universität wien

# MASTERARBEIT | MASTER'S THESIS

Titel | Title

Preconditioning in Graph Neural Network Optimization

verfasst von | submitted by

Mag.rer.nat. Simon Fetzel BSc

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

Master of Science (MSc)

Wien | Vienna, 2024

| | |
|---|---|
| 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. Dipl.-Ing. Dr. Wilfried Gansterer M.Sc. |

# Acknowledgements

I'd like to thank Samir Moustafa for his great support with weekly meetings and ideas for this thesis. I also thank Prof. Wilfried Gansterer for the guidance and the good collaboration during this thesis.

# Notation

The following list of notations used in this thesis is non exhaustive.

| | |
|---|---|
| $I_n$ | Identity matrix of size $n \times n$ |
| $I$ | Identity matrix where the size is implied by the context |
| $\delta_{ij}$ | The value of $\delta_{ij}$ is one if $i = j$ and zero otherwise. |
| $A_{ij}$ | the entry in the $i-$th row and the $j-$th column of the matrix $A$ |
| $A^\top$ | The transpose of matrix $A$ |
| $A^{-1}$ | The inverse of matrix $A$ |
| $(x, y)$ | If $x$ and $y$ are numbers, then a two-tuple. If $x$ and $y$ are vectors, then $x$ and $y$ concatenated to a single vector. |
| $\mathcal{N}_i$ | all neighbors of node $i$ |
| $A_{i:}$ | the $i$-th row of matrix $A$ |
| $A_{:i}$ | the $i$-th column of matrix $A$ |
| $\text{diag}(a)$ | A square diagonal matrix which diagonal entries are equal to the vector $a$ |
| $A \times B$ | The cartesian product of the sets $A$ and $B$ |
| $\mathbb{R}$ | The set of real numbers |
| $\mathbb{N}$ | The set of natural numbers |
| $a \cdot b$ | The product of the numbers $a, b \in \mathbb{R}$, the dot may be omitted. |
| $\mathbb{R}^{N \times F}$ | A set of real valued matrices of size $N \times F$ |
| $A \odot B$ | The element-wise product of the matrices $A$ and $B$, which are of the same size |
| $A \otimes B$ | The Kronecker-Product of $A$ and $B$ |
| $AB$ | The matrix multiplication of $A$ and $B$ |
| $\det(A)$ | The determinant of matrix $A$ |
| $\langle x, y \rangle = x^\top y$ | The standard scalar product/inner product of vectors $x$ and $y$ |
| $\|\|a\|\|$ | The 2-norm of $a$ |
| $\|\|a\|\|_p$ | The $p$-norm of $a$ |
| $f : A \to B$ | A function with domain $A$ and codomain $B$ |

| | |
|---|---|
| $f(x)$ | The function value of $f$ at point $x$, where $x$ is also called argument. |
| $f \circ g$ | The composition of $f$ and $g$, such that $(f \circ g)(x) = f(g(x))$ |
| $\nabla_X f(a)$ | The gradient of the function $f$ with respect to $X$ at point $a$ |
| $\lim_{x \to b} g(x)$ | The limit of the function $g$ as $x$ becomes $b$ |
| $\arg\min_{x \leq y} f(x)$ | The argument of function $x$, such that $f(x)$ is minimal and $x \leq y$ |
| $\frac{\partial f}{\partial x}$ | The partial derivative of the function $f$ with respect to variable $x$ |
| $\frac{\partial^2 f}{\partial y \partial x}$ | The second partial derivative of the function $f$ with respect to variables $x$ and $y$ |
| $J_f(a) = \frac{\mathrm{d}f}{\mathrm{d}x}(a)$ | The Jacobian matrix of function $f$ with respect to variable $x$ at point $a$ of the function domain. |
| $\log(x)$ | natural logarithm function applied to $x$ |
| $P(A)$ | The probability of the event $A$ |
| $\mathbb{E}_{P(x)}[X]$ | The expected value of $X$ over the probability distribution $P(x)$ |
| $\mathbb{E}[X]$ | The expected value of $X$ over a distribution, which is inferred from the context. |
| vec(X) | The vectorization of the matrix $X$ (all columns stacked to a column vector as explained in section 3.3.4) |

# Abstract

Preconditioning in the optimization of neural networks has gained attention, as Kronecker-Factored-Approximate-Curvature (K-FAC) showed to improve the training convergence. This thesis investigates the application of K-FAC to Graph Neural Networks (GNNs) for preconditioning and explores the specific considerations for GNNs in this context. K-FAC is compared with Hessian-Free optimization methods, which utilize the conjugate gradient algorithm for preconditioning. The efficiency and hyperparameters of K-FAC are also discussed. The theoretical part includes an introduction to multivariable calculus and explains the calculation of second-order derivatives with vectorization. It was observed that models trained on the Planetoid dataset for node classification with K-FAC showed a tendency to overfit. Additionally, experiments on graph classification tasks using K-FAC on the TUDataset demonstrated improvements on some datasets and models.

# Abstract (German)

Vorkonditionierung in der Optimierung von neuronalen Netzwerken hat Aufmerksamkeit erlangt, da Methoden wie Kronecker-Faktorisierte-Approximierte-Krümmung (K-FAC) die Konvergenz des Trainingprozesses verbessert. Diese Masterarbeit untersucht die Anwendung von K-FAC auf die Optimierung von Graph Neuronalen Netzwerken (GNN) und erklärt die Besonderheiten, die dabei auftreten. K-FAC wird mit anderen Vorkonditionierungsmethoden verglichen, wie der Hessian-Free Methode, die das conjugate gradient Verfahren für die Vorkonditionierung anwendet. Die Effizienz und die Hyperparameter werden ebenfalls diskutiert. Der theoretische Teil der Arbeit beinhaltet eine Einführung in mehrdimensionale Ableitungen und erklärt die Berechnung von zweiten Ableitungen mit Vektorisierung. Es konnte in Experimenten beobachtet werden, dass mit K-FAC trainierte Modelle für Knotenklassifikation eine Tendenz für Überanpassung hatten. Hingegen konnten Verbesserungen in Experimenten mit Graphenklassifizierung mit dem TUDataset beobachtet werden.

# Contents

# 1 Introduction

Neural Networks can be trained to approximate arbitrary functions [KL20]. They are used for various tasks, including classification and object detection[HZRS16][KW17][IFSL20]. The training of a model involves data to optimize its parameters so that the model's output matches the expected output from the training data [GBC16, p.82, p.100]. This is usually achieved by calculating the error between the actual output of the model and the expected output. In the context of machine learning, this error is commonly referred to as the loss. The loss is then used to improve the parameters of the model. The training of a neural network and iterative optimization processes in general can be compared to finding the valley with lowest altitude of a country. Assume a person standing on a mountain and seeking the lowest valley, then the person can go a step into the direction of the steepest descent to find another point which is certainly lower than the previous point. Repeating this procedure, the person would end up in a valley of lower altitude than the starting point. Algorithms that use the steepest descent are the standard approach for optimizing neural networks. In this analogy, each location corresponds to a parameter configuration, and the absolute altitude of the location corresponds to the error.

In mathematical terms, the direction of the steepest descent can be calculated using the gradient of the loss [GBC16, p.85], which is a form of the first derivative. Methods like Stochastic Gradient Descent (SGD) optimize the parameters of models using this approach, therefore such methods are called first-order methods. The negative gradient always points in the direction of the steepest descent of a model's loss function, and thus it can be used to improve the accuracy of the model. Neural networks with more than one hidden layer, also called deep neural networks, pose a challenge to state-of-the-art optimization techniques, due to their high number of parameters or sparse gradients. A sparse gradient means that many of the elements in the gradient are zero. This results in only parts of the model parameters being updated, which can potentially lead to slower training.

On the other hand, increased hardware capabilities allow the enhancement of existing optimization algorithms using the second derivative. Methods that use the second derivative are called second-order methods, which are the focus of this thesis. Incorporating second-order information in the training of a neural network can improve the accuracy and performance of the training, making it desirable to include second-order information in the training process.

## 1.1 Problem Introduction

Assume a model is described by a differentiable function $F(x;\Theta)$, where $x$ is the input for the model and $\Theta$ are the parameters of the model. The training uses a loss function $\varepsilon : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$, which quantifies the error (in other sources called distance or criterion) between the expected output $y$ and the actual output $F(x;\Theta)$ of the neural network [GBC16, p.168].

It is proven that neural networks can approximate a large set of functions [GBC16, p.198][HZRS16], therefore there are many uses for neural networks. However, this result only holds in theory because, in practice, the non-convexity of the training data results in an overall non-convex loss, making the training very difficult. Although most neural networks can achieve a high accuracy in practice, there are few convergence guarantees in general.

For training, a loss $\mathcal{L}(\Theta)$ is defined, combining the error function $\varepsilon$ with the output of the model $F(x;\Theta)$ and the training data $y$, as described in Equation 1.1.

$$\mathcal{L}(\Theta) = \varepsilon(F(x;\Theta), y) \tag{1.1}$$

During training, the gradient of the model loss $\nabla_\Theta \mathcal{L}(\Theta)$ with respect to the parameters $\Theta$ is used to optimize the parameters $\Theta$ of the model, such that the loss is decreased. In particular, the improved parameters $\Theta_{i+1}$ are calculated using the parameters $\Theta_i$ and a step size $\alpha \in \mathbb{R}_{>0}$[GBC16, p.85] as defined in Equation GD-1

$$\Theta_{i+1} = \Theta_i - \alpha \nabla_\Theta \mathcal{L}(\Theta), \tag{GD-1}$$

The expression $-\nabla_\Theta \mathcal{L}(\Theta)$ is the negative gradient of the loss function, which points in the direction of greatest decrease. It can be calculated using calculus rules. The step size $\alpha$ determines how far to move in the direction of the gradient. If the step size is too large, a minimum point might be missed. If it is too small, the training can be very slow. The repeated application of Equation GD-1 is called Gradient Descent Algorithm [GBC16, p.83]. The algorithm is the result of the following more general statement [Kri18] [Mar20, p.10]

$$\frac{-\nabla_\Theta \mathcal{L}}{||\nabla_\Theta \mathcal{L}||} = \lim_{\epsilon \to 0} \frac{1}{\epsilon} \arg\min_{||d|| \leq \epsilon} \mathcal{L}(\Theta + d). \tag{1.2}$$

which describes that the gradient is the direction of largest reduction in $\mathcal{L}$ per unit of change in $\Theta$. While the minimization of the loss function is obviously part of this expression, the constraint $||d|| \leq \varepsilon$ prevents the update from going too far in terms of the Euclidean distance [Gro17]. This is necessary, because the first order gradient is only valid in a local area around $\Theta$.

A widespread variant of Gradient Descent is Stochastic Gradient Descent (SGD), which samples a minibatch from the training data and performs the Gradient Descent steps on the minibatch [GBC16, p.294]. An important addition to SGD is momentum, which calculates a moving average of the gradients to improve the training process [GBC16,

p.311]. The term "momentum" is derived from physics: if a particle has a momentum and a low force is applied, particle's direction will change gradually due to the force. The particle will not respond instantaneously. Similarly, in an optimization algorithm with momentum, changes in the gradient are smoothed by the moving average, allowing the optimizer to escape local minima more effectively. The ADAM (Adaptive Moment Estimation) algorithm enhances SGD with momentum by using estimates of second-order information to adapt the step size $\alpha$ [KB15].

A local minimum of $\mathcal{L}(\Theta)$ is a critical point, which is indicated when the gradient of the loss function is zero, i.e $\nabla\mathcal{L}(\Theta) = 0$. This zero of the loss function can also be found using Newton's method, which uses the second derivative, the Hessian matrix $H$ [Mar20, p.12]:

$$\Theta_{i+1} = \Theta_i - \alpha P \nabla_\Theta \mathcal{L}(\Theta), \tag{GD-2}$$

where $P = (H_\mathcal{L}(\Theta))^{-1}$ denotes the inverse of the Hessian matrix. If the starting point for Newton's method is chosen carefully, the convergence is guaranteed to be quadratic [GLZ+21]. Therefore, it is desirable to use second-order methods in the training of neural networks. Another advantage of Newton's method is its invariance to linear transformations of the parameters $\Theta$ [GLZ+21]. A major disadvantage of using the Hessian is that it is not always positive-definite. If the matrix is not positive-definite, Newton's method cannot be applied. Therefore, other matrices may be chosen for $P$, for example the Fisher matrix, which will be discussed later.

