further posterior pruning. The different techniques will then be compared across a range of different datasets according to their F1-Micro and F1-Macro, two of the most popular metrics to evaluate machine learning models.

The obtained insights will be used to develop a pruning approach fitting for the tsl framework [66]. It is used in the SWAIN projects to produce the models that will eventually be deployed close to the waterways with pollution. Therefore, different variants of spatio-temporal GNNs are being developed, which in return would benefit from pruning after training. In theory, a spatio-temporal model can benefit more from posterior pruning as hidden feature dimensions increase according to a preconfigured temporal window, which should be considered during training and inference. For these tasks, more information is required and thus offers more opportunity to alleviate components without altering the perceived accuracies too much.

To realize the pruning technique, it is required to prune away the least relevant future dimension from the back to the front, while retaining the same number of input and output dimensions. This is highly architecturally dependent, as components vary in their usage. GNNs typically consist of a set of encoder layers, some processing middle layers, and at the end several decoder layers to produce the output. During pruning, these layers will be processed from back to front by evaluating the hidden feature vectors between the layers for dimensions to be removed. Therefore, the proposed Ridge or Elastic Net regression will be used. As the models to be used in the SWAIN project are not fully developed yet, the approach will be tested on an exemplary spatio-temporal GNN.

The novelty of the proposed work lies in the adaptation of Ridge and Elastic Net regression to GNN pruning, as well as performing the first pruning of spatio-temporal GNNs as of our best knowledge. Therefore, a proposed back-to-front channel pruning approach will be realized and adapted, which can be found in [6] and [17].

3.1 Metrics

To evaluate machine learning models, there is a wide range of established metrics to assert the resulting quality of their predictions. The process of correctly assessing these quality criteria already starts before training the model by splitting the training dataset into multiple parts that are used during different stages of developing a model. The most important split is the train dataset, which is used to develop a model by propagating it through the model and adjusting it according to the correctness of the obtained predictions. To not overfit a model on its training data, there often is a test split, which is used to validate the model's performance before training it. Since these are unseen data entries for a model, it acts similar to a real-world setting, where actual measurements are being ingested into the network to perform a prediction. Lastly, there can be a validation set as well, although it is often not required. It is used to understand the performance of the model in comparison to different model instances and hyperparameter choices, therefore helping to fine-tune the training procedure during several iterations.

In [81] popular machine learning metrics are being discussed. In a binary classification, there is only the distinction between a true or false label, representing if a data sample belongs to a class or not. As there are different possible errors, it is essential to distinguish between true positives, true negatives, false positives, and false negatives. The first two are correctly assigned labels, whereas the others are groups of falsely classified samples. For instance, a false negative belongs to a given class, but was excluded. Therefore, a false positive was assigned the class label, but does not belong to it. They often form the basis for evaluating how well a classification task has been solved.

The following table displays a selection of important metrics to assess the quality of machine learning and regression models found in [81]. These will also be used to evaluate the produced models by channel pruning in the results chapter 5:

Metric	Description
Accuracy	Accuracy is the most basic and popular metric used in machine learning. It represents the percentage of correctly classified instances by the model. It is computed by dividing the number of correct prediction by the total number of observations.
Precision	Precision measures the percentage of correctly identified positive instances among all the predicted positive instances. It is computed by dividing the total amount of true positives by all the positive observations. Precision is a good metric for applications where false positives can be costly. For instance, in the SWAIN setting, it is a highly undesirable behavior to cause a false alert due to the detected pollution and therefore alert the local authorities.
Recall	Recall measures the ability of a classifier to find positives instances in a dataset. It is computed by dividing the amount of true positives by the sum of true positives and false negatives. Therefore, expressing the percentage of found classes during a classification task. Recall is a good metric for applications where false negatives can be costly. For instance, in fraud detection, it is essential that no fraudulent data entry is overlooked.

Confusion Matrix	The confusion matrix is a tabulated representation of the predicted labels of a model according to their expected label. Therefore, it shows the four entries of true positives, false positives, true negatives, and false negatives. The confusion matrix is a basic and widespread tool to highlight the performance of a model without the inherent need to compute a single numerical metric. It can point out the general, the performance of both single- and multi-class classification problems.
F1 Score	The F1 Score puts the precision and recall of a model into relation. It is a good metric for applications that require a balanced performance of both metrics. The F1 score is calculated by: $$2 * (precision * recall)/(precision + recall) \quad (3.1)$$ As the F1 score is only computed for one class at a time, it is required to adapt the metric for multi-class classification. Thus, an average of all the classes' F1 scores has to be computed. There are three different ways of averaging these scores: macro average, weighted average, and micro average. Macro averaging computes the arithmetic mean between all the per-class F1 scores. Weighted averaging also considers the classes' support, which is the total number of predictions made for the class. Micro averaging removes the per-class look at the measures by aggregating all true positives, false negatives, and false positives and then computing a single F1 score over all entries. Therefore, the F1-micro score is more agnostic to the total number of classes to predict and their severity in misprediction, whereas the other two are putting more focus on the multi-class setting. In practice, mostly the F1-macro and F1-micro scores find application and will extensively be used in chapter 5.
Mean Absolute Error (MAE)	Mean Absolute Error measures the average absolute difference between the predicted values and the actual values. It is a good metric for regression problems where the magnitude of the error is important.
Mean Squared Error (MSE)	Mean Squared Error measures the average of squared differences between the predicted values and the actual values. It is a good metric for regression problems where larger errors are penalized more than smaller errors.
Root Mean Squared Error (RMSE)	Root Mean Squared Error is the square root of the Mean Squared Error. It is a good metric for regression problems where we want to penalize larger errors more than smaller errors.
Mean Absolute Percentage Error (MAPE)	Mean Absolute Percentage Error measures the average percentage difference between the predicted values and the actual values. It is a good metric for regression problems where the relative magnitude of the error is important.
Mean Relative Error (MRE)	The MRE is an accuracy metric that measures the average percentage difference between predicted and actual values in a dataset. It provides a measure of the average relative deviation of predictions from the true values.

Normalized Squared Error Basin Average (NSEBA)	The Normalized Squared Error Basin Average is typically employed to evaluate the performance of models that predict continuous values, such as regression models. It considers the squared differences between predictions and expected values that are normalized by the basin-specific standard deviation. Therefore, it describes how well a model performs on unseen data, as it considers the convergence behavior of training samples. Hence, it assesses how well it can assign observed data to its respective basins for the predictions. Lower values indicate better model performance.

Table 3.1: Important metrics used to evaluate regression and machine learning models.

There are several other established machine learning metrics available, but the listed ones suffice to form the basis of a proper evaluation for the produced pruned models in this work.

3.2 Realization of the Approach

To investigate the hypothesis of Lasso regression getting outperformed by Ridge or Elastic Net regression in detecting channels to prune, we first want to lay out the general approach.

Channels refer to column vectors inside the hidden feature matrix $h^{(i)}$, as shown in figure 2.7. To identify less relevant feature dimensions, different linear regressions are directly applied to the input channels to investigate the influence of feature dimensions on the output activation. Therefore, a model is trained, which is then being investigated back-to-front, from the output to the input layer. After a sequence of optimization steps, a percentage of channels is being pruned away, by applying a mask to the weight matrix, which contains zero elements in the position of the channel dimensions to prune. The percentage can be configured by a budget, either statically, by pruning away the same amount of every layer, or by defining a budget for every layer inside the model architecture.

Therefore, an optimization problem is being formulated to prune a single layer inside the model according to:

$$\underset{\hat{\beta}^{(i)},\hat{W}^{(i)}}{argmin} \left(\left\| Y^{(i)} - \tilde{A}h^{i-1} \odot \hat{\beta}^{(i)}\hat{W}^{(i)} \right\|_2^2 + \lambda \left\| \hat{\beta}^{(i)} \right\| \right) \text{[6]}$$

Where $Y^{(i)}$ is the expected feature matrix, $\tilde{A}$ the adjacency matrix and h^{i-1} the hidden feature vector of the previous layer, $\hat{\beta}$ the coefficient vector of the chosen regularization technique, and $\hat{W}^{(i)}$ the weight matrix.

The posed optimization problems consist of two sub-problems, where we want to optimize the weight matrix $\hat{W}^{i}$ in the first part of the equation with a fixed coefficient vector $\hat{\beta}$, and optimize β in the second part by either L1-norm, L2-norm, or a combination of both. Initially $\hat{W}^{i} = W^{i}$ and $\hat{\beta}^{(i)} = \vec{1}$.

The first part of the optimization problem corresponds to the Ordinary Least Squares (OLS) problem introduced in chapter 2.4. If the individual values for the penalization coefficients of λ are zero, the optimization problem gets reduced to the OLS. It seeks to determine a β coefficient

vector that minimizes the sum of squared residuals. This is equivalent to minimizing the MSE loss of a solution and will be used as such during the experiment.

The second part of the optimization problems can be formulated in three different ways:

	Type of Regression Model	β Optimization Problem
1	Lasso	$\underset{\hat{\beta}^{(i)}}{argmin} \left(\lambda \left\| \hat{\beta}^{(i)} \right\|_1 \right)$
2	Ridge	$\underset{\hat{\beta}^{(i)}}{argmin} \left(\lambda \left\| \hat{\beta}^{(i)} \right\|_2^2 \right)$
3	Elastic Net	$\underset{\hat{\beta}^{(i)}}{argmin} \left(\lambda_1 \left\| \hat{\beta}^{(i)} \right\|_1 + \lambda_2 \left\| \hat{\beta}^{(i)} \right\|_2^2 \right)$

Table 3.2: Types of extensions for the optimization problem of OLS to penalize the optimization task by differently norming a relevance coefficient vector β multiplied by the penalization coefficients of λ.

To realize the pruning technique, one needs to obtain a pre-trained model. The next step is to identify relevant layers inside the obtained GNN containing weight matrices. This is mostly a manual task as the structural components of a GNN are heterogeneous with several intermediate encoding, decoding, normalization or activation steps. Once the relevant layers are being extracted, they are being put into reverse order to ensure correct pruning of input and output dimensions throughout. This list is then iterated and pruned one at a time, by first pre-processing the pruning elements and parameters, then applying the technique with the configured linear regression model to the layer.

Firstly, the β coefficient vector is being optimized using a referential feature matrix, which represents the output of the layer according to a batch of training data. Then, the β coefficient vector is being optimized by performing a forward pass through the layer and comparing the output, generated by multiplying the weight matrix with the computed coefficient vector, to the referential feature matrix. Therefore, a MSE loss is being determined, which represents the left-hand side of the optimization problem. Then, this loss is appended by the chosen regularization loss of either the L1-norm, the L2-norm, or a combination of both. To utilize the computed value, a stochastic optimization function is used to adjust the coefficient vector for β. The degree of optimization can be controlled by setting a learning rate parameter.

This procedure is being repeated according to a configured number of epochs. Then, the computed coefficient vector is used to optimize the weight matrix, which requires readjustment according to the defined relevance coefficients. As this only relates to the first part of the optimization problem, only the MSE is computed to obtain a loss value for an optimization function working on the weight matrix. The severity of adjustment can again be controlled by setting a learning rate parameter.

After a second period of optimization epochs, the procedure concludes, and the β coefficient vector is being clipped according to the defined budget. Therefore, the lowest values of β are being replaced with zeros, and a mask is being determined representing the new input dimensions of the layer, which shall later be used to reassemble the pruned model.

Then, the layer's weight matrix is replaced by the optimized one to test the intermediary performance of the model after every optimization step. This is done to more easily spot high influences on output accuracies according to pruned hidden feature dimensions more easily.

Afterwards, the process carries on with the next layer, the predecessor of the prior pruned layer. It is already parameterized with the input mask of the previously pruned layer, such that the output dimensions will match the input dimensions of the following layer inside the network.

When all the layers have been treated, the weight matrices are aggregates, as well as input and output dimensions. Then, a pruned version of the model is reassembled. This model will then be retrained using the original training dataset, such that lost accuracies from dropping dimensions of hidden feature vectors can be recovered.

3.3 Pseudo Code

The pseudo code of an algorithm realizing the channel pruning approach can be found here:

Data: Model M, pruning budget
Result: Pruned Model M'
create a list of layers of M in reversed order apart from input and classification layer;
while *not all layers are pruned* **do**
  obtain W of current layer;
  compute referential feature matrix R using weight W and training data batch;
  initialize linear regression model with referential feature matrix, β vector of 1s with size of layers out dimensions, and W;
  for β *optimization epochs* **do**
    compute MSE loss by comparing $W * \beta$ and R;
    compute loss of linear regression variant for β;
    perform an optimization step using a gradient-based optimization technique for β based on the sum of losses;
  end
  for W *optimization epochs* **do**
    compute MSE loss by comparing $W * \beta$ and R;
    perform an optimization step using a gradient-based optimization technique for W based on the sum of losses;
  end
  apply β to W by sparse matrix multiplication;
  set lowest values in W values to zero according to budget and obtain pruned weight matrix W';
  obtain out mask m;
end
use pruned weight matrices W's and out masks ms to assemble pruned model M' with new internal input and output dimensions;
retrain lost accuracies for pruned model M';

Algorithm 1: Channel Pruning algorithm using a configured linear regression to compute the relevance coefficient vector of β.