The Equations (GD-1) and (GD-2) differ only by the matrix $P$. Incorporating second-order information is therefore also called preconditioning, as a second-order method alters the gradient using a matrix multiplication and then calculates the improved parameters using the same equation as first-order methods. Furthermore, the calculation and storage of the matrix $H_\mathcal{L}(\Theta)$ is often not feasible, so it is approximated in second-order methods. There are many different approximation techniques [Mar10][MG15][FKA21][YGS+21], which will be explained in Section 2 and two of them will be detailed in Section 5. These techniques mostly focus on general neural networks. There are not many studies known to date that focus on the optimization of Graph Neural Networks.

### 1.1.1 Graph Neural Networks (GNNs)

The main focus of this thesis is the improvement of the training of GNNs, regarding the enhancement of accuracy and convergence speed.

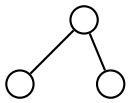
The uses of GNNs include node classification and graph classification tasks. In node classification, each node is assigned to a class using the features of the node and the graph structure. The GNN refines the features of the nodes in each layer to make an accurate estimation of the class. In a graph classification task, the features of all nodes in the final layer are aggregated to estimate the class for the whole graph, e.g. by averaging all node features. In detail, this can be explained using the following definitions.

Assume an arbitrary graph $G = (V, E)$, $V = \{1, \dots, N\}, E \subset V \times V$, where each node $i$ has a feature vector $X_i \in \mathbb{R}^{1\times F}, F \in \mathbb{N}_{>0}$. The feature matrix $X^{(0)} = X \in \mathbb{R}^{N\times F}$ consists

of all feature row vectors $X_i$ stacked. A molecule could be an example for a graph, where each node is an atom and the feature vector of each node could contain the mass of the atom and the type of the atom.

Graphs are different from usual input sources for neural networks such as images, text or arbitrary vectors. Conventional neural networks cannot be used to process graphs, because the description of a graph is not unique and the neural network needs to be compatible with graphs of different sizes. Typically, the input for neural networks is always of the same size and can be described uniquely. As an example, the description of a graph by an adjacency matrix $A \in \mathbb{R}^{N \times N}$ is not unique, where $A_{ij} = 1$ if nodes $i$ and $j$ are connected with an edge and $A_{ij} = 0$ otherwise. The matrix $D \in \mathbb{R}^{N \times N}$ with $D_{ii} = \sum_j A_{ij}$ is referred to as degree matrix of $A$. The matrix $\tilde{D}$ refers to the degree matrix of $A + I_N$. A permutation of the vertices gives a new adjacency matrix. For example, the following adjacency matrices describe the same graph:

$$A_1 = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \qquad A_2 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$



Therefore a new type of neural network is necessary. A Graph Convolutional Network (GCN) [KW17] is a permutation invariant neural network. It consists of layers, which output is calculated by multiplying the normalized adjacency matrix $\tilde{A} = \tilde{D}^{-1/2}(A + I)\tilde{D}^{-1/2}$ from the right with the input matrix $X^{(k)}$ and multiplying this product from the right with a weight matrix $W \in \mathbb{R}^{F \times F'}$ as defined in Equation 1.3:

$$X^{(k+1)} = \tilde{A} X^{(k)} W. \tag{1.3}$$

The normalized adjacency matrix is calculated according to [KW17]. The weight matrix describes a linear transformation of the features and is a learnable parameter, that will be optimized during the training. Typically, the number of output features $F'$ is lower than the number of input features $F$. During the training, the weight matrix will be adapted so that the output is useful for determining the correct class of a node. The structure of Equation 1.3 is similar to conventional linear layers used in neural networks, where the output $Y = XW$ is calculated by multiplying the input matrix $X$ from the right with a weight matrix $W$. A GCN layer adds the multiplication from the left with the normalized adjacency matrix.

A layer of a GCN will implement the message-passing scheme, where each node exchanges information with its neighbors and aggregates this information into a new feature vector [KW17]. It is realized by multiplying the feature vectors from left with the normalized adjacency matrix. In more detail, if $\mathcal{N}(i)$ denotes all neighbors of a node $i$, then the new feature vector $X_i'$ can be calculated by summing the transformed feature vectors $Y_j = X_j W$ of all neighbors of $v$ including itself, multiplied with the edge weight specified by the normalized adjacency matrix $\tilde{A}$ as described in Equation 1.4:

$$X_i' = \sum_{j \in \mathcal{N}(i) \cup \{i\}} \tilde{A}_{ij} Y_j \tag{1.4}$$

The idea of message passing is common to all variants of GNNs, such as Graph Isomorphism Networks (GIN) and Graph Attention Networks (GAT). Message passing also has an influence on the optimization process, because the adjacency matrix is sparse. An experiment in Chapter 4 on node classification illustrates, that the sparsity of the adjacency matrix results in sparse gradients, which can be a problem for the training of a neural network, because not all entries of the parameters are updated. Incorporating second-order information could improve the sparsity of gradients and enhance the convergence of the training.

### 1.1.2 Natural Gradient Descent

Natural Gradient Descent (NGD) is a second-order method. Its major advantage compared to Newton's method is its richer invariance property: while Newton's method is invariant to linear parameterizations, Natural Gradient methods aims to be invariant to all differentiable and invertible transformations of the parameters [GLZ+21]. This is achieved by a reformulation of the problem with respect to prediction functions [Mar20]. In Equation 1.2, the Euclidean distance is used, to prevent the update step from moving too far away from the current parameters $\Theta$.

This is problematic, as the actual problem specific metric may not be Euclidean. Therefore NGD replaces the Euclidean distance in Equation 1.2 with a different metric. The distance between two parameters $\Theta$ and $\Theta + d$ is measured with respect to the Kullback-Leibler-Divergence (denoted as KL) of the resulting distributions $P_{x,y}(\Theta)$ and $P_{x,y}(\Theta + d)$ of the model:

$$\mathrm{KL}(P_{x,y}(\Theta + d), P_{x,y}(\Theta)) \tag{1.5}$$

The Fisher matrix $F$ is a quadratic approximation of the KL-divergence, which can be used instead of the Euclidean distance in Equation 1.2. The matrix $F$ is defined as follows [Mar20]:

$$F = \mathbb{E}_{P_{x,y}}[\nabla \log(p_{x,y}(\Theta))(\nabla \log(p_{x,y}(\Theta)))^T], \tag{F}$$

where $p_{x,y}(\Theta)$ is the density function of $P_{x,y}(\Theta)$. The inverse of $F$ is used as the preconditioning matrix in natural gradient methods, so $P = F^{-1}$. The preconditioned gradient is the steepest descent direction in the distribution space [Mar20]. As GNNs usually have a lot of parameters (especially GAT), preconditioning might help to efficiently train these networks, as Natural Gradient Descent makes the training invariant to a large class of reparameterizations.

## 1.2 Synopsis

The next chapter describes Related Work, including approaches using diagonal approximations of the Hessian, procedures utilizing exact Hessian-vector products in combination

with the conjugate gradient algorithm for preconditioning, and Kronecker-Factored approximations. A paper applying Kronecker-Factored Approximate Curvature (K-FAC) on Graph Convolutional Networks is described as a promising approach.

The Background Chapter introduces derivatives and backpropagation in detail. Both a compact and a detailed derivation of the backpropagation equations for linear layers is explained. The chapter concludes with the necessary theory to calculate second-order derivatives using vectorization.

The Problem Statement Chapter outlines the research questions, including the issue of sparse gradients in Graph Neural Network optimization. Trivial approaches to addressing these questions are discussed, along with reasons for their exclusion.

The Methodology Chapter explains K-FAC and its application to GNNs in detail. Hessian-free preconditioning is the second approach, which will be discussed.

Both methods are applied to GNNs, and the results are discussed in the Experiments Section. While node classification experiments on the Planetoid dataset did not lead to improved test accuracies, graph classification experiments showed improvements in test accuracies on some datasets of the TUDataset collection. Additionally, experiments on the K-FAC hyperparameters are presented.

The Future Work Chapter briefly describes another preconditioning idea called QR-Inverse. It also mentions an idea stemming from Graph editing, which did not perform well in a quick experiment.

Lastly, the Conclusion Chapter summarizes the basic results, highlighting that preconditioning might improve accuracies for some datasets and models, but can also lead to overfitting.

# 2 Related Work

Optimization of neural networks gains constant research attention, as models get larger, more complex and new methods are incorporated into models. First-order optimization methods have been very successful and were improved using heuristics. For example, adding momentum to a first order optimization method makes it more robust against the noisy and non-convex loss landscape. In the following, second order methods are investigated. As a study of all the available literature on preconditioning is not feasible, only recent articles are studied, which usually base on established methods.

In general, two classes of second order methods can be distinguished. First, methods which approximate the second derivative using the first derivative. An example of such a method is the Broyden-Fletcher-Goldfarb-Shannon (BFGS) technique. It is an established second oder optimization technique and mathematically well discussed. These methods have several advantages. For example, the memory requirement is low compared to storing a dense Hessian matrix and memory usage can be limited. The majority of the discussed papers take this approach.

Secondly, there are methods which do not use the gradient to approximate the second derivative, but aim for exact calculations. Due to the large memory requirements, these exact calculations need to take assumptions or simplifications in order to compete with first-order methods.

## 2.1 Diagonal approximation of the Hessian

A good example of a method with such simplifications is AdaHessian [YGS+21], an optimization method which uses ideas from the ADAM optimizer and incorporates the second derivative. The paper shows the Hessian of a convolutional neural network as an image to demonstrate the sparsity of the Hessian (see Figure 2.1). All entries in the diagonal of the example are nonzero while the magnitude of non-diagonal elements are mostly much lower compared to the diagonal. Therefore it makes sense to approximate the Hessian using a diagonal matrix. Furthermore, the loss landscapes of the convolutional blocks of the first layer are plotted. The visualizations illustrate, that the loss landscapes have very different curvature. Applying second order information in the optimization process causes the loss landscape to be adequately rescaled. This can also be interpreted as rescaling the step size for each parameter. For example, if a model contains two parameters, one measured in metres and one measured in centimetres, they are likely to need different step sizes. If the full Hessian matrix (or a block diagonal approximation) is used, the landscape may rotate as well.

The Hessian will become noisy if the training is done in batches. Averaging the

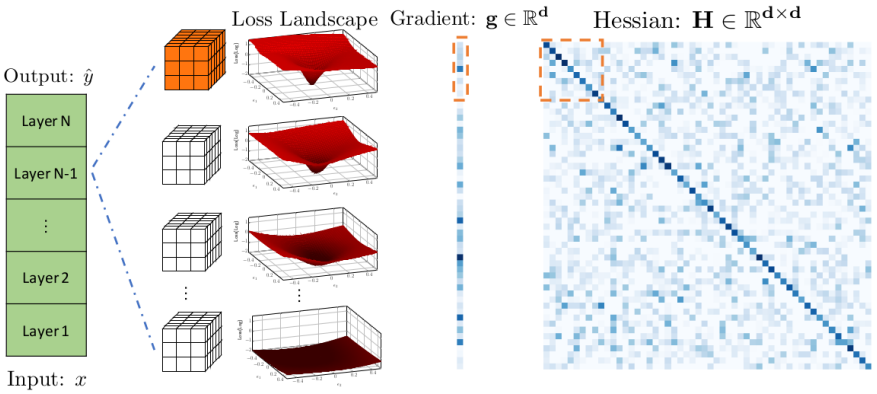



Figure 2.1: Figure 1 from [YGS+21], where a simple model with $N$ convolutional layers is presented. The first convolutional block is colored in orange and the corresponding entries in the gradient and Hessian are marked with orange.

Hessian to smooth the noise is not possible, because the memory and computation requirements would be too large. Another problem is the application of the Hessian to the gradient. The inverse of the Hessian needs to be calculated, which is very expensive. Both problems vanish if the Hessian is approximated using a diagonal matrix. This is done using Hutchinsons's method, which approximates the diagonal of a matrix, as well as an additional backpropagation step. This simplification also allows to incorporate spatial and temporal averaging.

The experimental results show that AdaHessian improves the test accuracy on the cost of an increased training time. Although the Hessian was only approximated using a diagonal matrix, the theoretical complexity of this method is twice the complexity of SGD. The training time can be improved by reducing the update frequency of the diagonal approximation of the Hessian (e.g. every 4th iteration). The presented experiments indicate, that this reduction is reasonable. In [DCJ23], this idea is called *lazy Hessian updates* and is investigated more thoroughly. The error introduced because of reusing an old Hessian can be counterbalanced by an increased regularization, i.e. making smaller steps in the optimization process. According to the derivations of [DCJ23], updating the Hessian every $d$ steps is optimal, where $d$ is the dimension of the optimization problem.

## 2.2 Block-diagonal approximations

Overall, AdaHessian improved the accuracy for the examples mentioned in [YGS+21] using second order information. But AdaHessian will not improve a sparse gradient compared to ADAM, because the approximation of the Hessian only includes the diagonal. Multiplying a diagonal matrix with a sparse vector will leave the vector sparse. Therefore, a better approximation of the Hessian matrix is desirable. In [DHH20], an approach to calculate a block-diagonal approximation of the Hessian matrix is presented. Basically, the blocks of the Hessian can be calculated by the second derivative of the individual

layers of the neural network. Directly calculating the second derivative of the layers by auto differentiation twice would result in a dense matrix, which inversion is costly. Instead, backpropagation of gradients can be used to calculate the block diagonal approximation of the Hessian matrix (similar as in AdaHessian). Unfortunately, this computation can result in non positive definite blocks, which cannot be used for the training. There are different approaches to alter the blocks, such that they become positive semidefinite. Changing the negative eigenvalues of the eigenvalue decomposition to zero or taking their magnitude can solve this problem. Furthermore, batch training causes problems, because it calculates an average out of multiple Hessians. This can break the Hessian block structure. Instead of computing an average over Hessians, an average Hessian is passed between the layers. In [DHH20], the conjugate gradient method is used to precondition the gradient using the batch averaged Hessian, as a linear system needs to be solved. This can be done efficiently in parallel. The conjugate gradient algorithm works only with positive definite matrices, which is ensured by the eigenvalue modification approach mentioned earlier.

Another method aiming for a better approximation of the Hessian matrix is presented in [XN09]. It does not investigate neural networks, but a general unconstrained minimisation problem. On contrary to previously discussed approaches, this paper assumes, that the Hessian matrix is available. Although this assumption is unrealistic for neural networks, the approach is still interesting because it is relevant to approximation techniques. An incomplete Hessian matrix is constructed out of the Hessian matrix at hand by filtering out entries of the Hessian with sparse patterns or sparse matrix techniques. This method is proven to be convergent if the incomplete Hessian matrix is symmetric and positive definite. Unfortunately, it cannot be assumed in general, that the Hessian matrix is positive definite. Furthermore, this method was only tested on a rather small optimization problems (e.g. Hessian matrix of size 300x300). However, the method did work well in the experiments from the paper, as the computation time was decreased and the accuracy was comparable.

## 2.3 Exact Hessian vector products

However, the calculation and storage of the Hessian matrix can be avoided as explained in [HEAV19]. It uses the reverse-mode automatic differentiation (RMAD), also known as back-propagation, and forward-mode automatic differentiation (FMAD) to compute the product $\hat{H}v$, where $\hat{H}$ is the Gauss-Newton matrix

$$\hat{H} = J_\phi H_l J_\phi^T,$$

$H_l$ is the Hessian of the loss function and $J_\phi$ is the gradient of the model. Methods that do not calculate the Hessian matrix directly are also referred to as Hessian-free methods. The Gauss-Newton matrix has the advantage, that it is positive-definite by construction, when the loss function is convex. The linear system arising from Equation (GD-1) is then solved using the conjugate gradient (CG) method, which only requires the product $\hat{H}J$, where $J$ is the gradient. The CG method can require many iterations

to convergence; as an alternative, gradient descent is used instead. Furthermore, a regularization term $\lambda I$ is added to the matrix $\hat{H}$ to avoid problems with a singular $\hat{H}$ matrix. The experiments show, that this method works well and achieves competitive results compared to established approaches.

## 2.4 Advanced Approximation Techniques

The methods discussed so far either use a block diagonal, diagonal or an implicit approach to represent the Hessian matrix. The following method is based on a well-established method to approximate the Hessian matrix. The article [GL03] presents an improved approximation to the Hessian matrix, based on the BFGS approach. The Hessian is written as a sum of a diagonal matrix and rank-one matrices. This decomposition is also possible for the inverse of the Hessian. The BFGS method can also be implemented in a memory-limited fashion, where parts of the rank-one matrices are dropped and replaced with newer estimates. Another mentionable property concerns the positive definiteness. The approximation will yield a positive definite matrix if the initial matrix is positive definite. Reduced-Hessian methods as used in [GL03] reformulate the BFGS approach to use an orthonormal basis, to represent the curvature information. The storage requirements as well as function evaluations could be reduced compared to other limited memory methods.

An interesting approach combining ideas from BFGS and the Fisher matrix is explained in [FKA21]. The presented methods approximate the Hessian using a sum of rank-1 matrices, similar to the usual BFGS approach. The paper gives a static algorithm for network compression and a dynamic algorithm for optimization. As a basis, the Fisher matrix is used to approximate the Hessian. The algorithms focus on calculating the inverse Hessian vector products (IHVP) without forming an explicit approximation of the Hessian. As this work focuses on optimization, only the dynamic algorithm is described here.

The basic idea uses the empirical Fisher matrix, where the expectation in Equation (F) is replaced with an average:

$$\hat{F} = \frac{1}{N} \sum_{i=1}^{N} \nabla l_i (\nabla l_i)^T,$$

where $\nabla l_i$ is the gradient of the loss in the $i$-th training iteration. Because of the outer products, the Fisher matrix can be approximated as a sum of rank-1 matrices. The Sherman-Morrison formula allows to recursively calculate the inverse of the Fisher matrix, adding one gradient $\nabla l_i$ at a time:

$$\left(A + uv^{\mathsf{T}}\right)^{-1} = A^{-1} - \frac{A^{-1} u v^{\mathsf{T}} A^{-1}}{1 + v^{\mathsf{T}} A^{-1} u}$$

$$\hat{F}^{-1} = \hat{F}_N^{-1} = \hat{F}_{N-1}^{-1} - \frac{\hat{F}_{N-1}^{-1} \nabla l_n \, (\nabla l_n)^T \, \hat{F}_{N-1}^{-1}}{N + (\nabla l_n)^T \hat{F}_{N-1}^{-1} \nabla l_n} \qquad F_0^{-1} = \lambda^{-1} I$$

Using the definition $v_i = \hat{F}_{N-1}^{-1} \nabla l_i$, the recursion can be written as:

$$\hat{F}_i^{-1} x = \hat{F}_{i-1}^{-1} x - v_i \frac{v_i^T x}{m + (\nabla l_i)^T v_i} = \lambda^{-1} x - \sum_{j=1}^{i} v_j \frac{v_j^T x}{m + (\nabla l_i)^T v_j}$$

During optimization, a sliding window of gradients with a fixed size is used. During training, the oldest gradient is removed from the sliding window and the latest gradient is added to the window. The experiments conducted do not use momentum. The results indicate a higher accuracy compared to first order methods. The method is also compared to AdaHessian. While the accuracy of both methods is comparable, the overhead of M-FAC is much smaller.

## 2.5 Kronecker-factored Hessian methods

In general, BFGS methods are robust and well accepted. In the setting of neural networks, the architecture of multiple layers allows to create more sophisticated approximations of the Hessian matrix than used in BFGS, as explained previously, using block-diagonals. The approximation of these blocks using Kronecker products is used in many approaches, including [GRB20]. While this technique was first used for Natural Gradient Descent, [GRB20] employs Kronecker factorization for the block-diagonal Hessian with the BFGS methods in mind. Therefore the method is called K-BFGS. Assuming a dense neural network with $L$ layers and an activation function $\phi_l$ for each layer $l \in \{1, \dots, L\}$, weight matrices $W_l$ which include the bias term, the output of the neural network is calculated using the following equations:

$$a_0 = x \qquad \overline{a}_{l-1} = (a_{l-1}, 1) \qquad h_l = W_l \overline{a}_{l-1} \qquad a_l = \phi_l(h_l)$$

The main idea of Kronecker factorization can be illustrated with the following equations [GRB20], where $g_l = \frac{\partial F}{\partial h_l}$ and only a single data point is considered:

$$\nabla f_l = g_l (a_{l-1})^T \qquad \Leftrightarrow \qquad \operatorname{vec}(\nabla f_l) = a_{l-1} \otimes g_l$$

The second derivative $G_l = \frac{\partial^2 f}{\partial h_l^2}$ can be written using:

$$\nabla^2 f_l = (a_{l-1}(a_{l-1})^T) \otimes G_l$$

If multiple points are considered, the Hessian $\nabla^2 f_l$ of a single layer $l$ is replaced with the expectation of the Hessian among the different data points:

$$\mathbb{E}[\nabla^2 f_l] \approx \mathbb{E}[a_{l-1}(a_{l-1})^T] \otimes \mathbb{E}[G_l] := A_l \otimes G_l \tag{IAD}$$

This approximation is also used in Kronecker-Factored-Approximate-Curvature (K-FAC) [MG15]. It is assumed, that activations and pre-activiation derivatives are independent [KAG+23]. The implementation in [GRB20] will further approximate the matrices $A_l$ and

$G_l$ with positive definite matrices $H_a^l$ and $H_g^l$. These matrices are used to precondition the gradient (in a slightly different variant than in (GD-2)). In the training process, the matrices $H_a^l$ and $H_g^l$ will be updated in each iteration. The approach from [GRB20] contains several more changes to the BFGS method, which are omitted here. The presented experiments show, that the method performs simliarly to K-FAC, a method which implements Natural Gradient Descent in a similar fashion as in K-BFGS.

## 2.6 Natural Gradient Descent

Before exploring various K-FAC methods, it is worth looking at how close an approximation to the Fisher matrix needs to be, to get a convergent training process. According to [KO20], the fast convergence of approximative NGD methods hold by the cost of more iterations, assuming an infinite-width deep neural network. The common property in the training process among different approximations of the Fisher matrix is the isotropic gradient, which means that the convergence of all parameters occurs at the same speed.

As a side note: while all methods explained in the following use the Fisher information matrix to precondition the gradient, the paper [DLZ+22] illustrates the connection of NGD to a more general structure, called a Riemann manifold. NGD defines a Riemann metric using the Fisher matrix. The paper illustrates well, that choosing an appropriate metric can accelerate the optimization process significantly.

The paper [KAG+23] tries to get rid of the assumption formulated in Equation (IAD). The first approach is to create a better approximation by solving

$$\min_{(R,S)} ||F_{i,i} - R \otimes S||_F$$

where $F_{i,i}$ is the Fisher matrix block of the i-th layer and $R, S$ are matrices. The minimization is solved using Kronecker product singular value decomposition technique and reformulating the problem using vectorization.
This approximation is only of rank-1, therefore another approach is presented, which extends $R \otimes S$ by adding another Kronecker product of two matrices $P \otimes Q$. The resulting problem can be solved using two solution methods. The first technique, deflation, solves the problem in two steps. First the matrices $R, S$ are calculated while ignoring P and Q, and then using the same approach $P, Q$ are calculated, while $R, S$ are fixed. The second technique is called Lanczos bi-diagonalization and uses the two largest singular values to determine the four matrices $R, S, P, Q$. The procedure is executed multiple times with different initial vectors $q^{(0)}$ for the Krylov subspace.

The last method presented uses the same matrices for $R$ and $S$ as in the original formulation of KFAC but adds additionally the matrices $P \otimes Q$ using the same procedure as in the deflation technique. As the preconditioning using the Fisher matrix needs the inverse of this matrix, the question how to efficiently calculate the inverse of the Fisher matrix blocks has to be clarified. Usually this is done using the identities

$$(A \otimes B)^{-1} = A^{-1} \otimes B^{-1} \qquad \text{as well as} \qquad (A \otimes B)^{-1}\text{vec}(X) = \text{vec}(B^{-1} X A^{-T}).$$

Unfortunately, there are no similar identities available for the rank-2 approximations and the resulting linear system

$$(A \otimes B + C \otimes D)u = v.$$

In [KAG+23], an algorithm of complexity $O(d^3)$ is described to calculate the inverse using the eigenvalue- and SVD-decompositions, where $d$ is the size of the matrices $A, B, C, D$.

The experiments with the datasets CURVES, MNIST and FACES show, that all presented approximations are more accurate than the original KFAC approach which used the assumption (IAD). The difference is especially large in the beginning of the training. The methods could achieve comparable or better accuracy than KFAC.

There are more papers explaining how to improve the complexity, storage requirements or accuracy of KFAC. Two papers basically show a similar idea: [KL22] and [GLB+18]. Using projection and basis changes, the KFAC algorithm is modified. In [KL22], a projection constructed using the eigenvectors with the largest eigenvalues is used to reduce the dimensionality of the gradients. Numerical experiments with LeNet-5 and the Fashion MNIST dataset indicate, that the accuracy does not degrade under this dimensionality reduction. Interestingly, event experiments using random projections did not show a reduction in accuracy.