3.4 Channel Pruning of spatio-temporal Models

In the second part of the practical evaluation of this work, the channel pruning technique with various linear regression techniques has been applied to spatio-temporal neural networks as well. Therefore, it was necessary to adapt the existing technique fitted for models with the GraphSAGE architecture to be suitable for models trained with the tsl framework. To make the implementation

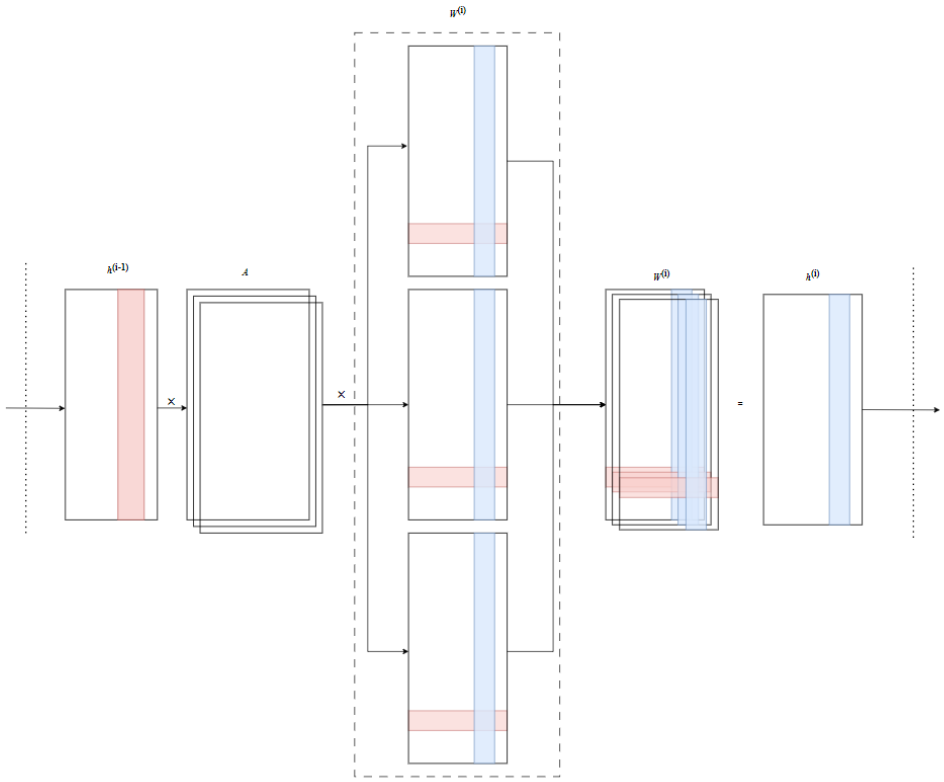


Figure 3.1: Visual illustration of an exemplary multidimensional pruning by splitting the weight matrix $W^{(i)}$ in layer i along the third dimension to obtain two-dimensional and prunable weight matrices. The hs are input and output hidden feature vectors, and A the adjacency matrix.The $\times$ symbol denotes a sparse matrix multiplication, and the $=$ sign stands for the final result after both sparse matrix multiplications. The red areas show the pruned channels from a previous layer, whereas the blue areas display the pruned channels inside the current layer.

more architecture agnostic, the first step was to refactor the existing code from [6] such that it is modularized, encouraging reuse, enhancing readability, and allows for extension.

Additionally, we opted for providing using the layers forward function to produce the referential matrix during the optimization process. In the original example, the researchers chose to implement it, by multiplying an adjacency matrix from a sample batch with the present weight matrix, which is not realistic in most settings. By utilizing higher-order aggregators, which solely perform matrix multiplications a number of times according to the provided order number, which will then get passed through an activating function, mostly ReLU, it was a fitting approach, but not possible to transfer to different model architectures.

Further, the provision of pruning parameters was simplified, the choice of pruning technique was removed from the actual implementation context, such that it is not considered to be essential during pruning itself, the aggregation of results was put into an own function, and the retraining procedure was separated from the training procedure.

Therefore, one can easily plug in the pruning functionality now to any given PyTorch model, as long as the correct parameters of the layer in question, the pruning parameters, the weight matrix to optimize, and a referential feature matrix can be provided. This technique works irrespective of layer or neural network block composition, as it will solely alter the weight matrix. Then a forward propagation through the layer or block can be performed by utilizing a given training batch, and comparing it to the referential feature matrix and computing its loss according to the MSE.

A task that often required significant effort was to correctly compute the referential feature matrix, as it is required to emulate the layer's or block's forward propagation function to the point where a single weight matrix is utilized. In standard GNNs this often solely requires calling the forward function a neural network module typically supports. In the spatio-temporal context of the tsl library, and in many different scenarios, this is not as easy, as there are often multiple convolutional layers utilized inside a single block, which sometimes even carry multiple weight matrices to perform their task. For instance, the exemplary model that was used to show the effectiveness of the approach contained one weight matrix per temporal step to consider during inference. Therefore, the referential feature matrices always had in the correct state of this exact inference step.

To prune multidimensional weight matrices the matrices are split along their dimensions, until a two-dimensional one is obtained, an illustration of multidimensional pruning can be seen in figure 3.1. The pruning approach can then be applied as before, by forwarding a batch of training data through the network and replacing the partial weight matrix after every β coefficient optimization step.

During the subsequent pruning, the partial weight matrix gets replaced inside the original object, to compute the referential feature matrix during and after pruning. After sequentially pruning all the split weight matrices, the input mask for the layer is determined, containing the information, which feature dimensions have been removed. This therefore defines the number of input dimensions a layer has, and needs to be used to reshape the weight matrix of the previous layer to ensure proper inter-layer dimensionalities.

Apart from pruning the layers from back to front, with their weight matrices, it is required to reassemble the obtained model afterwards. Therefore, the correct input and output dimensions and the input mask for a block of pruned elements need to be determined. The reassembling procedure therefore needs to account for a data stream of a given size inside the network to be able to reduce its dimensionality for pruned elements and increase it again afterward.

To accommodate pruned elements inside a model, it is often required to adjust their implementation such that they can utilize pruned weight matrices and input masks. Therefore, it is necessary to extend them with methods to load pruned weights, adjust their forward functions,

and reconfigure their internal dimensions according to newly obtained dimensions according to the desired compression level. Being able to set the weight matrix from the outside is not standard practice, thus will require lots of rewriting of cod and may need to be refactored if the underlying framework or its dependencies change.

Lastly, the retraining procedure needs to be configured and executed similar to the model's original training procedure, although it often requires fewer epochs to converge.

Obviously, there have been more adjustments to the original source code of [6] to run the tests in various combinations of compression levels, regression technique, and hyperparameters, which shall be further discussed at this point. The source code of the implemented linear regression models used during channel pruning, together with their optimization strategies and generalized base class, can be found in the appendix 7.1 and 7.2.

4 Experimental Setup

In the following sections, the experimental setup will be explained. First the utilized model structures are explained in 4.1, then the datasets used for training the models are motivated in 4.2, followed by the description of how the channel pruning of the obtained models was realized in 4.3.

4.1 Models

The experimental setup was to apply the different pruning variants to models produced with the GraphSAINT [26] learning method, which uses GraphSAGE [18] models. All of them are trained to solve node prediction tasks, with each working on multi-class datasets. Two are being trained to perform multi-label node classification, the other one is assigning single labels to a single prediction node. The underlying model architectures consist of a classifier, several convolutional and an input layer. The convolutional layers of all models are realized by higher-order aggregators mentioned in the GraphSAINT paper [26], therefore they offer great comparability between different model instances. Additionally, the convolutional operation of the higher-order aggregators only requires basic matrix multiplications of a provided feature matrix with the present weight matrix. This increases the influence of the defined pruning budget on the overall pruned model performance, as the overall operations including pruned weight matrices increase. This offers better interpretability of the resulting model structure, by highlighting the influence of the compression level of weight matrices on the measured prediction accuracies.

Afterwards, the techniques have also been applied to spatio-temporal models, which consist of more heterogeneous elements. They often contain different functional components, which either do not include weight matrices, or do there is no significant benefit in reducing the dimensionality of them, like norming operations.

4.2 Datasets

The three different datasets used to train the models are the Protein-Protein Interactions (PPI) dataset [18] used for protein classification, an undirected version of the Arxiv Open Graph Benchmark (OGB) dataset [2] classifying subject areas of Arxiv CS papers, and the Flickr image relationship dataset [82] classifying shared common metadata of user-uploaded images. An overview of important characteristics of the datasets are given in table 4.1. All of them are large-scale graph datasets with a high number of parameters. The Arxiv and Flickr datasets have been chosen as they are comparatively less big in training data, graph size and feature dimensions to other large-scale graph datasets. This has been done to anticipate the computational restrictions of edge devices, which are a limiting factor. Many large-scale graph datasets contain graphs with millions of nodes to run inferences on, which do not portray a suitable use case for the edge setting at the moment. Therefore, we do not assume that developers aim to deploy trained model instances for these scenarios close to the edge, as computational intensities tend to be high, input data that needs to be passed through the network is big, and therefore have long-running prediction tasks. Hence, we deem these large-scale networks as unfitting for resource-constrained

devices, and focus on datasets with less large components. The effectiveness of channel pruning has been shown for the large-scale datasets of Reddit [18], OGB-Products [2], and Yelp [26] in [6]. These are all high dimensional datasets, which tend to include a lot of data, which is less relevant to the output activation. Therefore, many feature dimensions can be omitted without a considerable loss in prediction accuracy. Further, the hypothesis should not be tested on extreme cases with huge values for internal dimensions, as this might influence the correctness of results.

Dataset Name	#Graphs	#Nodes	#Edges	#Classes
Arxiv (undirected)	1	2,449,029	1,166,243	40 (m)
Flickr	1	105,938	2,316,948	7 (s)
PPI	1	132,534	39,561,252	121 (m)

Table 4.1: Overview of utilized graph datasets available at `https://github.com/snap-stanford/ogb`. Their documentation can be found at `https://ogb.stanford.edu` [2]. Where m describes a multi-class multi-label classification problem, s describes a multi-class single-label classification problem.

The PPI dataset has been chosen as it displays a more complex setting. Protein-protein interaction classification is a challenging task, with many conventional techniques failing to achieve sufficient prediction accuracy. This can be observed irrespective of the chosen prediction task of link prediction, node or graph classification. This comes from the inherent nature of proteins being composed of long chains of amino acids, which can alter their behavior according to their spatial alignments to each other. In general, there are four different protein structural levels, which influence the proteins' traits. These structural levels are all part of different classification tasks, which tend to get less and less accurate the more elements need to be considered. The primary protein structure is simply the classification of a protein according to its amino acid chain. Then there is the secondary protein structure, adding a spatial alignment to it. Machine learning tasks aim to predict if the protein will align itself as an alpha-helix, or as a plated sheet. As amino-acid chain sequences are long, multiple different primary structures can occur inside a single protein. Therefore, the tertiary protein structure aims to include the interactions between the secondary structural elements. Lastly, the quaternary protein structure describes the presence of multiple amino acid chains inside a single protein structure. These then align themselves next to each other and tend to be entangled with each other, due to the interdependence of the attractions of all spatially close amino-acids.

In the given setting, the PPI dataset is used to train a model to predict the function of a protein when given its molecular structure represented by a graph. The prediction task includes the consideration of all structural levels.

Finding proper similarity functions to correctly spatially relate protein structure with each other is a popular research area to facilitate vaccine and drug discovery. In general, there is a lot of information to consider during training and inference, and it is difficult to correctly determine which parts of the graph's information have the highest or lowest amount of importance on the output activation. Therefore, we assume that pruning a corresponding model will result in heavy losses in accuracy, as the model information is expected to be highly interdependent. We not only choose this dataset, to show currently existing problems researchers face, when attempting to invent generalizable pruning approaches for all settings, but we consider it to be the most similar to a spatio-temporal dataset. Like the PPI dataset with its different levels of spatial information connected to each other, we consider a spatio-temporal network to be similar in the regard that convolutions are being done in more than one temporal plane.

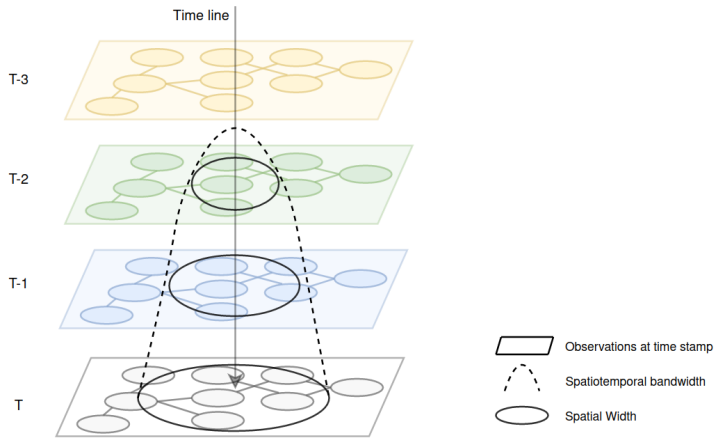


Figure 4.1: Visual representation of the inference space of a spatio-temporal model with a temporal window size of three.

A graphical example of a spatio-temporal inference space can be seen in figure 4.1. It shows a visual representation of how a spatial temporal model performs predictions according to a given time window. The highlighted bandwidth is an abstraction of an actual model's consideration, as it would not be uniformed and therefore ill-shaped. As the different temporal layers are interconnected to perform prediction tasks, extensively removing components from such networks, might lead to high deviations in prediction accuracies, as in our experiments it is not feasible to predict the influence of removing temporal counterparts of a weight matrix. Therefore, a model trained on the PPI dataset portrays similar traits to a spatio-temporal model, and serves as a fitting item of comparison.

4.3 Generation of Pruned Models

To obtain proper comparability between the produced pruned models, the same pre-trained model instances have been used for each dataset variant. The evaluation metrics of the baseline models can be found in table 4.2. They obtained a state-of-the-art performance for the given node classification tasks and were trained with the GraphSAINT method. All of them are GCNs and consist of several layers of higher-order aggregators, capable of accessing information about all neighbors inside a certain distance to perform information aggregation to compute node embeddings during forward passes in the network. The goal was to obtain pruned versions, which contain less feature dimensions according to the pruning budget, with an insignificant loss in accuracy. The main criteria for evaluating the goodness of pruning were F1-micro and F1-macro scores, explained in chapter 3.1.

All model variants have been pruned extensively with the help of different linear regression techniques. A total of 960 pruned models have been produced and will be evaluated in the upcoming chapter 5. Additionally, hyperparameter optimization has been done to find the correct penalization factors used during pruning. Further, an evaluation has been made according to losses in accuracies and compression level according to the pruning budget and how to trade them off correctly. The different compression levels used were 2x, 4x, and 8x. The varying values for

Dataset Name	#Convolutional Layers	F1-Mirco	F1-Macro
Arxiv (undirected)	3	0.715	0.5052
Flickr	2	0.5100	0.1987
PPI	2	0.9002	0.8873

Table 4.2: Overview of models used for pruning.

the λ penalization factors were multiplied by two between every new training iteration.

The proposed technique is generally applicable for GATs as well, but requires refitting of the current existing procedure, as no weight matrix exists. Their utilized attention mechanism uses so-called attention heads, which can be pruned according to the same strategy, but need to be reassembled differently. As there is no inherent need to prove the effectiveness of the technique on GATs, as it was already shown in [6], and a lack of comparability to the other GCN variants, pruning the considered feature dimensions in attention heads of GATs has been removed from the scope of this work.

Dataset Name	#Pruned Models	Size in MB	Average Size after Pruning in MB		
			2x	4x	8x
Arxiv (undirected)	305	4.9 MB	2.57 MB	1.47 MB	0.95 MB
PPI	334	6.8 MB	2.57 MB	2.06 MB	1.94 MB
Flickr	321	2.1 MB	0.81 MB	0.37 MB	0.19 MB

Table 4.3: Overview of the pruned models using the channel pruning technique found in algorithm 1 using Lasso, Ridge, and Elastic Net regression.

5 Results

In this chapter, first the results of the conducted benchmark of channel pruning different GNNs with the help of various linear regression techniques is being evaluated. This is done to answer the hypothesis that Lasso regression is being outperformed by Ridge or Elastic Net regression in identifying less relevant hidden feature dimensions to remove. Therefore, extensive experiments have been done with compression levels of 2x, 4x, and 8x, and differing values for the hyperparameters of λ and discussed in section 5.1.benchmark

Afterwards, the discussed channel pruning approach is being applied to spatio-temporal models trained with the tsl framework in section 5.2.

5.1 Results of Channel Pruning with different Linear Regression Techniques

To discuss the hypothesis of Lasso regression getting outperformed by different linear regression techniques when used during channel pruning GNNs for resource-constrained environments, extensive benchmarks have been conducted. The results are being examined according to the obtained F1-micro and F1-macro scores of the pruned models. Hence, 960 models have been pruned and retrained that have previously been produced using the OGB datasets of Arxiv, PPI, and Flickr [2]. The quality criteria of the pruned models have been tracked using CSV files. These have been processed in Python 3.9 using Matplotlib 3.7 [83], pandas 1.5.3 [84], and seaborn 0.12 [85].

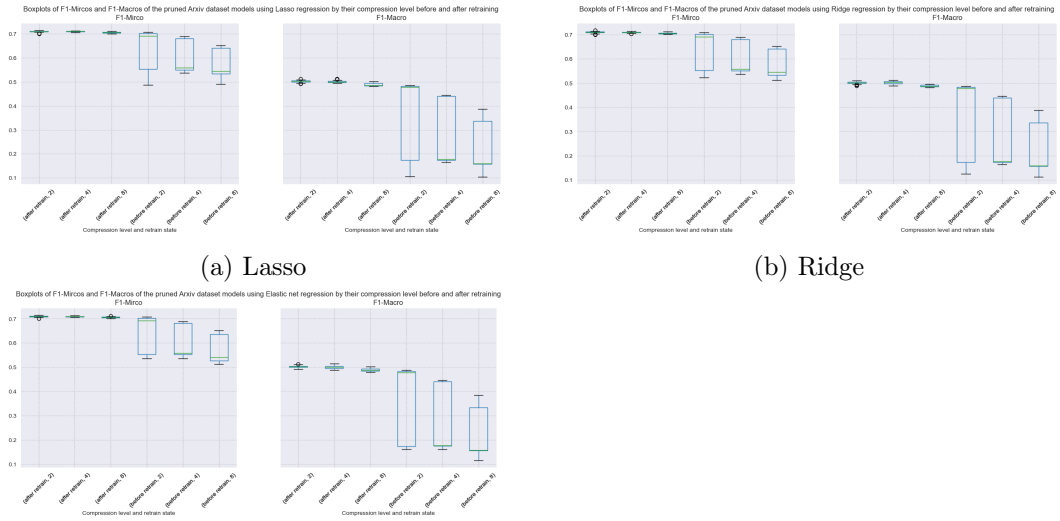
The first step in data analysis is to obtain the dataset and clean it from invalid values. Afterwards, the interpretation of the benchmark data could be done.

5.1.1 Interpretation of Box Plots

When analyzing the data with box plots, grouped by the models' compression level, F1-micro, F1-macro, and hyperparameters, the most noticeable aspect has been the importance of retraining the model. The box plots in figures 5.1, 5.2, and 5.3 show that the obtained accuracy metrics not only perform better after retraining, they also show that irrespective of obtained accuracies, the models tend to converge to a stable mean of F1 scores after retraining, without increasing the dimensionality of the weight matrices. This behavior can be observed in any dataset. In general, this is a requirement for performance optimizing approaches, as stability in the expected results incentivizes developers to apply a technique to their code, even if it causes implementation overhead.

Box plots show the average values of a dataset with the 25th quartile, and the 75th quartile as limiters for the drawn box, the median as the highlighted line inside the box, the "minimum" and "maximum" of the dataset as so-called whiskers, which are drawn as limiting bars above and below the box, as well as any outliers outside the whiskers [86]. The "minimum" and "maximum" values are computed by

$$25th\ quartile\ - 1.5 * (75th\ quartile - 25th\ quartile)[86] \tag{5.1}$$



(a) Lasso

(b) Ridge

(c) Elastic Net

Figure 5.1: Box plots of pruned **Arxiv** models' F1-micro and F1-macro scores before and after training, grouped by compression level.

and,

$$75th\ quartile + 1.5 * (75th\ quartile - 25th\ quartile)[86] \tag{5.2}$$

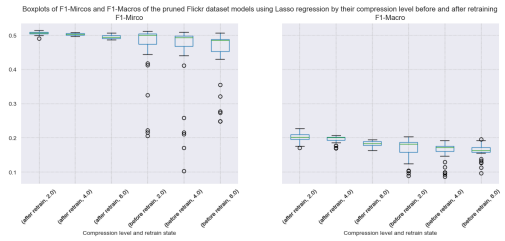
respectively.

The box plots of figure 5.1 show a large spread in obtained accuracies for both surveyed metrics before training and for all linear regression techniques. Especially the values for the F1-macro score, even at lower compression level, are fluctuating a lot. This means, that on average, each prediction class F1 score either gets highly influenced, or not too much at all, which can be related to the use of single training batches during pruning to optimize the performance of a layer. If the current batch under consideration does not include some class labels, it has a high influence on correctly determining the influence of removing certain dimensions on the network. This loss in predictive ability only resurfaces when evaluating the network after pruning.

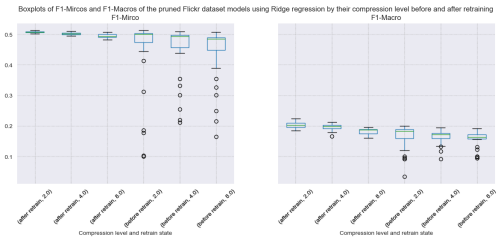
The medians of the F1-micro and F1-macro scores are spread out quite far as well, with a comparatively high expected median for lower compression levels, but a small one for higher compression levels. This means that before retraining, the network might even obtain good F1 scores, but often cannot be assumed to perform well as the spread in expected values is far too large.

The effect of retraining can be seen by how small the distances are between the elements of the box plots. This means that the observed F1 scores are extremely stable after retraining, and do not deviate much between different pruning attempts. This process does not increase the dimensionality of the weight matrices, thus does not induce higher computational complexity or adds to the size of the network, which would influence the memory efficiency of a model as well.

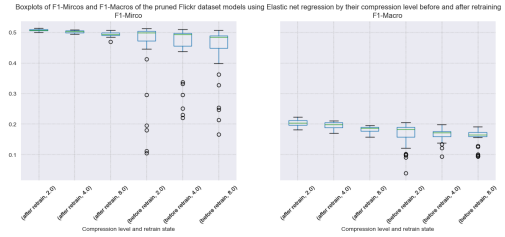
Similar to the pruned Arxiv models, the pruned Flickr models' box plots 5.2 show that the resulting F1 scores fluctuate heavily. There are numerous outliers before retraining for every F1 score, and linear regression technique used. The outliers can be related to the training batch as well, but show that the networks' accuracies are highly influenced by the selection of the training batch during pruning and can result in highly deviating results. In general, when looking at



(a) Lasso



(b) Ridge



(c) Elastic Net

Figure 5.2: Box plots of pruned **Flickr** models' F1-micro and F1-macro scores before and after training, grouped by compression level.

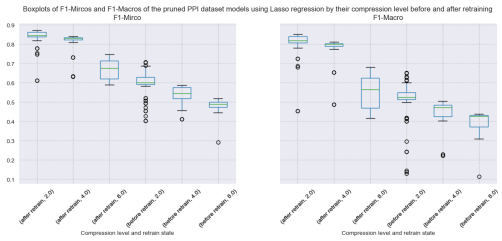
the medians, the added accuracies from retraining the network are not as large as for the Arxiv variant, but will result in far more stable results.

The pruned PPI models' box plots 5.3 show fluctuations in obtained F1 scores, especially for the compression level of 2x. This is rather interesting, but can often be attributed to spatial information getting noisy when an exact number of neighbors is present at a given inference step, therefore altering the prediction results a lot, as not enough nor not little enough information is available. This behavior even carries over to the state of the model after retraining, which was not observed for the other model variants. In general, it shows the worst expected results before and after training, irrespective of linear regression technique. As pointed out in the 4 section, the results are expected to be less good at a higher compression level, which can be observed for every variant at compression level 8x.

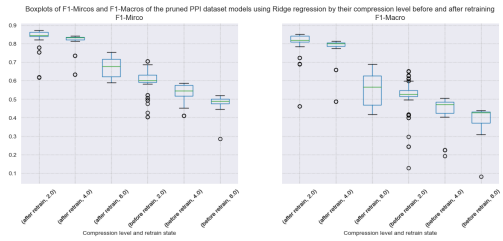
In general, the irregularities of the F1 scores often could be attributed to the choice of training sample used to compute the referential feature matrix during pruning. This is actually a desired behavior, as the procedure should perform well under any circumstances, irrespective of data coverage, such that the procedure can be applied in a timely manner. Further, this highlights the performance of the network, as input batches represent a sub-part of the graph to predict. Therefore, bad performance during pruning relates to a bad performance during inference for the original model instance as well. In numerous instances, the fluctuating performance showed the strengths of retraining the network, as even though accuracies were heavily diminished by a less good choice of training batch, they could be recovered afterward.

5.1.2 Overview of best performing Linear Regression Techniques

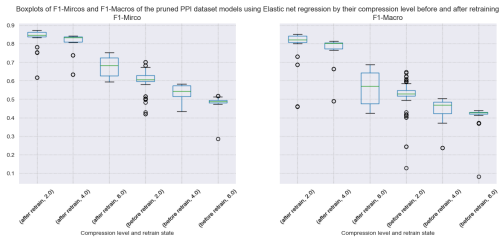
Next up was finding the best performing hyperparameter selection for the respective linear regression techniques and datasets. The results can be seen in the tables 5.1, 5.2 and 5.3. The best performing hyperparameters were attributed to settings with the best sums of F1-micro



(a) Lasso



(b) Ridge



(c) Elastic Net

Figure 5.3: Box plots of pruned **PPI** models' F1-micro and F1-macro scores before and after training, grouped by compression level.

and F1-macro scores. Meaning that both the arithmetic mean between all individual F1 scores, and the combination of all of them, performed best on average. As different settings will require different strengths in both of these quality criteria, we determined that assigning equal relevance to both of them will lead to the most satisfying results. The tables highlight the best performing technique for every setting according to the medians of achieved test scores, as it will best highlight an expected performance which is not influenced by outliers.

At first glance the results show that not Ridge regression, but Elastic Net seems to outperform Lasso regression by quite a bit. Even though the differences in median values are small, they still show an increasing effect on the quality criteria of a pruned network the higher the level of compression is. Therefore, when attempting to prune a network for the use in a resource-constrained environment, where often high numbers of feature dimensions are required to be removed, the focus has to shift to being cautious about which dimensions to remove.

Especially, when regarding the F1-macro scores, Elastic Net seems to be outperforming the other variants or at least achieves similar performance. In multi-class settings, this is a less good metric to focus on, as it treats all classes' performance the same, even when they are less relevant or occur far less often. The models trained with the Flickr dataset have been the only single-class classification problems and performed best with Elastic Net assisted pruning.

The evaluation of the other two datasets should focus on the F1-Micro performance, as the score combines all class label predictions and computes a single F1 score for all combined predictions. Here, no clear dominating method could be determined, but Elastic Net and Ridge regression appear to be the best choice in most settings.

To highlight the effectiveness of the approach in general, the best performing linear regression techniques together with their F1 scores are displayed inside table 5.4. The average losses in total F1-micro accuracies have been in the range of -0.003% - 1.3% for the Arxiv dataset, 0.9% - 2.5% for the Flickr dataset, and 4.4% - 19.3% for the PPI dataset. The performance has been good to exceptional for the first two, but applying the technique to the PPI dataset, did result in

		Compression Level		
Dataset	Lin Reg	2x	4x	8x
Arxiv (undir.)	Lasso	1.218129131	1.22219887099	**1.2063921055**
	Ridge	**1.22039800499**	1.216942713	1.1999198465
	Elastic Net	1.212288311	**1.226983601**	1.20614738
Flickr	Lasso	0.714868752	0.7046771485	0.68380842599
	Ridge	0.71427463699	0.704010476499	**0.6857077525**
	Elastic Net	**0.715773013**	**0.7049324155**	0.68409047799
PPI	Lasso	1.694294305	1.6358876025	1.349532683
	Ridge	1.6982329625	1.635660853	1.353266313
	Elastic Net	**1.699007307**	**1.6433437575**	**1.379445176**

Table 5.1: **Sum** of the **Medians** of **F1-Micro** and **F1-Macro** scores for varying linear regression techniques with optimal hyperparameters used in channel pruning models trained with the listed dataset according to their compression levels.

		Compression Level		
Dataset	Lin Reg	2x	4x	8x
Arxiv (undir.)	Lasso	0.712795506	0.71145814	0.7052119785
	Ridge	**0.716951629**	0.708042713	**0.7055698465**
	Elastic Net	0.7085673725	**0.713577351**	0.70534738
Flickr	Lasso	0.504907453	**0.5041007485**	0.4961681535
	Ridge	0.504817819	0.50358535399	**0.497579886**
	Elastic Net	**0.505288397**	0.503831847	0.494734011499
PPI	Lasso	0.8593054345	0.833639758499	0.717736722
	Ridge	0.860852257	0.8334477255	0.7193848
	Elastic Net	**0.86099160499**	**0.836799633**	**0.728930236**

Table 5.2: **Medians** of **F1-Micro** scores for varying linear regression techniques with optimal hyperparameters used in channel pruning models trained with the listed dataset according to their compression levels.

		Compression Level		
Dataset	Lin Reg	2x	4x	8x
Arxiv (undir.)	Lasso	**0.505333625**	0.510740731	**0.501180127**
	Ridge	0.503446376	0.5089	0.49435
	Elastic Net	0.5037209385	**0.51340625**	0.5008
Flickr	Lasso	0.206465579	0.2005764	0.187640272499
	Ridge	0.206902253	0.2004251225	0.1881278665
	Elastic Net	**0.207728375**	**0.2011005685**	**0.1893564665**
PPI	Lasso	0.834988870499	0.802582134	0.631795961
	Ridge	0.8373807055	0.8022131275	0.633881513
	Elastic Net	**0.838015702**	**0.8065441245**	**0.65051494**

Table 5.3: **Medians** of **F1-Macro** scores for varying linear regression techniques with optimal hyperparameters used in channel pruning models trained with the listed dataset according to their compression levels.