The approach from [GLB+18] uses the eigenvalue decomposition of the matrices $A_l$ and $G_l$ in Equation (IAD) to interpret the preconditioning in another way:

$$A_l = U_A S_A U_A^T \qquad G_l = U_B S_B U_B^T,$$

which leads to

$$A_l \otimes G_l = (U_A S_A U_A^T) \otimes (U_B S_B U_B^T) = (U_A \otimes U_B)(S_A \otimes S_B)(U_A \otimes U_B)^T.$$

In this sense the preconditioning of the gradient using the Kronecker-Factored-Approximated Fisher matrix projects the gradient onto another basis, rescales the vector and then projects it back. KFAC methods only use approximative scaling, which the presented method Eigenvalue-corrected KFAC (EKFAC) improves. This is implemented by replacing $S_A \otimes S_B$ with a matrix $S^*$, where $S^*_{ii} = \mathbb{E}[((U_A \otimes U_B)^T \nabla_\Theta)_i^2]$, which can be derived by the following minimization:

$$\text{argmin}_S \; ||G - (U_A \otimes U_B) S (U_A \otimes U_B)^T||_F.$$

Therefore the approximation of EKFAC to the Fisher matrix will be better than the approximation of KFAC with respect to the Frobenius norm. Experiments using an autoencoder and the MNIST dataset as well as VGG11 on CIFAR-10 indicate, that EKFAC performs comparably or better than KFAC, while the computation time is lower.

## 2.7 GNN Optimization with NGD

There aren't a lot of publications available conerning GNNs and NGD. The only publication currently known to the author is [IFSL20]. It introduces a semi-supervised training for

GNNs with KFAC, consisting of $\overline{n}$ labeled samples and $n - \overline{n}$ unlabeled samples. The unlabeled samples can also be used to approximate the Fisher matrix, with some adaptions. Additionally, the paper proposes to incorporate the unlabeled data in the training by setting the labels $y_{\overline{n}+1}, \ldots, y_n$ by sampling from $p(y|x; \Theta)$. The loss is then a sum of the loss of the labeled data and of the loss of the unlabeled data, including a hyperparameter $\lambda$:

$$\frac{1}{\overline{n}} \sum_{i=1}^{\overline{n}} l(y_i, F(X, A; \Theta)) + \frac{\lambda}{n - \overline{n}} \sum_{i=\overline{n}+1}^{n} l(y_i, F(X, A; \Theta))$$

As the labeles tend to improve over the training, the following definition of the hyperparameter $\lambda$ is proposed:

$$\lambda(t) := \left( \frac{t}{\max(t)} \right)^{\gamma}$$

where $t$ is the iteration, $\max(t)$ is the maximum count of iterations and $\gamma$ is a new hyperparameter. Therefore, the unlabeled data will not contribute much to the loss in the beginning of the training, but will contribute equally as the labeled data at the end of the training.
The proposed method was tested on the CiteSeer, Cora and PubMed datasets using a two-layer GCN. The experiments show, that including unlabeled data improved the accuracy. Stochastic Gradient Descent did not perform well, but using preconditioning it could achieve a similar accuracy as ADAM.

## 2.8 Graph editing

This survey is concluded with the discussion of the paper [BSAGBG24], which investigates the bias of GNNs with respect to the graph structure. For example, the mass of a molecule is independent of the actual structure of the molecule. A similar artifical experiment called "Node Sum", where the graph structure is irrelevant for the output labels, was conducted in [BSAGBG24]. The experiment was also done without the graph structure, i.e. an empty list of edges. For the Node sum experiment as well as two other experiments, the GNN trained without the graph structure performed better than the GNN trained with the graph structure. The authors conclude that the GNNs tend to overfit the graph structure.

However, regular graphs performed better (regular means all nodes have the same degree). Therefore a scheme to edit the training graphs is proposed, to make these graphs regular (or approximate a regular graph). Experiments on synthetic tasks showed, that this scheme improved the accuracy.

### 2.8.1 Summary

Different preconditioning approaches have been discussed including diagonal, block-diagonal and dense approximations. To date, the only paper applying preconditioning to

GNNs used K-FAC and reported improvements in test accuracy. This paper focused solely on Graph Convolutional Networks (GCNs) and did not address other GNN architectures.

Hessian-free methods, which compute Hessian-vector products and use other algorithms to calculate the inverse Hessian-vector product have not yet been applied to GNNs. The Hessian-free method discussed in [HEAV19] showed slightly faster convergence compared to training with the ADAM optimizer. Kronecker-Factored methods offer a greater improvement in convergence speed and are generally preferable.

Both Hessian-free methods and M-FAC offer versatility because they do not require adaptation to the neural network architecture. In contrast, K-FAC and K-BFGS [GRB20] are specifically designed for preconditioning linear layers and may require adaptation for use with other types of layers.

# 3 Background

This chapter will describe the problem of optimizing a graph neural network in detail and introduce the necessary theory to explain the problem statement.

First, an overview of Graph Neural Networks (GNNs) will be provided, highlighting how they differ from other neural networks. Next, the concept of derivatives, which are crucial for the optimization process, is explained. Finally, vectorization and second derivatives are discussed.

## 3.1 Optimizing a Graph Neural Network

Graph Neural networks, such as Graph Convolutional Networks (GCN), Graph Attention Networks (GAT) and Graph Isomorphism Networks (GIN) incorporate message passing along with linear layers. A GCN layer, which uses a weight matrix $W_k$, normalized adjacency matrix $\tilde{A}$ (see Section 1.1.1) and input $X^{(k)}$ can be implemented with two matrix multiplications. The input $X^{(k)}$ is multiplied from the left with the normalized adjacency matrix $\tilde{A}$ and from the right with the weight matrix $W_k$ as described in Equation (3.1):

$$X^{(k+1)} = \tilde{A}X^{(k)}W_k \tag{3.1}$$

The two matrix multiplications can be decomposed into two layers, each defined using one matrix multiplication. The first layer is a linear layer, describing the multiplication of the input with the weight matrix $W_k$ from the right. The second layer, which will be called the message passing layer, multiplies the input from the left with the normalized adjacency matrix $\tilde{A}$:

$$Y^{(k+1)} = X^{(k)}W_k \tag{3.2}$$

$$X^{(k+1)} = \tilde{A}Y^{(k+1)} \tag{3.3}$$

The experiments in this thesis will use this split to implement GCN layers.

This decomposition allows to focus on ordinary neural networks and then investigate the effect of message passing layers.
Furthermore, the training of Graph Neural Networks can be implemented with the usual Gradient Descent algorithm and its variants, as all architectures are differentiable. The Gradient descent algorithm is described in Equation (GD-1), where the gradient of the loss $\nabla L$ of the neural network is used. The calculation of this gradient includes the computation of the derivatives of each layer of the neural network.

## 3.2 First Order Derivatives

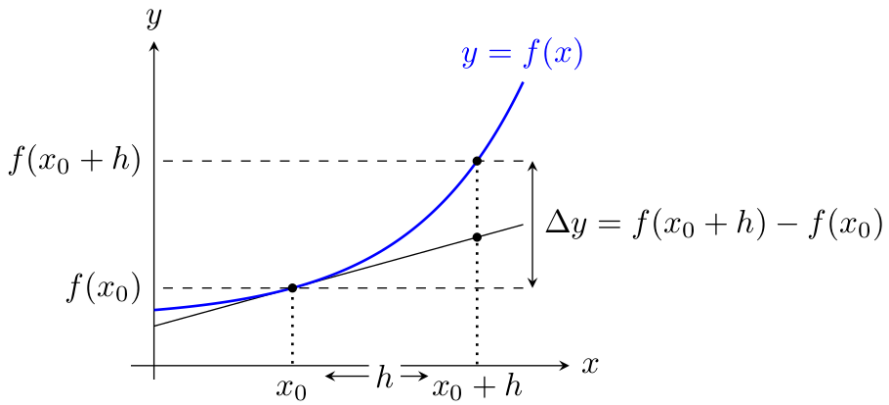



Figure 3.1: Illustration of the one dimensional derivative. The slope at $x_0$ can be approximated by $\Delta y / \Delta x$. The slope of the line touching the function at $x_0$ is equal to the derivative $f'(x_0)$.

To better understand the impact of message passing layers on gradients and the preconditioning of neural networks, the following sections describe multivariable calculus and the analytical formulation of neural network derivatives.

First, the term derivative must be clarified. The slope $k$ of a one dimensional function $f : \mathbb{R} \to \mathbb{R}$ at $x_0 \in \mathbb{R}$ with $h > 0$ can be approximated with the difference $\Delta y$ of the function values at $x_0 + h$ and $x_0$ divided by $\Delta x$ the difference of $x_0 + h$ and $x_0$ [Shu, p.143]. Figure 3.1 illustrates this approach in a plot and Equation 3.4 summarizes it analytically:

$$k \approx \frac{\Delta y}{\Delta x} = \frac{f(x_0 + h) - f(x_0)}{x_0 + h - x_0} = \frac{f(x_0 + h) - f(x_0)}{h} \tag{3.4}$$

Taking the limit of Equation 3.4 with respect to $h \to 0$ yields the definition of the slope of $f$ at $x_0$, called the derivative of $f$ at $x_0$, denoted with $f'(x_0)$ [MN19, p.90]:

$$k = f'(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h} \tag{3.5}$$

Assuming the limit exists, Equation 3.5 can be reformulated by subtracting $f'(x_0)$ on both sides and simplifying the right hand side:

$$0 = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h} - f'(x_0) \tag{3.6}$$

$$0 = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0) - f'(x_0) \cdot h}{h} \tag{3.7}$$

The resulting numerator of the right hand side fraction in Equation 3.7 will be called $r(h)$ :

$$r(h) := f(x_0 + h) - f(x_0) - f'(x_0) \cdot h \qquad \text{such that} \qquad \lim_{h \to 0} \frac{r(h)}{h} = 0 \tag{3.8}$$

Rearranging Equation 3.8 shows, that the function can be approximated using the derivative near $x_0$:

$$f(x_0 + h) = f(x_0) + f'(x_0) \cdot h + r(h) \tag{3.9}$$
$$\approx f(x_0) + f'(x_0) \cdot h \tag{3.10}$$

The form of Equation 3.9 is the same as the first order Taylor expansion. The idea of approximating a function at a point $x_0$ using a linear function is taken to define multivariable derivatives. First, consider a function $f(x_1, \ldots, x_F)$ with multiple input variables, but only one output. Assume a derivative of $f : \mathbb{R}^n \to \mathbb{R}$ at a point $a = (a_1, \ldots, a_F) \in \mathbb{R}^F$ should be calculated.

A linear approximation in the form of Equation 3.9 would require a vector $b \in \mathbb{R}^{1 \times F}$, which describes a linear function $g(h) = bh = b_1 \cdot h_1 + \cdots + b_F \cdot h_F$ where $h \in \mathbb{R}^{F \times 1}$ with output $g(h) \in \mathbb{R}$:

$$\lim_{x \to a} \frac{||f(x_1, \ldots, x_F) - f(a_1, \ldots, a_F) - b_1(x_1 - a_1) - b_2(x_2 - a_2) - \cdots - b_F(x_F - a_F)||}{||x - a||} = 0,$$

If all input variables but $x_i$ of the function $f$ are considered constant, the resulting function $f_{x_i}$ is again one-dimensional. Finding a linear approximation to $f_{x_i}$ at $a$ is the same problem as finding a linear approximation for a one-dimensional function:

$$\lim_{x \to a} \frac{||f_{x_i}(x_1, \ldots, x_F) - f(a_1, \ldots, a_F) - b_i(x_i - a_i)||}{||x_i - a_i||} = 0, \tag{3.11}$$

Hence the entries of the vector $b_i$ can be determined using one dimensional derivatives, where all inputs of the function are considered constant but one. A derivative of a function, where all variables but one are considered constant, is called partial derivative. It can be calculated using ordinary one-dimensional calculus rules.

This approach might seem difficult to understand at first, but it becomes clearer with an illustration. Consider the function $f(x, y) = x^2 + y^2$. This function can be plotted by drawing the points $(x, y, f(x, y))$, in other words, the function determines the z-coordinate (height) of the points. A plot of this function can be seen in figure 3.2. In three dimensions, the plot would look like a upward-facing bowl, with its lowest point in the center $(0, 0)$. If a saw cuts the bowl in half along the x-axis, then the height of all points on the cut is only dependent on the position of the saw on the x-axis (right plot in Figure 3.2). This illustrates that, with respect to partial derivatives, the slope along the x-axis is determined solely by the x-coordinate, while the y-coordinate has no effect on this particular slope. Note that this is due to the independence of the x and y directions, which is not always true. In summary, the derivatives can be calculated separately for

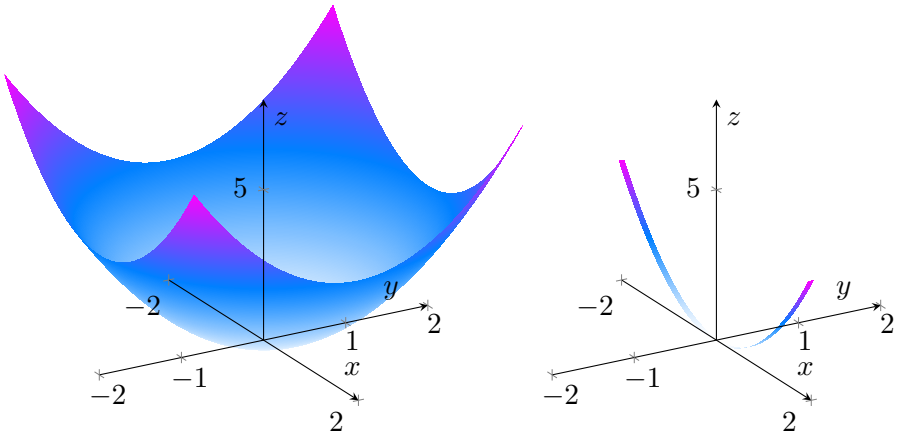



Figure 3.2: On the left is a plot of the function $f(x, y) = x^2 + y^2$. Imagine a saw cutting the figure along the x-axis in half, then all points on the cut are described by $f(x) = x^2$.

each variable and then combined at the end. In this example, a plane could approximate the function at any point, which can be calculated by the partial derivatives of $f$ with respect to $x$ and $y$.

Next, consider a function $f : \mathbb{R}^n \to \mathbb{R}^m$ with multiple outputs, which can be split up into separate functions: the component functions of $f$ are denoted with $f_i : \mathbb{R}^F \to \mathbb{R}$ such that

$$\begin{aligned} f : \mathbb{R}^F &\to \mathbb{R}^{F'} \\ (x_1, \ldots, x_F) &\mapsto (f_1(x_1, \ \ldots, \ x_F), \ldots, f_{F'}(x_1, \ldots, x_F)). \end{aligned} \tag{3.12}$$

Using the previously explained approach, the derivatives $c_k h$ with $c_k \in \mathbb{R}^{1 \times F}$ of the component functions $f_k$ can be calculated. The total derivative of $f$ will then be composed of the derivatives $c_k h$ of the component functions $f_k$, which can be rewritten as a matrix vector product.

In summary, the derivative of a multivariable function is defined as follows (simplified from [MN19, p.91], in accordance with [Shu, p.146]):

**Definition 3.2.1** (Total derivative)**.** Assume $f : \mathbb{R}^F \to \mathbb{R}^{F'}$ is a function with $F, F' \in \mathbb{N}$ and $a \in \mathbb{R}^F$. If there exists a linear function $Df_a : \mathbb{R}^F \to \mathbb{R}^{F'}$ with

$$\lim_{x \to a} \frac{||f(x) - f(a) - Df_a(x - a)||}{||x - a||} = 0, \tag{3.13}$$

then $Df_a$ is called the total derivative of $f$ at point $a$. As $Df_a$ is a linear function, it can be written as a matrix vector product:

$$Df_a(u) = J_f(a)u \tag{3.14}$$

where $J_f(a) \in \mathbb{R}^{F' \times F}$ is the transformation matrix of $Df_a$. The matrix $J_f(a)$ can also be denoted as $\frac{df}{dx}(a)$. Additionally, it is possible to omit the point $a$ and to write $\frac{df}{dx}$, if the point $a$ is clear from the context.

The same way as in the one dimensional case, the numerator in Equation (3.13) can be reformulated [MN19, p.91] to the first order Taylor expansion:

$$f(a+h) \approx f(a) + Df_a(x-a) = f(a) + \frac{df}{dx}(a)h. \tag{3.15}$$

In general, $\frac{df}{dx}$ can be calculated using partial derivatives, as explained in the following theorem [MN19, p.95-96], [Shu, p.158]:

**Theorem 3.2.1** (Calculation of total derivative)**.** *For a differentiable function $f : \mathbb{R}^F \to \mathbb{R}^{F'}$ and $a, u \in \mathbb{R}^F$ the total derivative can be written as*

$$Df_a(u) = J_f(a)u \tag{3.16}$$

*because $Df_a$ is a linear function. The matrix $J_f(a) \in \mathbb{R}^{F \times F'}$ is defined by*

$$(J_f(a))_{ij} = \frac{\partial f_i}{\partial x_j}(a),$$

*where $\frac{\partial f_i}{\partial x_j}$ is the partial derivative of the $i$-th component function $f_i$ with respect to the $j$-th variable [MN19, p.96]. The partial derivatives can be calculated using ordinary one-dimensional calculus rules, as all variables but $x_j$ are considered constant. The matrix $J_f(c) \in \mathbb{R}^{F' \times F}$ is also called Jacobian matrix.*

The previous theorem is illustrated with an example:

$$f(x,y,z) = (x^2 + y, 2x, 4z^2) \tag{3.17}$$

$$f_1(x,y,z) = x^2 + y \qquad f_2(x,y,z) = 2x \qquad f_3(x,y,z) = 4z^2 \tag{3.18}$$

$$\frac{\partial f_1}{\partial x} = 2x \qquad \frac{\partial f_1}{\partial y} = 1 \qquad \frac{\partial f_1}{\partial z} = 0 \tag{3.19}$$

$$\frac{\partial f_2}{\partial x} = 2 \qquad \frac{\partial f_2}{\partial y} = 0 \qquad \frac{\partial f_2}{\partial z} = 0 \tag{3.20}$$

$$\frac{\partial f_3}{\partial x} = 0 \qquad \frac{\partial f_3}{\partial y} = 0 \qquad \frac{\partial f_3}{\partial z} = 8z \tag{3.21}$$

In summary, the Jacobian matrix of $f$ at $a = (2,4,3)$ is equal to:

$$J_f(c) = \begin{pmatrix} 4 & 1 & 0 \\ 2 & 0 & 0 \\ 0 & 0 & 24 \end{pmatrix} \tag{3.22}$$

The calculation of the derivatives using the partial derivatives can be cumbersome. Fortunately, the derivatives of a linear layer can be calculated using simple considerations. Afterwards, the derivatives will also be calculated directly using partial derivatives.

### 3.2.1 Derivatives of a Linear Layer

If a function can be written in the form of the first order Taylor expansion as in Equation 3.15, the total derivative can be inferred. As an example, consider a linear function $f$ with input $x \in \mathbb{R}^{1\times F}$ and the trainable weight matrix $W \in \mathbb{R}^{F\times F'}$:

$$f(x) = xW. \tag{3.23}$$

The row vector $x$ can be interpreted as one sample of a training set. A linear layer might also contain a bias $b$, which is added to each training sample:

$$g(x) = xW + b \tag{3.24}$$

By extending the input $x$ with a one, the vector $b \in \mathbb{R}^{1\times F'}$ can be integrated into the matrix $W$:

$$\hat{x} = (x_1, \dots, x_F, 1) \in \mathbb{R}^{1\times(F+1)} \tag{3.25}$$

$$(x_1, \dots, x_F, 1)\begin{pmatrix} W \\ b \end{pmatrix} = (x_1, \dots, x_F)W + 1b \tag{3.26}$$

In the following, the bias term is not considered as this reformulation is always possible.

If all training samples (row vectors) are stacked to a matrix $X \in \mathbb{R}^{N\times F}$, the output of the linear function for all samples could also be calculated by $f(X) \in \mathbb{R}^{N\times F'}$. The total derivative with respect to the weights can be calculated by assuming $x$ is constant and $W$ is the only parameter of the function. The first order Taylor expansion can be calculated by evaluating the function at $W + h$ as in Equation 3.28:

$$f(W) = xW \tag{3.27}$$

$$f(W + h) = x(W + h) = xW + \underbrace{x}_{=\frac{df}{dW}} h \tag{3.28}$$

The expression in Equation 3.28 is in the same format as the first-order Taylor expansion. The derivative with respect to the weight matrix can be inferred, it is $Df_W(h) = xh$, which is independent of $W$. If $f(W) = XW$, then the total derivative with respect to $W$ is $Xh$ (which agrees with [Shu, p.162]). The derivative with respect to the input sample $x$ can be calculated in the same manner:

$$f(x) = xW \tag{3.29}$$

$$f(x + h) = (x + h)W \tag{3.30}$$

$$= xW + hW \tag{3.31}$$

It can be concluded, that the derivative of $f(x)$ with respect to $x$ is $Df_x(h) = hW$.

### 3.2.2 Total derivative and Gradient

With the total derivative and its calculation cleared, the term gradient can now be defined. It is an important term, because it is used in the steepest descent algorithm. There exist multiple definitions and the following aligns with [MN19, p. 97]:

**Definition 3.2.2** (Gradient)**.** The transpose of the Jacobian matrix $J_L(c)$ of a function $L : \mathbb{R}^{F\times 1} \to \mathbb{R}$ is called the gradient of $L$ at $c \in \mathbb{R}^{F\times 1}$:

$$\nabla L(c) := J_L(c)^\top \in \mathbb{R}^{F\times 1} \qquad \text{such that} \qquad DL_a(h) = J_L(c)h = (\nabla L(c))^\top h = \langle \nabla L(c), h\rangle$$

As the gradient is just the transpose of the Jacobian matrix, a lot of theorems are applicable to gradients as well. For example, the chain rule, which is used in the backpropagation algorithm [Shu, p.151], [MN19, p. 101]:

**Theorem 3.2.2** (Chain Rule)**.** *For $f : \mathbb{R}^F \to \mathbb{R}^{F'}$ and $g : \mathbb{R}^{F'} \to \mathbb{R}^{F''}$. If the composition $(g \circ f)(a)$ is differentiable at $a$, then the derivative is:*

$$D(g \circ f)_a = Dg_{f(a)} \circ Df_a \tag{3.32}$$

If both total derivatives have the form $J(c)h$, then the composite derivative can be calculated by multiplying the Jacobian matrices of the total derivatives:

$$Df_a(h) = J_f(a)h = Ah \qquad Dg_{f(a)}(h) = J_g(f(a))h = Bh \tag{3.33}$$

$$D(g \circ f)_a(h) = BAh \tag{3.34}$$

Equation 3.34 can be adapted for gradients, by taking the transpose of $BA$:

$$\nabla(g \circ f) = (BA)^\top = A^\top B^\top = A^\top(\nabla g) \tag{3.35}$$

In summary, if a gradient and a total derivative is combined, the Chain Rule can be formalized as follows:

**Corollary 3.2.1** (Chain Rule 2)**.** *If $L : \mathbb{R}^{F'} \to \mathbb{R}$ and $f : \mathbb{R}^F \to \mathbb{R}^{F'}$ are differentiable, then the gradient of the composition $L \circ f$ is:*

$$\nabla(L \circ f) = (B(a)^\top)\nabla L \tag{3.36}$$

*if $Df_a(h) = B(a)h$.*

With these theorems at hand, one of the general equations used in the backpropagation algorithm for linear layers $Y = f(X) = XW$ can be derived by combining the previous Corollary 3.2.1 and Equation 3.28:

$$\nabla_W(L \circ f) = X^\top \nabla_Y L \tag{3.37}$$

where $\nabla_X$ represents the gradient with respect to the input $X$ of $f$ and $\nabla_W$ represents the gradient with respect to the weight matrix $W$ of $f$. Backpropagation is the technique

used to calculate the gradient of a neural network, which will be explained in detail in Section 3.3. Using Equation 3.37 a neural network with one linear layer $f$ and an arbitrary differentiable loss function $L$ could be trained. This will be generalized to a neural network with an arbitrary count of layers.

The second equation used in backpropagation for linear layers is

$$\nabla_X(L \circ f) = \nabla L \; W^\top, \tag{3.38}$$

which will be proved later. Compared to the first equation, only the order changed. It can be concluded, that this difference might be because $Df(h) = hW$ if $X$ is considered constant and not $Df(h) = Wh$. Therefore the following conjecture is made:

**Hypothesis** If the total derivative has the form $Df_a(h) = hB(a)$ then the chain rule evaluates to:

$$\nabla(L \circ f) = \nabla L(B(a)^\top) \tag{3.39}$$

The considerations used in this derivation do not always work. The approach from theorem 3.2.1 is always applicable.