	Arxiv				Flickr				PPI			
Comp. Lvl.	-	2x	4x	8x	-	2x	4x	8x	-	2x	4x	8x
F1-Micro	0.7150	0.7170	0.7136	0.7056	0.5100	0.5053	0.5041	0.4976	0.9002	0.8610	0.8368	0.7289
F1-Marco	0.5052	0.5053	0.5134	0.5012	0.1987	0.2077	0.2011	0.1894	0.8873	0.8380	0.8065	0.6505

Table 5.4: Overall best achieved results for all model variants.

high drops of accuracies when a higher compression level is desired. This aligns with the overall notion of protein-based graph prediction tasks that require lots of information to reliably obtain good results. Therefore, the network suffers from removing hidden feature dimensions. Even so, it is worth noting that the GraphSAGE model comparatively already performs really well for the given task at hand, with a good F1-micro score of 0.9002. Therefore, higher losses in accuracies are expected due to pruning a highly optimized network.

Irrespective of the chosen model architecture or training dataset, there could not be a clearly dominating linear regression technique determined to use for identifying less relevant channels on the output activation with the current implementation. Therefore, different variations and hyperparameters need to be experimented with to find a properly fitting combination, such that the individual requirements of an application can be met. Nevertheless, in many settings Ridge and Elastic Net regression were able to outperform Lasso regression, especially for models of Flickr and PPI. One of the most essential findings was that retraining a network after pruning appears to be essential to obtain proper and stable performance.

5.1.3 Anomalies in Benchmark Data

When examining the benchmark results, several anomalies could be found in the measurements. For once, there have been occurrences where the pruned model was outperforming the original unpruned one. This can be observed in the Arxiv model with compression level 2x. Further, the overall performance of the different linear regression techniques were remarkably close according to their achieved F1 scores. This behavior was not expected, as experiments of section 2.4 indicated that the different linear regression techniques are at least somewhat expected to detect different

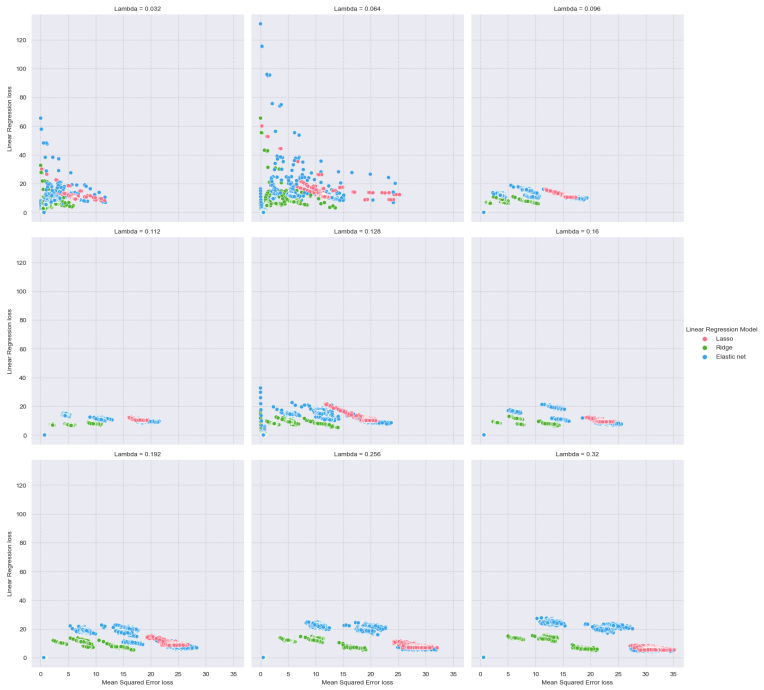


Figure 5.4: Facet grid of scatter plots displaying losses encountered during optimization epochs of **Arxiv** models with varying values for the penalization values of λ.

feature dimensions, especially in large-scale settings as the present ones. So, further investigations on the losses and influences of the hyperparameters needed to be done.

The encountered losses for the different linear regression models with varying hyperparameters can be seen in figures 5.4 and 5.5. They display facet grids with grouped scatter plots of several iterations of optimization epochs as one graph. Therefore, two pruned model variants were analyzed that were trained on the Arxiv and PPI dataset. The individual subgraphs show the MSE loss of the first part of the optimization problem as x-axis, and the linear regression identifying part of the second part of the equation, as y-axis. Therefore, they are being set into relation according to how much of the computed loss, and therefore input to the optimizer, can actually be attributed to the L1-norm, L2-norm, or a combination of both.

The graphs show that especially for small penalization parameters λ the loss is most often being dominated by the MSE loss and not the L-norms. This behavior can also present for higher λ values as well, as the pruned models of the Arxiv dataset show. During benchmarking, the pruning technique with small λ values, from 0.002 to 0.008, often produced the best results. These were incrementally increased up to 20 times by their base value during optimization epochs, therefore the λ values during the last epoch ranged from 0.04 to 0.16.

It is noticeable that the utilized pruning alternatives often did not influence the results much. As already pointed out in chapter 2.4, various linear regression techniques outperform Lasso regression in noisy settings, as many coefficients converge to zero and do not influence the computation of relevance coefficients anymore. Further, if there are multiple correlated feature dimensions, it tends to only select one, irrespective of the others' relevance. This can also be seen when observing the time it takes for the Lasso regression to converge. This signifies that

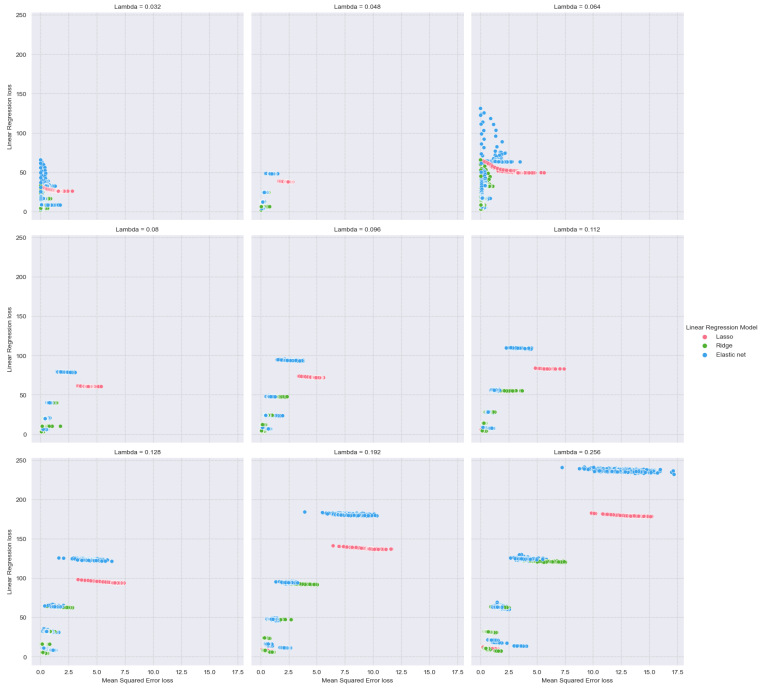


Figure 5.5: Facet grid of scatter plots displaying losses encountered during optimization epochs of **PPI** models with varying values for the penalization values of λ.

during optimization the regression loss tends to spike and does not normalize after a longer period of time, therefore the induced loss of the L1-norm is high during later optimization epochs. The magnitude of changes for the Ridge regression on the total loss of an optimization step is far smaller in almost all instances, as the values tend to stabilize after a small set of iterations because satisfying results could be found and no further altering of the β coefficient vectors is required. In almost all instances, the loss of the L2-norm in Ridge regression has been low, as well as its MSE loss, showing that it found well fitting feature dimensions to remove.

For lower values of, λ the effect of using Elastic Net regression is already significantly more present in comparison to its two counterparts. This is actually a desired behavior, as it indicates that the L1- and L2-norms optimization is playing a larger role in earlier stages of the optimization problem, therefore converging sooner to a stable level. This can also be observed when considering how low and stable the MSE loss is, when encountering high losses for the L1- and L2-norm loss.

Over all iterations and datasets, Lasso has shown the least promising results, irrespective of high or low levels of penalization. This is especially important, when considering highly noisy datasets with feature dimensions that are far less relevant than others, and not well cleaned up graph datasets that were used for the present experiments. There, higher penalization parameters should be chosen to influence the effect of the β coefficient vector on the entire optimization problem, as it signifies the influence of the computed relevance of all the feature dimensions. Hence, Lasso regression would be outperformed by Ridge and Elastic Net regression in highly penalized environments.

5.1.4 Influences of Penalization Coefficients on the Prediction Accuracy

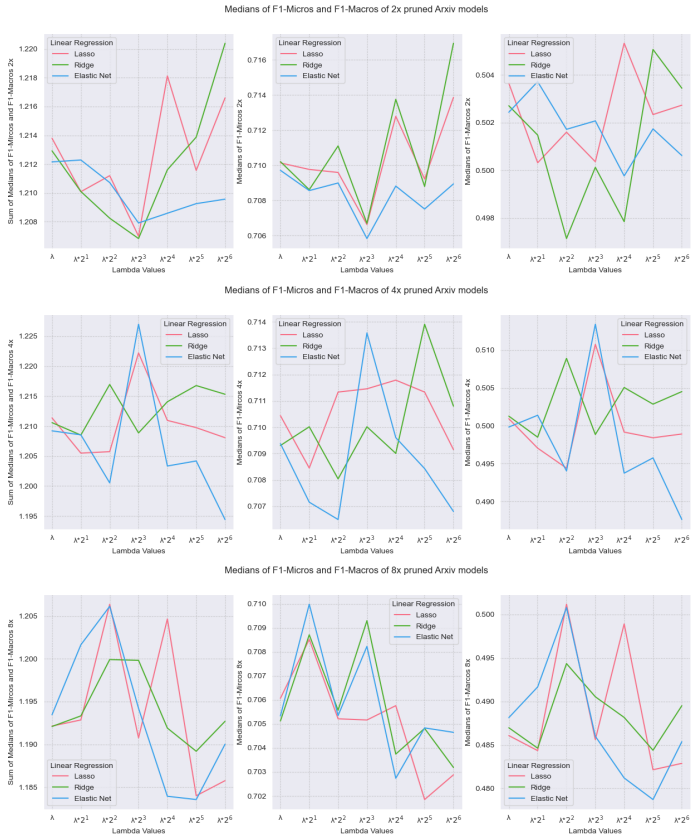


Figure 5.6: Observed medians of F1 scores for different linear regression techniques after pruning **Arxiv** models with compression levels 2x, 4x, and 8x.

Figures 5.6, 5.7, and 5.8 show the medians of observed F1 scores of the respective pruned models with varying values for the penalization coefficients of λ. As expected, Ridge regression, and Elastic Net outperformed Lasso in many instances, when training the models with varying levels of hyperparameters. As pointed out in the previous section, higher values of λ will result in a higher influence of the β coefficient vector on the result, as the MSE loss will get influenced by it more. Additionally, it has a higher influence on the penalization of badly performing feature dimensions on the output activation and adds more incentive to reduce their relevance. Therefore, we will put more focus on the parts of the drawn lines with higher λ values.

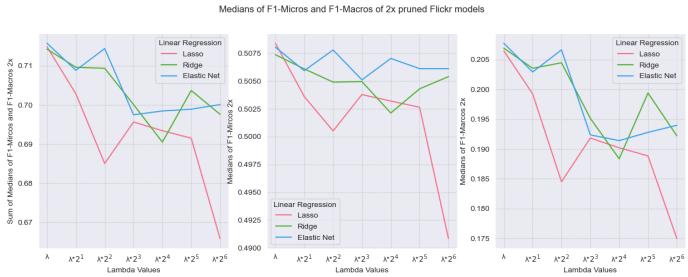
The graphs show the medians of observed F1-micro, and F1-macro scores, as well as their sum, to express a combined metric. For the Arxiv model with compression factor of 2, Ridge regression performed best in all instances, but Lasso regression came comparatively close to it. For the models pruned to be 4 times smaller, a similar tendency can be observed, as well as for the models pruned with a budget of 8. Ridge regression is the strictly dominant strategy when heavily penalizing Arxiv models, followed by Lasso regression, and lastly Elastic Net.

The results of the Flickr models have been less distinct, as no single strategy dominated. Elastic Net and Ridge regression are performing the best, with Lasso regression close behind. Depending

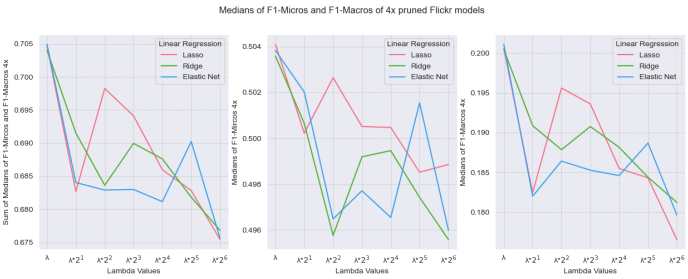
on a higher importance of F1-micro or F1-macro, developers should choose Ridge or Elastic Net for settings, where good F1-macro scores are more essential, whereas they should default to Lasso regression in settings, where F1-micro is more important.

This is irrespective of the required compression level, as the results of the upcoming section 5.2 show because most models are not solely composed of well prunable structural components. So, a trial-and-error approach should be taken, when aiming at achieving an exact level of resource efficiency.

(a) 2x



(b) 4x



(c) 8x

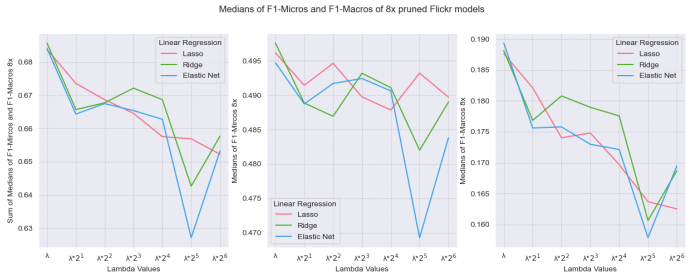


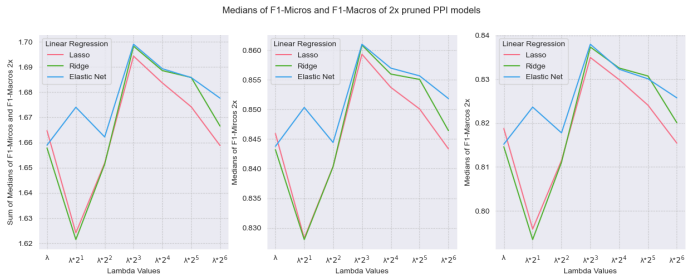
Figure 5.7: Observed medians of F1 scores for different linear regression techniques after pruning **Flickr** models with compression levels 2x, 4x, and 8x.

The results of the PPI models showed an interesting behavior, with noticeable drops in F1 scores around the λ values of 0.004-0.048. The nature of this problem could not be found. Protein classification relies on a lot of structural information for proper prediction accuracies, so the penalization levels might have resulted in pruning away elements, which appeared to be less relevant at first, but did result in bad predictions down the road. For instance, if there is a lot of information to be drawn from 2-hop neighbors, but less from 1-hop ones, it might make sense

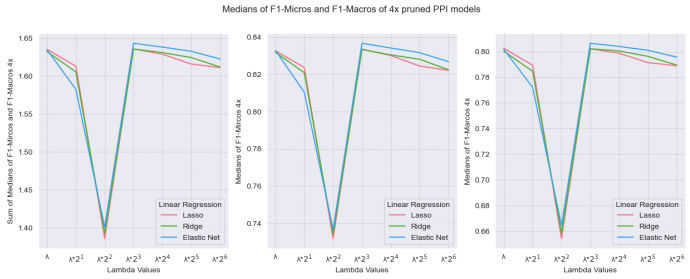
to prune them away, without noticing the influence on layers, which are investigated later on. In these cases, λ values and budgets should be defined layer-wise and pruned according to the observations.

In general, all approaches performed fairly similarly with Elastic Net being the best one, followed by Ridge regression, and Lasso regression for all tested compression levels.

(a) 2x



(b) 4x



(c) 8x

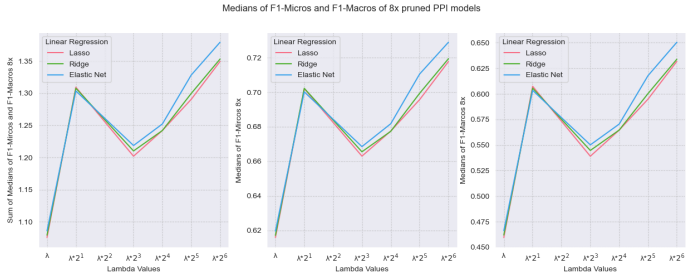


Figure 5.8: Observed medians of F1 scores for different linear regression techniques after pruning **PPI** models with compression levels 2x, 4x, and 8x.

To summarize the findings of the prior statements, the choice of linear regression technique plays a large role in obtaining good performance according to the quality criteria a prediction task may require. In single-class classification, Elastic Net appears to perform the best, whereas Ridge regression had strong performance under any penalization level and combination with any compression level. Lasso regression had the most little influence on the result, as it was often dominated by the MSE, from optimizing the first part of the optimization problem, and could be found as a less good fit in higher penalized settings and at higher compression levels.

Therefore, Ridge regression and Elastic Net have shown to be a superior choice to Lasso

	Memory Consumption in Mb			
Model Variant	-	2x	4x	8x
Arxiv	1320.14	1018.08	852.71	770.02
Flickr	351.02	263.86	220.28	198.49
PPI	471.16	359.94	304.33	276.53

Table 5.5: Memory consumption of all model variants at inference.

regression in most settings, and should especially be utilized at higher compression levels in single- and multi-class label node prediction tasks. If a lower level of compression suffices, no clear favorite strategy could be assessed, and every surveyed linear regression technique obtained good results when utilized in the adapted channel pruning approach of [6].

Lastly, the experiments indicated that networks can easily be pruned to high compression levels, i.e. 8x, and still obtain comparative accuracies to the original models. Therefore, channel pruning should and can be applied to GNNs utilized in resource-constrained devices to facilitate processing of data close to its creation, reduce privacy concerns, and data offloading to the cloud. Further, it significantly reduces the computational and memory intensity according to equations 2.4 and 2.5.

5.1.5 Memory Efficiency of Channel-Pruned Models

In this section, the observed memory consumption of a channel pruned network will be discussed according to the PyTorch's profiler toolkit[1]. The profiler analyzes allocated tensors during the execution of a model's operations during different tasks. It then outputs the amount of memory it required or the amount of time it took to perform the computation for each operation. Therefore, it was used to display the list of memory consumptions with their respective actions in descending order. As we want to analyze the compatibility with resource-constrained devices, we solely focus on the most expensive operations in the upcoming section. These could most often be attributed to the expensive matrix multiplications during computation of node embeddings - the operation this works aims to optimize. If the memory demand surpasses the hardware's threshold, no more memory can be allocated and the process terminates. In resource-constrained devices, the amount of available memory tends to be far lower than powerful environments that usually run GNN inferences. Therefore, a developer's goal is to prune the network heavily enough such that it can perform predictions, without surpassing this amount.

Firstly, no differences in memory consumption could be measured when running inferences before and after retraining, reassuring that no additional overhead gets added by optimizing the weight matrices. This comes as no surprise, as the values inside the weight matrices, which are set to zero by pruning, are omitted during the reassembling process with their input and output dimensions. Therefore, there are no dimensional differences for weight matrices of pruned models before and after retraining.

Table 5.5 shows the maximum memory consumption of the surveyed pruned models during inference with all tested compression levels.

The Arxiv model variant requires the most memory out of all the different variants with 1320 Mb - 770 Mb. This aligns with the size of the training graph, as well as the number of classes to predict. It carries the largest hidden feature matrices and therefore requires the most amount of

[1]Documentation is available at: https://pytorch.org/docs/stable/profiler.html

memory out of the examined model variants. Next up are the PPI models with 471 Mb - 276 Mb memory requirement, followed by the Flickr ones with 351 Mb - 198 Mb.

When relating this to the number of nodes and edges inside a training graph of the underlying datasets 4.1, one can observe that the number of nodes appears to dominate the memory intensity, followed by the number of edges. Training models with the Arxiv dataset, which contains over 2.4 million nodes, and over 1.1 million edges, resulted in the highest memory requirements. It is then followed by the PPI dataset with over 130,000 nodes and over 39 million edges. The Flickr dataset only contains over 105,000 nodes and over 2.3 million edges. Even though the PPI dataset and the Flickr dataset have a similar number of nodes, they have vastly different numbers of edges. This can be observed in the size of the produced networks, but does not align with the magnitude of changes in memory requirement. These appear to be dominated by the number of nodes present during the computation of node embedding.

This can also be shown in the simplified propagation function of neural networks:

$$h^i = A * W^i * h^{i-1} \tag{5.3}$$

Here, the adjacency matrix A signifies the adjacency matrix, containing an entry for every node sharing an edge with another node. The hidden feature vector hi represents the hidden representation of a node at this layerstep containing all the attributes of a node, which have been produced by the previous layer. Lastly, there is the weight matrix W^i, which contains the number of nodes of the adjacency matrix in one dimension, and the number of hidden feature dimensions in the other dimension.

The formula of matrix multiplication is shown in 2.3. Its equation is dominated by the dimensionalities of the matrices A and B and not by its contents. Therefore, the number of nodes, guiding the size of A, dominates the size and therefore the computational intensity when used in a matrix multiplication, not the number of edges. The number of edges has an influence on it though, as if there is no connection from node a to node b the computation is not required as, a multiplication by 0 will be removed from consideration in sparse matrix multiplication, which is most often being utilized inside neural networks.

Pruning the network according to a certain compression level will result in less memory consumption. But, this behavior does not scale linearly, it approaches an asymptotic limit. This can be seen in graph 5.9 as well, where the dashed lines display exemplary asymptotic limits. This indicates that there is a baseline set of required operations for running inferences, that eventually consume more memory than the expensive matrix multiplications inside the middle layers, irrespective of achieved compression level.

As already stated in section 2.3.3, the input layer of a GNN is responsible for the largest portion of the computational complexity. To reach a computational intensity below the displayed asymptotic thresholds, it would be required to preprocess the input graphs and prune the first layer as well. So far, there are only heuristics available that do not add significant computational overhead to the network's input layer. These then attempt to sample the correct number of nodes from an input graph. Further, not all heuristics are applicable for all kinds of graphs and model architectures. Therefore, we can conclude that a model can only be pruned to a given extent, when expecting consistent predictive performance without relying on the performance of heuristics.

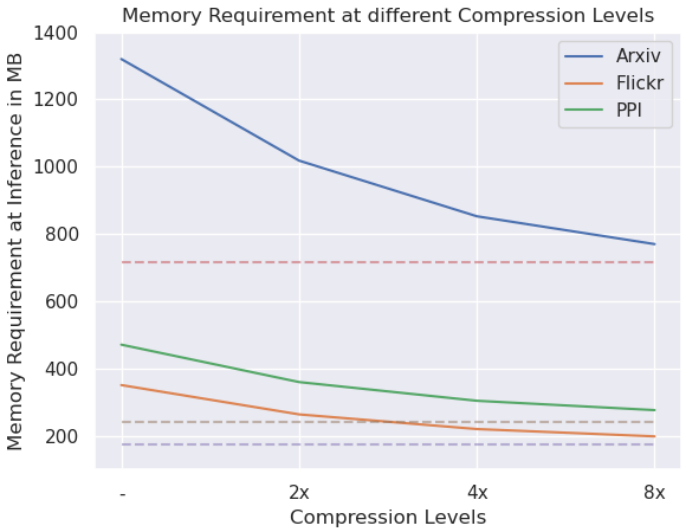


Figure 5.9: Memory requirements of different variants of GNN with varying compression levels in MB. The dashed lines show exemplary asymptotic limits.

5.2 Application of Channel Pruning to spatio-temporal Models

After confirming the hypothesis of Lasso regression getting outperformed by Ridge or Elastic Net regression when used to identify feature dimensions to remove during channel pruning, the next step was to apply the pruning approach to spatio-temporal models used in the SWAIN context.

As pointed out in section 3.4, in comparison to the homogeneously built models consisting of only one kind of layer, spatio-temporal models trained with the tsl framework often contain many differently shaped building blocks categorized as layers, blocks, and models. This facilitates building spatio-temporal GNNs, as developers can draw from already existing implementations. Thus, it eases building custom solutions in a shorter amount of time. Hence, complex neural network architectures can be experimented with, without the inherent need of developing all the required architectural elements. This reduces developmental overhead, but makes computational optimizations more cumbersome as more complex objects need to be accounted for.

Therefore, tsl models often contain multiple activation functions, encapsulated weight matrices, lots of norming and conditional layers, as well as specialized parts, such as GATs or temporal convolutional layers. Thus, there are numerous different elements to account for, when attempting to reduce the dimensionality of hidden feature vectors between middle layers. In spatio-temporal models, a popular approach is to utilize a Multilayer Perceptron to output the final label. A MLP is a simple neural network with interconnected layers and a non-linear activation function. It excels in learning from node embeddings produced by a network that are heavily contextualized. As it is trained by the output of the rest of the network, it shares similar traits with a student model in knowledge distillation, as it is trained on the output labels of the rest of the network, the teacher model. It also includes its own weight matrices.

The main predictive power of a spatio-temporal network comes from the spatial and temporal

convolutions done in some middle layers. Therefore, the input data is being processed either temporally or spatially first and then ingested into the remaining part. Hence, multiple splits of the input data are being made and processed individually according to the information they are expected to provide. A temporal convolutional operation utilizes a so-called time window, which can be regarded as a spatial operation in a temporal graph spanning a number of neighbors. It defines the number of time-steps in the past that should be considered during inference. A visual representation of this can be seen in figure 4.1. Additionally, it is common that a stride can be configured as well, which delays the consideration of a time window. This makes sense, for instance in monthly forecasts with a stride of 12, therefore the temporal data entries of a year prior would be considered.

For our experiment, we trained a spatio-temporal model trained on the LamaH dataset [87], a hydrologic dataset specifically focused on Central Europe. It is designed to provide comprehensive and detailed information about various hydrologic variables and processes in the region.

The model we experimented included the following architectural elements in order, together with their input and output dimensions:

1. **Conditional block** to condition the input on a set of node variables. (64 input & 64 output dimensions)
2. **Conditional block** to condition the input on a set of exogenous variables. (64 input & 64 output dimensions)
3. **Spatio-temporal Convolution Neural Network**:
   a) **Norming** layer (64 input & 64 output dimensions)
   b) **Temporal Convolutional Network** containing three temporal convolutional layers (64 input & 64 output dimensions)
   c) **Spatial Convolutional Network** consisting of a GAT block performing spatial convolutions (64 input & 64 output dimensions)
   d) **Dropout** only retaining the most essential values (64 input & 64 output dimensions)
4. **Multilinear layer** to transform the input into a lower dimensional form (64 input & 64 output dimensions)
5. **MLP**:
   a) **Multidense** layer with an activation unit at the end (64 input & 32 output dimensions)
   b) **Multilinear** layer to transform the dense input into a single label (32 input & 1 output dimensions)

Applying the channel-pruning approach to GATs requires significant readjustment of the algorithm. They are used to aggregate spatial information in the convolutional layers of the network. Their attention heads can conceptually be treated as branches inside a layer, be pruned on their own and reassembled [6], but no existing implementation of the approach could be found. On top, additional expertise would be required to compute the attention heads' relevance coefficient vectors β correctly according to their neighbors, such that the GAT's performance stays intact. Therefore, the main focus was put on optimizing the temporal layers and the MLP of the network.

Pruning the temporal convolutions posed a challenge, as the underlying blocks utilize a single weight matrix for all temporal dimensions. Thus, the weight matrix had to be split up for pruning and reassembled afterwards. Further, the network conditions its input variables through two

State	Test Metrics					
	Test Loss	MAE	MAPE	MRE	MSE	NSEBA
Before Optimization	6.63	6.63	3298.55	0.156	2900.32	85.73
After Pruning	47.62	47.62	231148	1.11	51899.21	6749.36
After Retraining	6.70	6.70	2224.77	0.157	2988.77	142.37

Table 5.6: Results of channel pruning the spatio-temporal model trained with the tsl framework using **Lasso** regression, in its three different states, according to different metrics.

conditional layers, which are then passed into the spatio-temporal convolutional neural network. Inside the network, it was possible to determine the relevance of the processed hidden feature vectors, but optimizing the conditioning of the input variables in the input layer was not feasible. Therefore, the computed input mask for the first temporal convolutional layer was taken to ensure the correct amount of feature dimensions.