### 3.2.3 Derivatives of a Linear Layer using Partial Derivatives

The calculation of multivariable derivatives directly using partial derivatives can be cumbersome, because the expressions can be very long. To make the expressions shorter, a variant of the Einstein convention is used: if an index appears more than once in an expression and is not defined, a sum will be added:

$$\text{for } a, b \in \mathbb{R}^N: \qquad \sum_{i=1}^{N} a_i b_i \longrightarrow a_i b_i \tag{3.40}$$

As an example, a matrix multiplication of $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{m \times k}$ can be written as:

$$(AB)_{ij} = A_{i:}B_{:j} = \sum_{l=1}^{m} A_{il} \cdot B_{lj} \longrightarrow A_{il}B_{lj} \tag{3.41}$$

where $A_{i:}$ denotes the $i-$th row of $A$ and $B_{:j}$ is the $j-$th column of $B$. It is important to note, that an expression $A_{il}B_{lj}$ can be transformed back to a matrix multiplication $(AB)_{ij}$, if the first index of the first factor is $i$ and the second index of the second factor is $j$. Of course the remaining indices must be equal, in this example they are $l$. Furthermore, the chain rule can be formulated with partial derivatives for the functions $f$ and $g$ as defined in theorem 3.2.1:

$$(J_{(g \circ f)})_{ij} = \sum_{m=0}^{F} (J_g)_{im} \cdot (J_f)_{mj} = \sum_{m=0}^{F} \frac{\partial g_i}{\partial x_m} \frac{\partial f_m}{\partial x_j} \longrightarrow \frac{\partial g_i}{\partial x_m} \frac{\partial f_m}{\partial x_j} \tag{3.42}$$

By defining $y = f(c)$, the chain rule can be written compactly as:

$$\frac{\partial g_i}{\partial y_m} \frac{\partial y_m}{\partial x_j} \tag{3.43}$$

In the following, the derivative of a linear layer in the context of backpropagation is derived explicitly using partial derivatives, which is a common way to derive the backpropagation equations. Note, that these derivations will have the same results as in the previous section. However, the derivations with partial derivatives will be needed to derive statements about the derivatives of vectorized expressions.

Using the Einstein convention, the output $Y$ of a linear layer can be written as (see Equation 3.41):

$$Y_{ij} = X_{im} W_{mj} \tag{3.44}$$

Taking the partial derivative with respect to the weight matrix yields:

$$\frac{\partial Y_{ij}}{\partial W_{ko}} = \frac{\partial X_{im} W_{mj}}{\partial W_{ko}} = X_{im} \delta_{mk} \delta_{jo} = X_{ik} \delta_{jo} \tag{3.45}$$

Note, that it can be directly inferred from (3.45), that the second derivative $\frac{\partial X'_{ij}}{\partial W_{ko} \partial W_{pq}}$ will be zero. Assuming a neural network consisting of one linear layer and an arbitrary differentiable loss, the derivative of the loss $L = L(Y)$ with respect to the weight matrix $W$ can be calculated using the chain rule:

$$\frac{\partial L}{\partial W_{ko}} = \frac{\partial L}{\partial Y_{ij}} \frac{\partial Y_{ij}}{\partial W_{ko}} \overset{(3.45)}{=} \frac{\partial L}{\partial Y_{ij}} X_{ik} \delta_{jo} = \frac{\partial L}{\partial Y_{io}} X_{ik} \tag{3.46}$$

To write the expression 3.46 using a matrix multiplication (as in Equation 3.41), the first index of the first factor needs to be $k$ and the second index of the second factor needs to be $o$ (as the expression $\frac{\partial L}{\partial W_{ko}}$ is indexed by $ko$). If the factors are swapped, the second index of the second factor is correct. To get the correct index in the first factor, the transpose needs to be applied. These steps are summarized in Equation 3.47:

$$\frac{\partial L}{\partial W_{ko}} = \frac{\partial L}{\partial Y_{io}} X_{ik} = X_{ik} \frac{\partial L}{\partial Y_{io}} = (X^\top)_{ki} \frac{\partial L}{\partial Y_{io}} \rightarrow \frac{\partial L}{\partial W} = X^\top \frac{\partial L}{\partial Y} \tag{3.47}$$

The result in Equation 3.47 is the same as in the previous section, written in Equation 3.37.

In the backpropagation algorithm, the derivatives of a linear layer with respect to the input $X$ are also needed. They can be calculated in a similar fashion:

$$\frac{\partial Y_{ij}}{\partial X_{ko}} = \frac{\partial X_{im} W_{mj}}{\partial X_{ko}} = W_{mj} \delta_{ik} \delta_{mo} = W_{oj} \delta_{ik} \tag{3.48}$$

$$\frac{\partial L}{\partial X_{ko}} = \frac{\partial L}{\partial Y_{ij}} \frac{\partial Y_{ij}}{\partial X_{ko}} = \frac{\partial L}{\partial Y_{ij}} W_{oj} \delta_{ik} = \frac{\partial L}{\partial Y_{kj}} W_{oj} \tag{3.49}$$

As before, the last expression in Equation 3.49 can also be written using a matrix multiplication. As $\frac{\partial L}{\partial X_{ko}}$ is indexed by $ko$, the first index of the first factor must be $k$ and the second index of the second factor must be $o$. The index of the first factor is correct, but the indices of $W_{oj}$ need to be swapped. Applying a transpose to $W$ yields an expression, which can be transformed to a matrix multiplication:

$$\frac{\partial L}{\partial Y_{kj}} W_{oj} = \frac{\partial L}{\partial Y_{kj}} (W^\top)_{jo} \rightarrow \frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} (W^\top) \tag{3.50}$$

In summary, the equations needed for the backpropagation algorithm for linear layers are again:

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} W^\top \qquad\qquad \frac{\partial L}{\partial W} = X^\top \frac{\partial L}{\partial Y} \tag{3.51}$$

## 3.3 Backpropagation

With the derivatives of a linear layer, the gradient of a Multi-Layer-Perceptron (MLP) can be calculated. A MLP consists of linear layers, each one followed by an non-linear differentiable activation function. Therefore, the derivatives of activation layers is discussed briefly.

The derivative of an activation layer $Y = \sigma(X)$ with a nonlinear differentiable function $\sigma : \mathbb{R} \rightarrow \mathbb{R}$, which is applied element-wise $Y_{ij} = \sigma(X_{ij})$ can be calculated with partial derivatives:

$$\frac{\partial Y_{ij}}{\partial X_{ko}} = \frac{\partial \sigma(X_{ij})}{\partial X_{ko}} = \sigma'(X_{ij}) \delta_{ik} \delta_{jo} \tag{3.52}$$

Consider a neural network consisting of an activation layer and a loss $L$. The derivative with respect to the input is the element-wise multiplication of the derivative of the loss with $\sigma'(X)$ as derived in Equation 3.53:

$$\frac{\partial L}{\partial X_{ij}} = \frac{\partial L}{\partial Y_{ij}} \frac{\partial Y_{ij}}{\partial X_{ko}} = \frac{\partial L}{\partial Y_{ij}} \sigma'(X_{ij}) \delta_{ik} \delta_{jo} = \frac{\partial L}{\partial Y_{ij}} \sigma'(X_{ij}) \Rightarrow \frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} \odot \sigma'(X) \tag{3.53}$$

The element-wise multiplication is denoted with $\odot$. The following example illustrates Equation 3.53 with a one layer neural network consisting of an activation layer, where the loss is the sum of the squared differences between the actual output $Y_i$ and the expected output $Z_i$:

$$X = (a, b) \qquad\qquad Z = (d, e)$$

$$Y = \sigma(X)$$

$$L = \sum_{i=1}^{2} (Y_i - Z_i)^2 = (\sigma(a) - d)^2 + (\sigma(b) - e)^2$$

$$\frac{\partial L}{\partial a} = 2(\sigma(a) - d) \cdot \sigma'(a) \qquad\qquad \frac{\partial L}{\partial b} = 2(\sigma(b) - e) \cdot \sigma'(b)$$

$$\frac{\partial L}{\partial X} = \begin{pmatrix} 2(\sigma(a) - d) \\ 2(\sigma(b) - e) \end{pmatrix} \odot \begin{pmatrix} \sigma'(a) \\ \sigma'(b) \end{pmatrix}$$

Now, all equations to calculate derivatives of an MLP are defined. A simple example is used to illustrate how the equations are put together. First, consider a two-layer neural network, where the output of the first layer is $X^1 = f(X)$ and the output of the second layer is $X^2 = g(X^1)$. Note that the numbers in $X^1$ and $X^1$ are not exponents, they are only only captions. The chain rule is applied to calculate the derivative of the loss $L = L(X^2)$ with respect to a parameter $W^2$ of the second layer $X^2$. The result in Equation 3.54 shows, that the derivative of the first layer is irrelevant.

$$\frac{\mathrm{d}L}{\mathrm{d}W^2} = \frac{\mathrm{d}L}{\mathrm{d}X^2}\frac{\mathrm{d}X^2}{\mathrm{d}W^2} \tag{3.54}$$

Note, that the derivative $\dfrac{\mathrm{d}X^2}{\mathrm{d}W^2}$ is calculated at $X^1$, which is the only influence of the first layer.

Next, assume the first layer has a parameter $W^1$. The derivative of the loss with respect to $W^1$ as shown in Equation 3.55 contains the derivative of the second layer $X^2 = g(X^1)$ with respect to its input $X^1$ due to the chain rule:

$$\frac{\mathrm{d}L}{\mathrm{d}W^1} = \underbrace{\frac{\mathrm{d}L}{\mathrm{d}X^2}\frac{\mathrm{d}X^2}{\mathrm{d}X^1}}_{=:\frac{\mathrm{d}L}{\mathrm{d}X^1}}\frac{\mathrm{d}X^1}{\mathrm{d}W^1} = \frac{\mathrm{d}L}{\mathrm{d}X^1}\frac{\mathrm{d}X^1}{\mathrm{d}W^1} \tag{3.55}$$

The quantity $\dfrac{\mathrm{d}L}{\mathrm{d}X^1}$ is the derivative of the loss with respect to the output of the first layer (which is the input for the second layer). It is sometimes also called the upstream derivative or upstream gradient. In general, the upstream gradient is the gradient of the loss with respect to the output of the currently considered layer.

If there would be another layer $X^0 = h(X)$ before the first layer with parameter $W^0$, then the derivative of the loss with respect to a parameter $W^0$ of that layer would be the product of the derivatives of the loss and the previous layers, each with respect to their input, lastly multiplied with the derivative of the layer $X^0$ with respect to its parameter $W^0$ as shown in Equation 3.56:

$$\frac{\mathrm{d}L}{\mathrm{d}W^0} = \underbrace{\frac{\mathrm{d}L}{\mathrm{d}X^2}\frac{\mathrm{d}X^2}{\mathrm{d}X^1}\frac{\mathrm{d}X^1}{\mathrm{d}X^0}}_{=:\frac{\mathrm{d}L}{\mathrm{d}X^0}}\frac{\mathrm{d}X^0}{\mathrm{d}W^0} \tag{3.56}$$

In general, the calculation of derivatives with respect to parameters of the $l$-th layer $X^l$ always involves the derivative of the loss with respect to the output $X^l$ of the $l$-th layer, as summarized in Equation 3.57:

$$\frac{\mathrm{d}L}{\mathrm{d}W^l} = \frac{\mathrm{d}L}{\mathrm{d}X^l}\frac{\mathrm{d}X^l}{\mathrm{d}W^l} \tag{3.57}$$

For the calculation of the derivatives of the previous layer $l-1$, the upstream derivative $\dfrac{\mathrm{d}L}{\mathrm{d}X^l}$ is used to calculate the derivative of the loss with respect to the output of the

previous layer $l-1$ as shown in Equation 3.58:

$$\frac{\mathrm{d}L}{\mathrm{d}X^{l-1}} = \frac{\mathrm{d}L}{\mathrm{d}X^{l}} \frac{\mathrm{d}X^{l}}{\mathrm{d}X^{l-1}} \tag{3.58}$$

In summary, the calculation of the derivatives (or gradients) with respect to parameters of any layer can be considered an iterative process. Starting at the last layer, the derivative of the loss with respect to any parameter and with respect to the input is calculated (which is the output of the previous layer). The derivative of the loss with respect to the input of the last layer can be used to calculate the derivatives of the next layer and so on.

Therefore it is necessary to know how to calculate the derivatives with respect to the parameters and the input of each layer of a neural network, which has been already explained and resulted in the equations:

$$\frac{\partial L}{\partial W} = X^\top \frac{\partial L}{\partial Y} \qquad \frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} W^\top \qquad \frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} \odot \sigma'(X) \tag{3.59}$$

where $X$ is the input and $\dfrac{\partial L}{\partial Y}$ is the upstream gradient. These equations can be used to calculate the gradients of a MLP.