The pre-trained model requires 11.13 MB of memory for inference. Thus, it portrays a comparatively lightweight model to the ones trained on large-scale datasets in the section 5.1. Further, the internal matrices have dimensions with a maximum of 64 in one direction, a comparatively small amount for GNNs. Therefore, the approach was solely tested with the compression level of 2x, as further compression would not result in significantly less memory intensity, as no linear downward scaling of required memory during inference can be assumed, as already pointed out in the previous section 5.1.5.

5.2.1 Effect of Channel Pruning on spatio-temporal Models

Therefore, from the original 64 feature dimensions, 32 input and output dimensions remained for the temporal convolutional operation at each time step. Further, the utilized GAT block used for spatial convolutions reduced from 64 to 32 input and output dimensions, without further evaluation of which dimensions to remove and complete retraining from the ground up. Lastly, the MLP inside the readout layer, had its dimensionalities cut in half for the multidense and multilinear layer with 32 in- and 16 output dimensions, and 16 in- and one output dimensions to produce the final label at the end.

The effect of channel pruning the spatio-temporal model using Lasso regression can be seen in table 5.6. When evaluating the goodness of hydrological models, different quality criteria are required, as different aspects need to be surveyed. Hence, the models discussed in this section are evaluated according to their MAE, MAPE, MRE, MSE, and NSEBA. The metrics are explained in section 3.1. The results show that by channel pruning the network, most qualities could be retained and just slightly got worse for the MAE, MSE, MRE, and MAPE. This displays that the overall results do not deviate from the original performance of the model too much, as the magnitudes of changes in prediction values remain small. If there would have been more outliers, by the many dropped feature dimensions from the network, the results would show higher deviations, especially in MSE, which squares observed errors, therefore quadratically increasing the error.

Further, the network even performs better according to its MAPE after retraining. It's

	Test Metrics					
State	Test Loss	MAE	MAPE	MRE	MSE	NSEBA
Before Optimization	6.63	6.63	3298.55	0.156	2900.32	85.73
After Pruning	47.50	47.5	228074	1.11	51877.55	6569.83
After Retraining	6.67	6.67	2173.72	0.156	2962.21	216.87

Table 5.7: Results of channel pruning the spatio-temporal model trained with the tsl framework using **Ridge** regression, in its three different states, according to different metrics.

particularly often used in forecasting tasks. It measures the average percentage difference between the predicted and actual values of a target variable. The high observed values indicate the prediction errors are significantly large. The MAPE metric has certain limitations, especially when it comes to small numbers close to zero. Even though the numeric value of the error in prediction values might not be big, the resulting MAPE score can grow to be extremely high or even reach infinite for values close to zero. When comparing observed values to the other metrics, it is likely that the high error is being caused by mispredicting smaller numerical values. Nevertheless, this means that these predictions perform better in the pruned model, than beforehand. This relates to the reduction of over-smoothing inside networks, as the influence of bigger, more frequent numerical values in the classification classes is being reduced.

Only the NSEBA scores perform worse, which is an essential quality criterion for hydrological models. As it assesses their predictive performance due to the observed data variability, which requires to be captured by the model. Although, it is worth mentioning that the model before pruning did not perform well according to this metric regardless.

The effects of pruning the model using Ridge and Elastic Net regression can be found in the tables 5.7 and 5.8.

Again, Lasso regression was outperformed by Ridge regression when identifying feature dimensions to prune for almost all metrics, as seen in table 5.8. The only metric where Lasso regression obtained superior results was the NSEBA, with a significantly lower value. This relates to a better interpretation of the data according to its regional significance. Therefore, the Lasso regression performed better for more popular areas, whereas the overall performance of the network was better when pruned using Ridge regression. A deployed network should perform well irrespective of where it is deployed along the river, thus which nodes of the graph a classification should be done for. Therefore, the channel-pruned network using Ridge regression should be preferred due to its generalizable application along the river ways. Lastly, the pruned model exceeded the base model in MAPE, likely related to a better prediction of small numerical values.

The results of pruning the network using Elastic Net can be seen in table 5.8. It performed the best among all produced variants for the quality criteria of MAE, MAPE, MRE, and MSE. Similar to Ridge regression, it got surpassed by Lasso regression in NSEBA. It appears that the addition of the L2-norm to the optimization problem adds capabilities, which help scores such as the F1-macro, and MSE. But it performs worse in multi-class classification settings if all classes are weighted similarly.

Another interesting observation has been the number of retraining epochs were required to regain lost accuracies from pruning. In comparison to the examined large-scale GNNs of the

State	Test Metrics					
	Test Loss	MAE	MAPE	MRE	MSE	NSEBA
Before Optimization	6.63	6.63	3298.55	0.156	2900.32	85.73
After Pruning	47.58	47.58	230059	1.11	51892.88	6685.47
After Retraining	6.66	6.66	1852.13	0.156	2937.39	299.2

Table 5.8: Results of channel pruning the spatio-temporal model trained with the tsl framework using **Elastic Net** regression, in its three different states, according to different metrics.

previous section, that required a fraction of the original training time, the pruned spatio-temporal models often required more training time than the original model itself. In the present example, the baseline model required 87 epochs to converge to its optimum, whereas the 2x-pruned models with Lasso, Ridge, and Elastic Net regression required 207, 216, and 251 epochs respectively to stabilize. When utilizing a compression level of 4x, more than 350 training epochs were required. The versions for the compression level of 8x did not converge after the full 500 epochs of training.

This points to the reduction of parameters inside the model largely reducing under higher compression levels. Therefore, the elements able to be fine-tuned are reduced, with less area for optimization being present. Nevertheless, the efficiency of the GNNs increased while retaining their performance. Therefore, longer training stages are tolerable, as they can be reused over longer periods of time without the inherent need of retraining.

To summarize the findings of this section, utilizing linear regression techniques with L2-norms to perform feature selection in channel pruning spatio-temporal GNN proved to be advantageous in comparison to alternatives solely utilizing the L1-norm. This adds to the findings of the previous section and challenges the conventional belief in Lasso regression being the preferred method for performing feature selection in regression models. Therefore, the utilization of Ridge or Elastic Net regression should be preferred over Lasso regression in channel pruning spatio-temporal networks, as the overall observed error margins were smaller, while competitive performances of the models could be preserved. Hence, the proposed hypothesis of Ridge and Elastic Net regression outperforming Lasso regression in compression tasks could be confirmed for spatio-temporal settings as well.

5.2.2 Effect on the Memory Consumption

By channel pruning the pre-trained spatio-temporal model with the budget of 2x, its total size reduced from 1.3 MB to 799 KB. When applying higher compression levels of 4x and 8x to the network, its size reduced to 676.5 KB and 644.4 KB, respectively. Therefore, higher compression levels do not lead to linear reduction in required disk space for models. Obtaining smaller model sizes is essential when keeping in mind the proposed TinyML workflow explained in section 2.2.5. When planning to deploy machine learning models on MCUs, a sequence of steps should be followed to achieve a network that has been enhanced sufficiently to be able to run on low-end devices. This sequence is finalized by burning an optimized model on the memory chip of a MCU. Typical memory sizes on MCUs range from 1 KB to 2 MB, therefore obtaining a network below 1 MB should be the goal. In the present example, the models are not expected to be

compressed to a size below 500 KB. Nevertheless, they showed sufficient efficiency such that they can theoretically be deployed MCUs.

When running inferences, the original model consumed 11.13 Mb of memory for the matrix multiplication, the most memory intensive operation. By pruning the network with a compression level of 2x, we were able to reduce the maximum memory consumption at inference to 8.61 Mb. Interestingly, this cannot be attributed to the matrix multiplication anymore, which required 8.54 Mb, but to the clamping of values inside the conditioned hidden feature vector ingested into the spatio-temporal convolutional network.

As already pointed out in section 5.1.5, the memory consumption of pruned neural networks does not reduce linearly with the reduction of internal feature dimensions, but rather approaches an asymptotic limit. Therefore, a model trained with budget 4x still uses 7.90 Mb during inference, while sacrificing significantly more accuracy. The compression to 8x showed that an asymptotic limit is close, with a memory requirement of 7.59 Mb, corresponding to a reduction of only 3.9% with high losses in performance. The loss in performance is expected, when scaling down most of the internal dimensions from 64 down to 8. Therefore, high compression levels should mostly be applied to large-scale GNNs if high accuracies are a necessity.

Alternatively, an evaluation of the network's hidden feature dimensions using Singular Value Decomposition (SVD) should be done to ensure the most expressive power of the network is being kept. It computes the power each feature dimension carries on the output and shows the influence of the removal of a dimension. When applied correctly, it can reliably identify a number of dimensions to remove without altering the expressive power of a machine learning problem much. It is closely related to the least squares solution used in many linear regression models.

The small changes in memory requirements translate to the memory consumption of other operations during GNN inference, apart from the expensive matrix multiplications. Therefore, the expected amount of memory efficiency gained under high levels of compression is low. This is due to the effect of other operations overtaking the expenses matrix multiplication. Although, we expect the 8.61 Mb of required memory, obtained with the compression of 2x, to be low enough such that the model can be executed in a resource-constrained environment.

In comparison to the large-scale models benchmarked in section 5.1, the memory to accommodate the runtime of PyTorch would demand most of the memory during inference tasks. Hence, to deploy the pruned models on MCUs, it is advised to transform them into a more efficient form like the Open Neural Network Exchange (ONNX) format [88]. It was invented to grant portability between different machine learning frameworks, and the possibility to execute multiple different models in a single environment without the requirement to install numerous dependencies. It is solely executed on the CPU of a machine and tends to accelerate a network during inference by a large margin, as undesired dependencies do not need to be accounted for during inference.

As this work focuses on algorithmic optimizations of GNNs, evaluations on memory consumption as well as overhead induced by using different runtimes falls out of the scope of it. The effectiveness of the technique has been shown for large-scale GNNs, as well as spatio-temporal models. It can significantly reduce the memory and computational requirements during inference tasks with a negligible loss in accuracy. Under specific conditions, the pruned models were even able to outperform the original models.

The proposed technique of channel pruning spatio-temporal networks has shown promising results in the reduction of computational intensity and memory requirements during inference while maintaining their competitive performance. Finding an adequate runtime environment and therefore exploring the conversion of spatio-temporal models to different formats demands further examination. This has to be done such that the performance gains are not made irrelevant by the inherent requirements of a runtime environment, for instance PyTorch.

When aiming to deploy GNNs in resource-constrained environments, the proposed channel

pruning technique should be applied in addition to other optimization approaches according to the proposed TinyML workflow explained in section 2.2.5. Next to pruning, we presume converting the network to the standardized ONNX format as the most essential step to take. To reaffirm the strong belief in sufficient level of optimization, the obtained models should then be deployed in the SWAIN project's setting.

6 Conclusion

In this work, the topic of channel pruning Graph Neural Network (GNN)s was explored and evaluated according to the linear regression techniques used in them to identify channels to remove. Special focus was put on comparing three of the most popular linear regression methods in Lasso regression, Ridge regression, and Elastic Net regression. Additionally, the differing techniques were applied to spatio-temporal models of the Torch Spatiotemporal library as well, with the aim to assess their performance in resource-constrained environments.

Throughout the thesis, a comprehensive analysis of channel pruning GNNs and its significance in various applications was done. Therefore, the importance of model compression and optimization was highlighted to overcome the limitations of resource intensive GNN inference. Particularly, scenarios with limited computational resources require further optimization to run prediction tasks. Channel pruning showed to be a promising approach to remove redundant feature dimensions in middle layers of neural networks. This reduces the model complexity and enhances its efficiency without considerable loss in accuracy.

The evaluation of the benchmarks of linear regression techniques used in channel pruning GNNs revealed compelling insights. The experiments were performed with a range of different large-scale datasets, considering different settings and dimensionalities of the input data. The interpretation of the results revealed that Lasso regression consistently gets outperformed by both Ridge and Elastic Net regression in multi-class classification tasks. They especially yielded better prediction accuracies and stability in highly penalized environments, and under high compression levels.

This revelation challenges the conventional view of Lasso regression being a superior choice when it comes to feature selection and sparsifying feature vectors. The findings suggest that the inclusion of L2 regularization, as provided by Elastic Net and Ridge regression, comes with better stabilization of results in later optimization epochs as feature dimensions are not omitted too soon. Further, they showed a faster convergence, which is attractive for extensively large-scale graph datasets that exceed the size of a few GBs. Further, when high F1-macro scores are required in multi-class multi-label classification tasks, using linear regression models with the L2 regularization appears to be the clear dominating choice.

Furthermore, the channel pruning technique has successfully been applied to spatio-temporal models of the tsl library. It is used in the SWAIN project for analyzing and modeling spatio-temporal data of waterways, with intents to deploy trained models close to the network's edge on resource-constrained devices. Therefore, requiring more lightweight models for running inferences, and thus making it a viable candidate for the evaluation of the approach. Hence, spatio-temporal models have been trained and fine-tuned using different linear regression and hyperparameters. The results showcased the effectiveness of channel pruning in improving the efficiency of spatio-temporal models, without significant reduction of their predictive capabilities.

The success of the experiments highlights the potential of channel pruning spatio-temporal models. By selectively removing redundant channels, in the intermediary middle layers, the approach was able to reduce the computational intensity of the networks during inference while maintaining high accuracy and stability in prediction and simultaneously reducing memory intensity. On top, some instances were even able to outperform the original baseline models through reduction of overfitting inside the network. These observations have important implications for

real-world applications of spatio-temporal models where computational resources are limited, or latency sensitive predictions need to be made.

It is worth mentioning that our evaluation and findings are not without limitations. While extensive experiments have been conducted on diverse datasets, there is always a possibility of dataset-specific biases and unusual characteristics. Additionally, the choice of the three linear regression techniques was based on their popularity and ease of application. Other regression techniques might yield different results and should be investigated in future studies. Further, the benchmarks were limited to models of GraphSAGE and the tsl library, and the generalizability of the findings depends on how transferable they are to different libraries and frameworks.

To further extend this research, future investigations could explore other aspects of channel pruning in tsl. For instance, exploring different pruning algorithms, further investigating the impact of different hyperparameters on model performance. Additionally, coming up with an efficient way to sample the input graph of the network such that the input layer can be pruned sufficiently, to further reduce memory and computational intensities, appears necessary. Also, studying the transferability of channel pruning techniques across different GNN architectures and tasks, like GATs, would provide a broader understanding of their applicability.

Furthermore, the efficiency of using channel pruning in the proposed TinyML deep compression workflow should be validated and obtained models should be deployed on Microcontroller Unit (MCU)s to fully approve of the theoretical proven usability of GNNs in resource-constrained environments. Therefore, conversion of model formats and different runtime engines should be considered.

In conclusion, this master thesis provided a comprehensive analysis of channel pruning GNNs, evaluated linear regression techniques used to identify feature dimensions to remove, and successfully applied the technique to spatio-temporal models of the tsl library. The findings demonstrated the superiority of Ridge and Elastic Net regression over Lasso regression, challenging the conventional belief in feature selection tasks. Moreover, the successful application of channel pruning in spatio-temporal models highlighted its potential in improving efficiency without sacrificing predictive accuracy while simultaneously reducing memory intensity. This research contributes to the growing body of knowledge in GNN optimization.

Bibliography

[1] X. Liu, "Survey on graph neural network acceleration: An algorithmic perspective," *Retrieved from: https://arxiv.org/abs/2202.04822*, 2022. Last accessed 17 Feb 2023.

[2] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," *CoRR*, vol. abs/2005.00687, 2020.

[3] Y. Yu, X. Si, C. Hu, and J. Zhang, "A review of recurrent neural networks: Lstm cells and network architectures," *Neural Computation*, vol. 31, no. 7, pp. 1235–1270, 2019.

[4] Zhou, Zhi, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Retrieved from: https://ieeexplore.ieee.org/document/8736011*, 2019.

[5] Dutta, Lachit, "Tinyml meets iot: A comprehensive survey," *Retrieved from: https://www.sciencedirect.com/science/article/abs/pii/S2542660521001025*, 2021.

[6] H. Zhou, A. Srivastava, H. Zeng, R. Kannan, and V. Prasanna, "Accelerating large scale real-time gnn inference using channel pruning," *Proc. VLDB Endow.*, vol. 14, p. 1597–1605, may 2021.

[7] R. C. Fair, "A Theory of Extramarital Affairs," *Journal of Political Economy*, vol. 86, pp. 45–61, February 1978.

[8] G. Garrison, S. Kim, and R. L. Wakefield, "Success factors for deploying cloud computing," *Commun. ACM*, vol. 55, p. 62–68, sep 2012.

[9] Carreira, Joao, "A case for serverless machine learning," *Retrieved from: https://andrewmzhang.com/assets/pdf/cirrus_ nips_ final2.pdf*, 2018.

[10] N. Hassan, S. Gillani, E. Ahmed, I. Yaqoob, and M. Imran, "The role of edge computing in internet of things," *IEEE Communications Magazine*, vol. 56, no. 11, pp. 110–115, 2018.

[11] A. Aral and T. Ovatman, "A decentralized replica placement algorithm for edge computing," *IEEE Transactions on Network and Service Management*, vol. 15, no. 2, pp. 516–529, 2018.

[12] J. Zhang, B. Chen, Y. Zhao, X. Cheng, and F. Hu, "Data security and privacy-preserving in edge computing paradigm: Survey and open issues," *IEEE Access*, vol. 6, pp. 18209–18237, 2018.

[13] Satyanarayanan, Mahadev, "The emergence of edge computing," *Retrieved from: https://ieeexplore.ieee.org/document/7807196*, 2017.

[14] Brady, "Edge computing changes everything," *Retrieved from: https://www.ibm.com/blogs/industries/rob-high-edge-computing*, 2019.

[15] SWAIN, "Swain project website," *Retrieved from: http://swain-project.eu*, 2021. Last accessed 13 May 2021.

[16] T. Aittokallio and B. Schwikowski, "Graph-based methods for analysing networks in cell biology," *Briefings in Bioinformatics*, vol. 7, pp. 243–255, 09 2006.

[17] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," *Retrieved from: https://arxiv.org/abs/1707.06168*, 2017. Last accessed 20 Feb 2023.

[18] Hamilton, "Inductive representation learning on large graphs," *Retrieved from: https://arxiv.org/abs/1706.02216*, 2017. Last accessed 17 Feb 2023.

[19] Rong, "Dropedge: Towards deep graph convolutional networks on node classification," *Retrieved from: https://arxiv.org/abs/1907.10903*, 2019. Last accessed 17 Feb 2023.

[20] Srinivasa, "Fast graph attention networks using effective resistance based graph sparsification," *Retrieved from: https://arxiv.org/abs/2006.08796*, 2020. Last accessed 17 Feb 2023.

[21] Chen, "A unified lottery ticket hypothesis for graph neural networks," *Retrieved from: https://arxiv.org/abs/2102.06790*, 2020. Last accessed 17 Feb 2023.

[22] He, "Lightgcn: Simplifying and powering graph convolution network for recommendation," *Retrieved from: https://arxiv.org/abs/2002.02126*, 2020. Last accessed 17 Feb 2023.

[23] Mao, "Ultragcn: Ultra simplification of graph convolutional networks for recommendation," *Retrieved from: https://arxiv.org/abs/2110.15114*, 2021. Last accessed 17 Feb 2023.

[24] Yan, "Tinygnn: Learning efficient graph neural networks," *Retrieved from: https://dl.acm.org/doi/10.1145/3394486.3403236*, 2020. Last accessed 17 Feb 2023.

[25] I. M. Andrea Cini, "Torch spatiotemporal - neural spatiotemporal forecasting with pytorch." `https://torch-spatiotemporal.readthedocs.io/en/latest/`, 2023. Accessed: May 5, 2023.

[26] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. K. Prasanna, "Graphsaint: Graph sampling based inductive learning method," *CoRR*, vol. abs/1907.04931, 2019.

[27] Y. Zhang, X. Wang, X. Jiang, C. Shi, and Y. Ye, "Hyperbolic graph attention network," *CoRR*, vol. abs/1912.03046, 2019.

[28] Cai, Tianle, "Graphnorm: A principled approach to accelerating graph neural network training," *Retrieved from: https://arxiv.org/abs/2009.03294*, 2020.

[29] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," 2018.

[30] A. Y. Ding, E. Peltonen, T. Meuser, A. Aral, C. Becker, S. Dustdar, T. Hiessl, D. Kranzlmuller, M. Liyanage, S. Magshudi, N. Mohan, J. Ott, J. S. Rellermeyer, S. Schulte, H. Schulzrinne, G. Solmaz, S. Tarkoma, B. Varghese, and L. Wolf, "Roadmap for edge ai: A dagstuhl perspective," 2021.

[31] Cisco, "Ecisco global cloud index: Forecast and methodology," *Retrieved from: https://time.com/wp-content/uploads/2015/05/cloud_ index_ white_ paper.pdf*, 2016-2021.

[32] Gartner, "What edge computing means for infrastructure and operations leaders," *Retrieved from: https://www.gartner.com/smarterwithgartner/what-edge-computing-means-for-infrastructure-and-operations-leaders*, 2018. Last accessed 13 May 2021.

[33] A. Aral, A. Esposito, A. Nagiyev, S. Benkner, B. D. Martino, and M. A. Bochicchio, "Experiences in architectural design and deployment of ehealth and environmental applications for cloud-edge continuum," in *Advanced Information Networking and Applications - Proceedings of the 37th International Conference on Advanced Information Networking and Applications (AINA-2023), Juiz de Fora, Brazil, 29-31 March 2023, Volume 3* (L. Barolli, ed.), vol. 655 of *Lecture Notes in Networks and Systems*, pp. 136–145, Springer, 2023.

[34] Han, Song, "Learning both weights and connections for efficient neural network," *Retrieved from: https://arxiv.org/abs/1506.02626*, 2015.

[35] Han, Song, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *Retrieved from: https://arxiv.org/abs/1510.00149*, 2015.

[36] Dai, Bin, "Compressing neural networks using the variational information bottleneck," *Retrieved from: https://arxiv.org/abs/1802.10399*, 2020.

[37] Kang, Yiping, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *Retrieved from: https://dl.acm.org/doi/10.1145/3037697.3037698*, 2017.

[38] Ko, Jong Hwan, "Edge-host partitioning of deep neural networks with feature space encoding for resource-constrained internet-of-things platforms," *Retrieved from: https://ieeexplore.ieee.org/document/8639121*, 2018.

[39] Li, Chuang, "Ionn: Incremental offloading of neural network computations from mobile devices to edge servers," *Retrieved from: https://dl.acm.org/doi/10.1145/3267809.3267828*, 2018.

[40] Hu, "Dynamic adaptive dnn surgery for inference acceleration on the edge," *Retrieved from: https://ieeexplore.ieee.org/document/8737614*, 2019.

[41] Mao, Jiachen, "Modnn: Local distributed mobile computing system for deep neural network," *Retrieved from: https://ieeexplore.ieee.org/document/7927211*, 2017.

[42] Teerapittayanon, Surat, "Distributed deep neural networks over the cloud, the edge and end devices," *Retrieved from: https://arxiv.org/abs/1709.01686*, 2016.

[43] Teerapittayanon, Surat, "Distributed deep neural networks over the cloud, the edge and end devices," *Retrieved from: https://ieeexplore.ieee.org/document/7979979*, 2017.

[44] Li, En, "Edge intelligence: On-demand deep learning model co-inference with device-edge synergy," *Retrieved from: https://arxiv.org/abs/1806.07840*, 2018.

[45] Lo, Chi, "A dynamic deep neural network design for efficient workload allocation in edge computing," *Retrieved from: https://ieeexplore.ieee.org/abstract/document/8119222*, 2017.

[46] Yu-Han Chen, Tiffany, "Glimpse: Continuous, real-time object recognition on mobile devices," *Retrieved from: http://people.csail.mit.edu/yuhan/doc/sen060-chenA.pdf*, 2015.

[47] Venugopal, Srikumar, "Shadow puppets: Cloud-level accurate ai inference at the speed and economy of edge," *Retrieved from: https://www.usenix.org/conference/hotedge18/presentation/venugopal*, 2018.

[48] Utsav, Drolia, "Precog: Prefetching for image recognition applications at the edge," *Retrieved from: https://dl.acm.org/doi/abs/10.1145/3132211.3134456*, 2017.

[49] Kang, Daniel, "Noscope: Optimizing neural network queries over video at scale," *Retrieved from: https://www.vldb.org/pvldb/vol10/p1586-kang.pdf*, 2017.

[50] Zhang, Chen, "Ffs-va: A fast filtering system for large-scale video analytics," *Retrieved from: https://dl.acm.org/doi/10.1145/3225058.3225103*, 2018.

[51] Jain, Samvit, "Rexcam: Resource-efficient, cross-camera video analytics at enterprise scale," *Retrieved from: https://arxiv.org/abs/1811.01268*, 2017.

[52] Park, Eunhyeok, "Big/little deep neural network for ultra low power inference," *Retrieved from: https://ieeexplore.ieee.org/document/7331375*, 2015.

[53] Taylor, Ben, "Adaptive deep learning model selection on embedded systems," *Retrieved from: https://dl.acm.org/doi/10.1145/3211332.3211336*, 2018.

[54] Stamoulis, Dimitrios, "Designing adaptive neural networks for energy-constrained image classification," *Retrieved from: https://arxiv.org/abs/1808.01550*, 2018.

[55] Hinton, Geoffrey, "Distilling the knowledge in a neural network," *Retrieved from: https://arxiv.org/pdf/1503.02531.pdf*, 2015.

[56] Romano, Nathanael, "Distilling knowledge to specialist networks for clustered classification," *Retrieved from: http://cs231n.stanford.edu/reports/2016/pdfs/120_Report.pdf*, 2015.

[57] Han, Song, "Learning both weights and connections for efficient neural networks," *Retrieved from: https://arxiv.org/abs/1506.02626*, 2015.

[58] Guo, Yiwen, "Dynamic network surgery for efficient dnns," *Retrieved from: https://arxiv.org/abs/1608.04493*, 2016.

[59] Polino, Antonio, "Model compression via distillation and quantization," *Retrieved from: https://arxiv.org/pdf/1802.05668.pdf*, 2018.

[60] David, Robert, "Tensorflow lite micro: Embedded machine learning on tinyml systems," *Retrieved from: https://arxiv.org/pdf/2010.08678.pdf*, 2020.

[61] R. J. Townshend, M. Vögele, P. Suriana, A. Derry, A. Powers, Y. Laloudakis, S. Balachandar, B. Anderson, S. Eismann, R. Kondor, R. B. Altman, and R. O. Dror, "Atom3d: Protein-protein interactions (ppi) dataset," June 2021.

[62] Y. A. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," *Advances in neural information processing systems*, vol. 2, pp. 598–605, 1990.

[63] S. Chen and Z. Lu, "Hardware acceleration of multilayer perceptron based on inter-layer optimization," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pp. 164–172, 2019.

[64] J. Redmon, "Darknet: Open source neural networks in c," *http://pjreddie.com/darknet/*, 2013.

[65] G. J. Wong and W. Koehrsen, "Yolov5: A universal object detector," *arXiv preprint arXiv:2104.14134*, 2021.

[66] A. Cini and I. Marisca, "Torch Spatiotemporal," 3 2022.

[67] Y. Li, R. Yu, C. Shahabi, and Y. Liu, "Diffusion convolutional recurrent neural network: Data-driven traffic forecasting," 2018.

[68] F. Santosa and W. W. Symes, "Linear inversion of band-limited reflection seismograms," *SIAM Journal on Scientific and Statistical Computing*, vol. 7, no. 4, pp. 1307–1330, 1986.

[69] A. E. Hoerl and R. W. Kennard, "Ridge regression: Biased estimation for nonorthogonal problems," *Technometrics*, vol. 12, pp. 55–67, 1970.

[70] H. Zou and T. Hastie, "Regularization and variable selection via the elastic net," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 67, no. 2, pp. 301–320, 2003.

[71] C. F. Gauss, "Method of least squares," *Memoirs of the Royal Society of Göttingen*, vol. 7, pp. 1–24, 1823.

[72] X. Yue, Z. Zhang, and Q. Yang, "Associative knowledge graphs: A framework for integrating heterogeneous knowledge sources with neural networks," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 3689–3699, 2019.

[73] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[74] H. Drucker, D. Wu, and V. Vapnik, "Support vector regression machines," *Advances in Neural Information Processing Systems*, vol. 9, pp. 155–161, 1997.

[75] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 58, no. 1, pp. 267–288, 1996.

[76] R. A. Fisher, "Statistical methods for research workers," *Oliver and Boyd*, 1925.

[77] T. Hastie, R. Tibshirani, and M. Wainwright, *Statistical Learning with Sparsity: The Lasso and Generalizations*. CRC press, 2015.

[78] L. Freijeiro-González, M. Febrero-Bande, and W. González-Manteiga, "A critical review of lasso and its derivatives for variable selection under dependence among covariates," 2020.

[79] H. Zou and T. Hastie, "Regularization and variable selection via the elastic net," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 67, no. 2, pp. 301–320, 2005.