### 3.3.1 Example: A two layer MLP

To illustrate the process of backpropagation, the gradients of a two layer MLP are calculated in the following. The following equations define a two layer MLP which is trained with the Mean-Square-Error (MSE) as loss function on the dataset $(x_0, y_0), \ldots, (x_N, y_N)$, where $x_i \in \mathbb{R}^{1\times F}$ and $y_i \in \mathbb{R}^N$. In the following, positive numbers in superscripts are captions (no exponents). The output of the MLP can be calculated using the following equations:

$$a^0 = X = (x_0^\top \; x_1^\top \; \ldots \; x_n^\top)^\top \in \mathbb{R}^{N\times F} \tag{3.60}$$

$$s^1 = a^0 W^1 \qquad a^1 = \sigma(s^1) \tag{3.61}$$

$$s^2 = a^1 W^2 \qquad a^2 = \sigma(s^2) \tag{3.62}$$

$$L = \sum_{i=1}^{N} (a_i - Y_i)^2 \tag{3.63}$$

The weights $W^1$ and $W^2$ need to be optimized, therefore the gradient of the loss with respect to these weight matrices is needed. Using the backpropagation Equations (3.53) and (3.51), the gradients for these matrices can be derived.

First, the gradient with respect to the input $a^2$ of the loss is calculated in Equation 3.64. This gradient is used to calculate the gradient $\nabla L_{s^2}$ in Equation 3.65 of the previous layer $a^2 = \sigma(s^2)$ with respect to its input $s^2$, by applying Equation 3.53.

$$\nabla L_{a^2} = 2(a^2 - Y) \tag{3.64}$$

$$\nabla L_{s^2} = \nabla L_{a^2} \odot \sigma'(s^2) \tag{3.65}$$

The linear layer with output $s^2$ is next. The gradient with respect to the weight matrix $W^2$ in Equation 3.66 can be calculated by applying Equation 3.37 using $\nabla L_{s^2}$ and the input $a^1$. The gradient with respect to the input of the linear layer $s^2$ in Equation 3.67 is calculated with Equation 3.51.

$$\nabla L_{W^2} = (a^1)^\top (\nabla L_{s^2}) \tag{3.66}$$

$$\nabla L_{a^1} = (\nabla L_{s^2}) (W^2)^\top \tag{3.67}$$

The gradients for the first linear layer are calculated in the same manner:

$$\nabla L_{s^1} = \nabla L_{a^1} \odot \sigma'(s^1) \tag{3.68}$$

$$\nabla L_{W^1} = (a^0)^\top (\nabla L_{s^1}) \tag{3.69}$$

The neural network could be trained using these gradients and Equation GD-1.

### 3.3.2 Backpropagation for Graph Neural Networks

Graph Neural Networks use message passing layers, which are implemented by multiplying the sparse normalized adjacency matrix $\tilde{A}$ with the input:

$$Y = \tilde{A}X \rightarrow Y_{ij} = \tilde{A}_{im} X_{mj} \tag{3.70}$$

For the backward propagation, the derivative of 3.70 with respect to the input $X$ is needed:

$$\frac{\partial Y_{ij}}{\partial X_{ko}} = \frac{\partial \tilde{A}_{im} X_{mj}}{\partial X_{ko}} = \tilde{A}_{im} \delta_{mk} \delta_{jo} = \tilde{A}_{ik} \delta_{jo} \tag{3.71}$$

Combining 3.71 with the loss:

$$\frac{\partial L}{\partial X_{ko}} = \frac{\partial L}{\partial Y_{ij}} \frac{\partial Y_{ij}}{\partial X_{ko}} = \frac{\partial L}{\partial Y_{ij}} \tilde{A}_{ik} \delta_{jo} = \frac{\partial L}{\partial Y_{io}} \tilde{A}_{ik} \tag{3.72}$$

which can be rewritten the same way as in Equation 3.47. To rewrite the expression using a matrix multiplication like in 3.41, the first index of the first factor must be $k$ and the second index of the second factor must be $o$. The order of the factors can be changed in Equation (3.72) to have the correct indices in the second factor. If the transpose is applied, the first index of the first factor is also correct and the expression can be written as a matrix multiplication:

$$\tilde{A}_{ik} \frac{\partial L}{\partial Y_{io}} = \left(\tilde{A}^\top\right)_{ki} \frac{\partial L}{\partial Y_{io}} \longrightarrow \frac{\partial L}{\partial X} = \tilde{A}^\top \frac{\partial L}{\partial Y} \tag{3.73}$$

Equation 3.73 is the same as $\frac{\partial L}{\partial W}$ in Equation 3.47 for linear layers if $X$ is replaced with $\tilde{A}$ and $W$ is replaced with $X$.

### 3.3.3 Example: A two layer Graph Convolutional Network

A GCN can be described using the following equations, where $X = (x_0^\top \ \dots \ x_n^\top)^\top \in \mathbb{R}^{N \times F}$ is the matrix consisting of the stacked node feature vectors and $\tilde{A}$ is a normalized version of the adjacency matrix:

$$a^0 = X \tag{3.74}$$

$$t^1 = \tilde{A}a^0 \qquad s^1 = t^1 W^1 \qquad a^1 = \sigma(s^1) \tag{3.75}$$

$$t^2 = \tilde{A}a^1 \qquad s^2 = t^2 W^2 \qquad a^2 = \sigma(s^2) \tag{3.76}$$

$$L = \sum_{i=1}^{N} (a_i - Y_i)^2 \tag{3.77}$$

The superscripts in $a, t, s$ are captions (no exponents). Using Equations 3.73, (3.53) and (3.51), the derivatives can be calculated:

$$\nabla L_{W^2} = (t^2)^\top (\nabla L_{a^2} \odot \sigma'(s^2)) \tag{3.78}$$

$$= ((a^1)^\top \tilde{A}^\top)(\nabla L_{a^2} \odot \sigma'(s^2)) \tag{3.79}$$

$$\nabla L_{W^1} = ((t^1)^\top)((\tilde{A}^\top (\nabla L_{a^2} \odot \sigma'(s^2)(W^2)^\top)) \odot \sigma'(s^1)) \tag{3.80}$$

$$= (X^\top \tilde{A}^\top)((\tilde{A}^\top (\nabla L_{a^2} \odot \sigma'(s^2)(W^2)^\top)) \odot \sigma'(s^1)) \tag{3.81}$$

The adjacency matrix will be part in the gradients $\nabla L_{a^l}$, but in the expression for the last layer.

### 3.3.4 Hessian matrix and Vectorization

The second derivative of a real valued function $f : \mathbb{R}^F \to \mathbb{R}$ is can be defined as the matrix containing all second partial derivatives of $f$ [MN19, p.112].

**Definition 3.3.1** (Hessian matrix)**.** Assume $c \in \mathbb{R}^F$ and all partial derivatives

$$\frac{\partial^2 f}{\partial x_i \partial x_j}(c) = \frac{}{\partial x_i} \frac{\partial f}{\partial x_j}(c) \tag{3.82}$$

exist. Then the $F \times F$ matrix $Hf(c)$ is defined as:

$$(Hf(c))_{ij} := \frac{\partial^2 f}{\partial x_i \partial x_j}(c) \tag{3.83}$$

Assume $g = L \circ f$ describes the error of a neural network as a function of a weight matrix $W \in \mathbb{F} \times \mathbb{F}'$. The function $g$ has the domain $g : \mathbb{F} \times \mathbb{F}' \to \mathbb{R}$. Therefore it is not possible to calculate the Hessian matrix as described in the previous definition. It would be necessary to reshape the matrix $W$ to a vector.

From another perspective, the first derivative of $g$ with respect to a weight matrix is a matrix of the same shape. The second derivative would again result in a matrix of the

same shape as the weight matrix for each entry in the weight matrix. Thus, the overall second derivative is a new object, which can be called a tensor.

To avoid the complexity of a higher order tensor, the first derivative is vectorized. In this case, the second derivative will result in a matrix. In this thesis, column vectorization is used: a matrix $A \in \mathbb{R}^{n \times m}$ is vectorized by stacking all columns:

$$\text{vec}(A) \in \mathbb{R}^{n \cdot m \times 1}, \qquad \text{vec}(A)_k := \text{vec}(A)_{j \cdot n + i} = A_{ij} \tag{3.84}$$

where $0 \leq k < n \cdot m, 0 \leq i < n, 0 \leq j < m$. Note, that $k, i$ and $j$ are zero-based indices.

Example:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \\ A_{20} & A_{21} \end{pmatrix} \in \mathbb{R}^{3 \times 2} \qquad \text{vec}(A) = \begin{pmatrix} 1 & 3 & 5 & 2 & 4 & 6 \end{pmatrix}^\top \tag{3.85}$$

The following derivatives of vectorized expressions are needed for the calculation of a second order derivative of a neural network with linear layers:

$$\frac{\partial \,\text{vec}(W)_i}{\partial \,\text{vec}(W)_j} = \delta_{ij} \qquad\qquad \frac{d \,\text{vec}(W)}{d \,\text{vec}(W)} = I \tag{3.86}$$

Furthermore, a vectorization of a matrix multiplication is common in expressions for second derivatives. It can be rewritten using the following rule:

**Theorem 3.3.1** (Vectorization of a Matrix Product)**.** *For two matrices $A \in \mathbb{R}^{k \times l}, B \in \mathbb{R}^{l \times m}$ [Dhr00]:*

$$vec(AB) = (I_m \otimes A)\, vec(B) = (B^\top \otimes I_k)\, vec(A) \tag{3.87}$$

This theorem is very important for the calculation of the second derivative, for example to derive the second derivative of linear expressions.

**Theorem 3.3.2** (Derivative of a linear expression)**.** *If $A$ is independent of $C$, but $B$ is a function of $C$:*

$$\frac{d\, vec(AB)}{d\, vec(C)} = (I_m \otimes A) \left( \frac{d}{d\, vec(C)}\, vec(B) \right) \tag{3.88}$$

*Proof.* First, Theorem 3.3.1 is applied:

$$\frac{d \,\text{vec}(AB)}{d \,\text{vec}(C)} = \frac{d}{d \,\text{vec}(C)} (I_m \otimes A) \,\text{vec}(B) \tag{3.89}$$

The expression $(I_m \otimes A)$ is independent of $C$, leading to the statement of the theorem:

$$\frac{d}{d \,\text{vec}(C)} (I_m \otimes A) \,\text{vec}(B) = (I_m \otimes A) \left( \frac{d}{d \,\text{vec}(C)} \,\text{vec}(B) \right) \tag{3.90}$$

□

As an illustration, this theorem is applied to a vectorized linear layer $Y = XW$ with input $X \in \mathbb{R}^{N \times F}$ and weight matrix $W \in \mathbb{R}^{F \times F'}$:

$$\frac{d\,\mathrm{vec}(XW)}{d\,\mathrm{vec}(W)} = (I_{F'} \otimes X)\frac{d\,\mathrm{vec}(W)}{d\,\mathrm{vec}(W)} = I_{F'} \otimes X \tag{3.91}$$

**Theorem 3.3.3** (Linear Layer and Activation Function Vectorized Derivative)**.** *For an activation function* $\sigma : \mathbb{R} \to \mathbb{R}$ *:*

$$\frac{d\,vec(\sigma(XW))}{d\,vec(W)} = diag\left(\sigma'(vec(XW))\right)(I_m \otimes X) \tag{3.92}$$

*Proof.*

$$\frac{\partial\,\mathrm{vec}(\sigma(X_{ip}W_{pj}))_{jm+i}}{\partial\,\mathrm{vec}(W_{lk})_{kn+l}} = \frac{\partial\sigma(\mathrm{vec}(X_{ip}W_{pj})_{jm+i})}{\partial\,\mathrm{vec}(W_{lk})_{kn+l}} = \sigma'(\mathrm{vec}(X_{ip}W_{pj})_{jm+i})\underbrace{\frac{\partial\,\mathrm{vec}(X_{ip}W_{pj})_{jm+i}}{\partial\,\mathrm{vec}(W_{lk})_{kn+l}}}_{\Rightarrow I_m \otimes X} \tag{3.93}$$

Since the factor $\sigma'(...)$ is multiplied with every row of $I_m \otimes X$, this part can be depicted as:

$$\mathrm{diag}(\sigma'(\mathrm{vec}(XW))) \tag{3.94}$$

□

Furthermore, for activation layers, the following theorem will be used:

**Theorem 3.3.4** (Element-wise Products and Vectorization)**.** *For* $A, B \in \mathbb{R}^{n \times m}$*:*

$$vec(A \odot B) = vec(A) \odot vec(B) \tag{3.95}$$

*Proof.* The element-wise product of $A$ and $B$ can be defined as:

$$(A \odot B)_{ij} = A_{ij} \cdot B_{ij} \tag{3.96}$$

By definition of the vectorization and the previous definition of the element-wise product, the left hand side of Equation 3.95 can be written as:

$$\mathrm{vec}(A \odot B)_{j \cdot n+i} = (A \odot B)_{ij} = A_{ij} \cdot B_{ij} \tag{3.97}$$