[80] S. Seabold and J. Perktold, "statsmodels: Econometric and statistical modeling with python," in *9th Python in Science Conference*, 2010.

[81] M. Grandini, E. Bagli, and G. Visani, "Metrics for multi-class classification: an overview," *arXiv preprint arXiv:2008.05756*, 2020.

[82] J. Wang, J. Yang, K. Yu, F. Lv, and T. S. Huang, "Image labeling on a network: using social-network metadata for image classification," *Proceedings of the 18th ACM international conference on Multimedia*, pp. 771–774, 2010.

[83] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[84] T. pandas development team, "pandas-dev/pandas: Pandas," Feb. 2020.

[85] M. L. Waskom, "seaborn: statistical data visualization," *Journal of Open Source Software*, vol. 6, no. 60, p. 3021, 2021.

[86] J. W. Tukey, "Exploratory data analysis," *Addison-Wesley*, 1977.

[87] C. Klingler, K. Schulz, and M. Herrnegger, "Lamah-ce: Large-sample data for hydrology and environmental sciences for central europe," *Earth System Science Data*, vol. 13, no. 9, pp. 4529–4565, 2021.

[88] ONNX Community, "Onnx: Open neural network exchange." `https://onnx.ai/`. Accessed: June 10, 2023.

Acronyms

AI Artificial Intelligence. 2, 15, 17

CNN Convolutional Neural Network. 9, 12, 13, 18, 26

DNN Deep Neural Network. ix, 11, 12, 15–20

EI Edge Intelligence. v, ix, 2, 3, 7, 15–19, 21, 26

GAT Graph Attention Network. 4, 5, 10, 12–15, 24, 54, 68–70, 76

GCN Graph Convolutional Network. 4–6, 13, 53, 54

GNN Graph Neural Network. i, iii, ix, x, 2–7, 10–14, 22–29, 36, 41, 45, 48, 55, 66–68, 70–73, 75, 76, 87

IoT Internet of Things. 1, 2, 15, 20, 21, 86

LSTM Long Short-Term Memory. 10

MAC Multiply–accumulate. 23, 26

MAE Mean Absolute Error. 43, 70–72

MAPE Mean Absolute Percentage Error. 43, 70–72

MCU Microcontroller Unit. ix, 21, 22, 72, 73, 76

MLP Multilayer Perceptron. 68–70

MRE Mean Relative Error. 43, 70–72

MSE Mean Squared Error. 10, 32, 33, 43, 45, 46, 48, 61–63, 65, 70–72, 86

NSEBA Normalized Squared Error Basin Average. 44, 70–72

OGB Open Graph Benchmark. 51, 55

OLS Ordinary Least Squares. vii, 33–37, 44, 45

ONNX Open Neural Network Exchange. 73, 74

PPI Protein-Protein Interactions. x, 24, 51–55, 57–62, 64–67

RMSE Root Mean Squared Error. 43

RNN Recurrent Neural Network. 10, 85

tsl Torch Spatiotemporal. i, v, vii, 10, 28, 29, 41, 46, 48, 55, 68, 70–72, 75, 76

UGS Unified GNN Sparsification. 5, 7

Glossary

Artificial Intelligence refers to the development of machine learning models to perform prediction tasks that require learning contextualized interrelations of a given dataset. Typically these tasks require human intelligence to solve, such as speech recognition, complex problem-solving, or object detection. 2, 83

Convolutional Neural Network is a type of neural network architecture that applies convolutional operations on the input data to extract spatial information and transform them input higher level information. 9, 83

Deep Neural Network refers to a super class of neural network architecture consisting of multiple hidden layers between the input and output layers. Each layer adds to the context of the current input, therefore enabling the model to learn high-level information from the data. 11, 83

Edge Intelligence describes the ongoing pursuit for processing and analyzing data closer to the network's edge on low-end devices. It reduces privacy concerns, allows for latency-sensitive processing, and reduces the reliance on cloud data centers to process data. v, ix, 2, 3, 7, 15–17, 19, 21, 83

Graph Attention Network is a graph neural neural network architecture that utilizes the attention mechanism to learn node embeddings by assigning different relevance spatially close neighbors. 4, 10, 83

Graph Convolutional Network is a type of neural network architecture designed specifically to operate on graph-structured data. To perform its predictions it uses spatial information through convolutional operations. 4, 13, 83

Graph Neural Network is a type of neural network architecture that works on graph-structured data. It is designed to learn higher level information by considering relationships between interconnected nodes in a graph. i, 2, 11, 75, 83

Internet of Things describes the network of interconnected low-end ubiquitous devices with sensors, own processing software, and connection to the internet to exchange data. 1, 83

Long Short-Term Memory is an element often used in RNN architectures to handle the way information should be retained when propagating through the network. It is used to contextualize a sequence of information. Therefore, it contains different kinds of memory cells supporting the gating mechanism to selectively keep or forget information over time. 10, 83

Mean Absolute Error is a metric used to evaluate machine learning models by assessing the mean difference between the absolute value of predicted and expected values. 43, 83

Mean Absolute Percentage Error is a metric used to assess the performance of machine learning models by determining the average percentage difference between the absolute expected and predicted values. It is useful in models that deal with many values of different sizes. 43, 83

Mean Relative Error is a metric used in regression models to determine the mean relative difference between expected and predicted absolute values in percentages. 43, 83

Mean Squared Error measures the average squared difference between predicted and actual values of machine learning models to evaluate its performance. Can reach high values if the magnitude of the error is large. 32, 43, 83

Microcontroller Unit is a small computer on a single circuit board that contains a processor core, memory and programmable input/output peripherals. It is commonly used in embedded systems or on IoT devices to process information. 21, 76, 83

Multilayer Perceptron is a simple type of neural network composed of multiple layers with interconnected neuronal units to perform supervised learning tasks like classification or regression. It excels in learning complex non-linear data relationships and generally has a wide range of applications. 68, 83

multiply–accumulate is a fundamental operation in computing tasks involving multiplying two values and adding their results by using an accumulator. It is crucial to assess the performance of hardware and software. 23, 83

Normalized Squared Error Basin Average is a metric used to assess the quality of machine learning models by calculating the average squared error between predicted and actual values according to the normalized size of the possible basins. It describes how well the model performs on unseen data, as it takes into account the convergence behavior of training samples and therefore explains the assumed variance and bias in the dataset. Additionally, it evaluates how well a machine learning model assigns its values to regions in the data. 44, 83

Open Graph Benchmark is a standardized benchmarking platform providing graph datasets used to evaluate the performance of machine learning models aimed at solving graph-related tasks, such as node classification or link prediction. 51, 83

Open Neural Network Exchange is an open-source format to enable interoperability between different deep learning frameworks, such that a standardized way of deploying and exchanging them exist. It is used as a conversion format after producing a model and comes with a lightweight runtime environment, that does not include dependencies to train the networks. 73, 83

Ordinary Least Squares is a method used to estimate the parameters of a linear regression model by minimizing the sum of squared differences between the predicted and expected values. A popular method to solve it is to optimize the MSE of the model. 33, 44, 83

Protein-Protein Interactions is a large-scale graph dataset that models the physical interaction between proteins. These determine their functional relationships, biological processes and chemical properties. 24, 51, 83

Recurrent Neural Network is a type of neural network architecture that is designed to process sequential data with a temporal context. It therefore enables prediction tasks to contextualize current predictions with prior seen values. 10, 83

Root Mean Squared Error is a metric to evaluate the accuracy of regression models by measuring the square root of the average squared difference between predicted and expected values. Therefore, giving larger errors more influence on the result. 43, 83

Torch Spatiotemporal is a PyTorch framework that provides tools and functionalities to develop spatio-temporal models. Therefore, it mostly focuses on the development GNNs. i, v, 28, 29, 75, 83

Unified GNN Sparsification is an algorithmic acceleration framework specifically aimed at GNNs. It can identify sub-networks inside a GNN by applying the lottery ticket hypothesis that perform similar to the larger network while reducing computational complexity and memory requirements. 83

7 Appendix

In the following code snippets of the linear regression models used during channel pruning can be found:

```
import torch
from torch import nn
from linear_regression_model import LinearRegressionModel

class Lasso(LinearRegressionModel):
    def __init__(self, dim_in, weight, lmbd_1, lmbd_2, lmbd_1_step,
                 lmbd_2_step, beta_lr, weight_lr, object_with_weight):
        LinearRegressionModel.__init__(self, dim_in, weight, lmbd_1,
    lmbd_2, lmbd_1_step,
                 lmbd_2_step, beta_lr, weight_lr, object_with_weight)

    def optimize_beta(self, inputs, ref):
        self.beta_optimizer.zero_grad()
        self.beta.requires_grad = True
        self.weight.requires_grad = False
        out = self(inputs)
        loss_beta = nn.MSELoss()(out, ref)
        loss_beta += self.lmbd_1 * torch.linalg.norm(self.beta, dim=0,
    ord=1)
        loss_beta.backward(retain_graph=True)
        self.beta_optimizer.step()
        return loss_beta

class Ridge(LinearRegressionModel):
    def __init__(self, dim_in, weight, lmbd_1, lmbd_2, lmbd_1_step,
                 lmbd_2_step, beta_lr, weight_lr, object_with_weight):
        LinearRegressionModel.__init__(self, dim_in, weight, lmbd_1,
    lmbd_2, lmbd_1_step,
                 lmbd_2_step, beta_lr, weight_lr, object_with_weight)

    def optimize_beta(self, inputs, ref):
        self.beta_optimizer.zero_grad()
        self.beta.requires_grad = True
        self.weight.requires_grad = False
        out = self(inputs)
        loss_beta = nn.MSELoss()(out, ref)
        loss_beta += self.lmbd_2 * torch.pow(torch.linalg.norm(self.beta,
     dim=0, ord=2), 2)
        loss_beta.backward(retain_graph=True)
        self.beta_optimizer.step()
        return loss_beta

class ElasticNet(LinearRegressionModel):
```

```
40      def __init__(self, dim_in, weight, lmbd_1, lmbd_2, lmbd_1_step,
41                   lmbd_2_step, beta_lr, weight_lr, object_with_weight):
42          LinearRegressionModel.__init__(self, dim_in, weight, lmbd_1,
    lmbd_2, lmbd_1_step,
43                   lmbd_2_step, beta_lr, weight_lr, object_with_weight)
44
45      def optimize_beta(self, inputs, ref):
46          self.beta_optimizer.zero_grad()
47          self.beta.requires_grad = True
48          self.weight.requires_grad = False
49          out = self(inputs)
50          loss_beta = nn.MSELoss()(out, ref)
51          loss_beta += self.lmbd_1 * torch.linalg.norm(self.beta, dim=0,
    ord=1)
52          loss_beta += self.lmbd_2 * torch.pow(torch.linalg.norm(self.beta,
     dim=0, ord=2), 2)
53          loss_beta.backward(retain_graph=True)
54          self.beta_optimizer.step()
55          return loss_beta
```

Listing 7.1: Implementation for Linear Regression Models.

```
1  import torch
2  from torch import nn
3
4  class LinearRegressionModel(nn.Module):
5      def __init__(self, dim_in, weight, lmbd_1, lmbd_2, lmbd_1_step,
6                   lmbd_2_step, beta_lr, weight_lr, object_with_weight):
7          super(LinearRegressionModel, self).__init__()
8          self.beta = nn.Parameter(torch.ones(dim_in))
9          self.weight = nn.Parameter(torch.zeros(weight.shape))
10         self.weight.data.copy_(weight.data)
11         self.weight_norm_factor = torch.norm(self.weight, p='fro')
12         self.params = nn.ParameterList([self.beta, self.weight])
13         self.beta = self.params[0]
14         self.weight = self.params[1]
15         self.lmbd_1 = lmbd_1
16         self.lmbd_2 = lmbd_2
17         self.lmbd_1_step = lmbd_1_step
18         self.lmbd_2_step = lmbd_2_step
19         self.beta_optimizer = torch.optim.Adam([self.beta], lr=beta_lr)
20         self.weight_optimizer = torch.optim.Adam([self.weight], lr=
    weight_lr)
21         self.object_with_weight = object_with_weight
22
23     def forward(self, inputs):
24         return self.object_with_weight(inputs)
25
26     #The base class for linear regression will only optimize the ordinary
     least squares loss for determening the beta coefficient vector
27     def optimize_beta(self, inputs, ref):
28         self.beta_optimizer.zero_grad()
29         self.beta.requires_grad = True
30         self.weight.requires_grad = False
```

```
        out = self(inputs)
        loss_beta = nn.MSELoss()(out, ref)
        loss_beta.backward()
        self.beta_optimizer.step()
        return loss_beta

    def lmbd_step(self):
        self.lmbd_1 += self.lmbd_1_step
        self.lmbd_2 += self.lmbd_2_step

    def clip_beta(self, budget):
        with torch.no_grad():
            _, indices = torch.sort(torch.abs(self.beta), descending=
    False)
            self.beta[indices[:int(self.beta.shape[0] * budget)]] = 0
            mask_out = torch.ones(self.beta.shape[0], dtype=bool)
            mask_out[indices[:int(self.beta.shape[0] * budget)]] = 0
            self.mask_out = mask_out
        return mask_out

    def seperated_clip_beta(self, self_budget, neigh_budget):
        with torch.no_grad():
            beta_self = self.beta[:int(self.beta.shape[0] / 2)]
            beta_neigh = self.beta[int(self.beta.shape[0] / 2):]
            mask_out = torch.ones(self.beta.shape[0], dtype=bool)
            _, indices = torch.sort(torch.abs(beta_self), descending=
    False)
            mask_out[indices[:int(beta_self.shape[0] * self_budget)]] = 0
            _, indices = torch.sort(torch.abs(beta_neigh), descending=
    False)
            indices += beta_self.shape[0]
            mask_out[indices[:int(beta_self.shape[0] * neigh_budget)]] =
    0
            self.mask_out = mask_out
        return mask_out

    def optimize_weight(self, inputs, ref):
        self.weight_optimizer.zero_grad()
        self.beta.requires_grad = False
        self.weight.requires_grad = True
        out = self(inputs)
        loss_weight = nn.MSELoss()(out, ref)
        loss_weight.backward(retain_graph=True)
        self.weight_optimizer.step()
        return loss_weight

    def norm(self):
        with torch.no_grad():
            normp = torch.norm(self.weight, p='fro') / self.
    weight_norm_factor
            self.weight /= normp
            self.beta *= normp
        return
```

```
79
80      def apply_beta(self):
81          with torch.no_grad():
82              self.weight *= self.beta.unsqueeze(1)
83          return
```

Listing 7.2: Base Class for Linear Regression Models.