Rewriting the statement using vectorization operations yields:

$$A_{ij} \cdot B_{ij} = (\mathrm{vec}(A)_{j \cdot n+i}) \cdot (\mathrm{vec}(B)_{j \cdot n+i}) \tag{3.98}$$

Combining Equation 3.97 with 3.98 and replacing $j \cdot n + i$ with a variable $m$ makes the equality clear:

$$\mathrm{vec}(A \odot B)_m = \mathrm{vec}(A) \cdot \mathrm{vec}(B)_m \to \mathrm{vec}(A \odot B) = \mathrm{vec}(A) \odot \mathrm{vec}(B) \tag{3.99}$$

□

While the previous theorems are enough to calculate the block diagonals of the second derivative of neural networks, for the calculation of off-diagonal blocks the following statement is needed:

**Theorem 3.3.5** (Derivative of a transpose)**.** *For $W \in \mathbb{R}^{n\times m}$, $vec(W), vec(W^\top) \in \mathbb{R}^{n\cdot m\times 1}$*

$$S_{ko} := \frac{\partial\, vec(W^\top)_k}{\partial\, vec(W)_o} \in \mathbb{R}^{nm\times nm} \tag{3.100}$$

$$S_{ko} = \begin{cases} 1 & \text{if } k//m = o\%n, l\%m = o//n \\ 0 & \text{else} \end{cases} \tag{3.101}$$

*in accordance with [MN19, p.54, p.204], where // is the integer division and % is the modulo operator.*

*Proof.* For $k = lm + p$, $o = jn + i$ with $l = k//m$, $p = l\%m$, $j = o//n$, $i = o\%n$:

$$\frac{\partial \operatorname{vec}((W^\top)_{pl})_k}{\partial \operatorname{vec}(W_{ij})_o} = \frac{\partial \operatorname{vec}(W_{lp})_k}{\partial \operatorname{vec}(W_{ij})_o} = \delta_{li}\delta_{pj} \tag{3.102}$$

□

As $0 \le l \le n$ and $0 \le p < m$, the following algorithm will calculate $S$:

**Data:** Input data: $n, m \in \mathbb{N}_{>0}$
**Result:** The matrix $S$
Initialize $S \in \mathbb{R}^{nm\times nm}$ with zeros;
**while** $l = 0, \dots, n;\ p = 0, \dots, m$ **do**
  $k = lm + p$;
  $o = pn + l$;
  $S_{ko} = 1$
**end**

**Algorithm 1:** An algorithm to calculate the matrix $S$ from theorem 3.3.5

**Corollary 3.3.1** (Transpose and Vectorization)**.** *For $W$ and $S$ as defined before:*

$$vec(W^\top) = S\, vec(W) \tag{3.103}$$

*Proof.* The matrix $S$ is a permutation matrix, because every row has exactly one nonzero entry.

$$\frac{\mathrm{d}\operatorname{vec}(W^\top)}{\mathrm{d}\operatorname{vec}(W)} = S = S\frac{\mathrm{d}\operatorname{vec}(W)}{\mathrm{d}\operatorname{vec}(W)} = \frac{\mathrm{d}(S\operatorname{vec}(W))}{\mathrm{d}\operatorname{vec}(W)} \tag{3.104}$$

□

### 3.3.5 Second derivatives

As already noted, the second derivative will be calculated using the vectorized first derivative. The first derivative is calculated using the respective backpropagation equations, which all consist of matrix multiplications and element-wise multiplications. Positive numbers in superscript in this section represent captions (no exponents).

Therefore the second derivative needs to be calculated using the product-rule [PP12] for the matrix multiplication and the element-wise multiplication. Assume $X$ and $Y$ are matrix functions in dependence of the variable $Z$:

$$\frac{\partial(XY)}{\partial Z} = \left(\frac{\partial X}{\partial Z}\right) Y + X \left(\frac{\partial Y}{\partial Z}\right) \tag{MPR}$$

$$\frac{\partial(X \odot Y)}{\partial Z} = \left(\frac{\partial X}{\partial Z}\right) \odot Y + X \odot \left(\frac{\partial Y}{\partial Z}\right) \tag{EPW}$$

These rules are also valid if $X, Y, Z$ are vectors. In the following, a neural network consisting of $l$ layers is considered. The input for the neural network is $X^0$ and the output of the layer $l$ is $X^l$. For simplicity, it is assumed, that the gradient is a product of matrices:

$$\frac{\mathrm{d}L}{\mathrm{d}W^k} = \frac{\mathrm{d}L}{\mathrm{d}X^l} \frac{\mathrm{d}X^l}{\mathrm{d}X^{l-1}} \cdots \frac{\mathrm{d}X^k}{\mathrm{d}W^k} \tag{3.105}$$

The Hessian $H$ of the vectorized gradient consists of blocks, because the gradient actually consists of the gradients with respect to each parameter of the neural network concatenated:

$$\Theta = (W^1, W^2, \ldots, W^l) \tag{3.106}$$

$$\frac{\partial L}{\partial \Theta} = \left(\frac{\partial L}{\partial W^1}, \ldots, \frac{\partial L}{\partial W^l}\right) \tag{3.107}$$

$$H_{ij} = \frac{\partial}{\partial W^j} \frac{\partial L}{\partial W^i} \tag{3.108}$$

For Neural Networks like MLPs the matrix $H$ is symmetric, because of Schwarz's theorem: $H_{ij} = H_{ji}$. As the matrix $H$ is very large, it is common to approximate it. In the following, the block diagonals $H_{ii}$ are investigated. This means, that the derivative of the gradient from Equation 3.105 needs to be calculated, with respect to $W^k$.

The product rules from Equations (MPR) and (EPW) are applied to Equation (3.105). It is important to clarify, which factors in Equation 3.105 are dependent on $W^k$. If the considered neural network is a MLP, then all linear layers, which are part of the expression, are independent of $W^k$. Furthermore only the activation layers and the loss function have a nonzero second derivative. As an example, for a four layer neural network, where layers dependent on $W^0$ are denoted with $Y$ and all other layers are denoted with $X$, the gradient is:

$$\frac{\mathrm{d}L}{\mathrm{d}W^0} = \frac{\mathrm{d}L}{\mathrm{d}Y^4} \frac{\mathrm{d}Y^4}{\mathrm{d}X^3} \frac{\mathrm{d}X^3}{\mathrm{d}Y^2} \frac{\mathrm{d}Y^2}{\mathrm{d}X^1} \frac{\mathrm{d}X^1}{\mathrm{d}W^0} \tag{3.109}$$

Applying the product rule will result in a sum:

$$\frac{\mathrm{d}}{\mathrm{d}W^0}\frac{\mathrm{d}L}{\mathrm{d}W^0} = \frac{\mathrm{d}}{\mathrm{d}W^0}\left(\frac{\mathrm{d}L}{\mathrm{d}Y^4}\frac{\mathrm{d}Y^4}{\mathrm{d}X^3}\frac{\mathrm{d}X^3}{\mathrm{d}Y^2}\frac{\mathrm{d}Y^2}{\mathrm{d}X^1}\frac{\mathrm{d}X^1}{\mathrm{d}W^0}\right) \tag{3.110}$$

$$= \left(\frac{\mathrm{d}}{\mathrm{d}W^0}\frac{\mathrm{d}L}{\mathrm{d}Y^4}\right)\frac{\mathrm{d}Y^4}{\mathrm{d}X^3}\frac{\mathrm{d}X^3}{\mathrm{d}Y^2}\frac{\mathrm{d}Y^2}{\mathrm{d}X^1}\frac{\mathrm{d}X^1}{\mathrm{d}W^0} \tag{3.111}$$

$$+ \frac{\mathrm{d}L}{\mathrm{d}Y^4}\left(\frac{\mathrm{d}}{\mathrm{d}W^0}\frac{\mathrm{d}Y^4}{\mathrm{d}X^3}\right)\frac{\mathrm{d}X^3}{\mathrm{d}Y^2}\frac{\mathrm{d}Y^2}{\mathrm{d}X^1}\frac{\mathrm{d}X^1}{\mathrm{d}W^0} \tag{3.112}$$

$$+ \frac{\mathrm{d}L}{\mathrm{d}Y^4}\frac{\mathrm{d}Y^4}{\mathrm{d}X^3}\frac{\mathrm{d}X^3}{\mathrm{d}Y^2}\left(\frac{\mathrm{d}}{\mathrm{d}W^0}\frac{\mathrm{d}Y^2}{\mathrm{d}X^1}\right)\frac{\mathrm{d}X^1}{\mathrm{d}W^0} \tag{3.113}$$

It is clearly visible, that the calculation of the second derivative results in much larger expressions than the first derivative.

### A one layer MLP

Assuming a neural network with one linear layer with weight matrix $W \in \mathbb{R}^{F\times F'}$, input matrix $X \in \mathbb{R}^{N\times F}$. The output of the first layer is $Y = XW$. The loss function $L$ is twice differentiable as well as the activation function $\sigma : \mathbb{R} \to \mathbb{R}$. The overall function describing the loss of this neural network is: $L(\sigma(XW))$. With $Z = \sigma(Y)$, the gradient with respect to the weights is:

$$\frac{\partial L}{\partial W} = (X)^\top\left(\frac{\partial L}{\partial Z} \odot \sigma'(XW)\right) \tag{3.114}$$

The second derivative of the vectorized gradient can be calculated by the following expression:

$$\frac{\partial}{\partial \operatorname{vec}(W)} \operatorname{vec}\left((X)^\top\left(\frac{\partial L}{\partial Z} \odot \sigma'(XW)\right)\right) \tag{3.115}$$

The application of theorem 3.3.1 yields:

$$\left(I \otimes (X)^\top\right)\operatorname{vec}\left(\frac{\partial L}{\partial Z} \odot \sigma'(XW)\right) = \left(I \otimes (X)^\top\right)\left(\operatorname{vec}\left(\frac{\partial L}{\partial Z}\right) \odot \operatorname{vec}\left(\sigma'(XW)\right)\right) \tag{3.116}$$

Assuming the second derivative of the loss is zero, the derivative can be calculated by applying theorem 3.3.3:

$$\tilde{H} = \left(I \otimes (X)^\top\right)\operatorname{diag}\left(\operatorname{vec}\left(\frac{\partial L}{\partial Z}\sigma''(XW)\right)\right)(I \otimes X) \tag{3.117}$$

If the second derivative of the loss is nonzero, then the overall derivative is

$$H = \left(I \otimes (X)^\top\right)\left(\operatorname{vec}\left(\frac{\partial^2 L}{\partial Z\,\partial Z}\right) \odot \operatorname{vec}\left(\sigma'(XW)\right)\right) + \tilde{H} \tag{3.118}$$

If this neural network is modified to a Graph Convolutional Network, then a message passing layer is added before the first linear layer. This is equivalent to replacing the input of the neural network $X$ with $AX$, where $A$ is the normalized adjacency matrix $A \in \mathbb{R}^{N \times N}$.

### A two layer MLP

To gain a better understanding of the structure of the vectorized second derivative of MLPs, the second derivative is calculated for a two layer MLP. To make the equations easier to read, the second derivative of the loss function is assumed to be zero, which is not a realistic assumption. If the second derivative is considered, an additional term needs to be calculated.

The notation is the same as in Section 3.3.1. The derivatives with respect to the weight matrix of the second layer will have the same form as for a one layer MLP, which was calculated in Equation 3.117. The results are comparable, because the gradient does not contain any expressions from the first layer, only a different input.

The Hessian block for the weight matrix of the first layer differs. The gradient with respect to $W^1$ is:

$$\frac{\mathrm{d}L}{\mathrm{d}W^1} = X^\top \left( \left( \frac{\mathrm{d}L}{\mathrm{d}a^2} \odot \sigma'(s^2) \right) (W^2)^\top \odot \sigma'(s^1) \right) \tag{3.119}$$

The second derivative evaluates to:

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}\,\mathrm{vec}(W^1)} \mathrm{vec}\left( \frac{\mathrm{d}L}{\mathrm{d}W^1} \right) &= (I_{F'} \otimes X^\top) \left[ (W^2 \otimes I_N) \frac{\mathrm{d}L}{\mathrm{d}a^2} \underbrace{\frac{\mathrm{d}}{\mathrm{d}\,\mathrm{vec}(W^1)} \mathrm{vec}(\sigma'(s^2))}_{=D_1} \right] \mathrm{vec}(\sigma'(s^1)) \\ &\quad + (I_{F'} \otimes X^\top) \left[ (W^2 \otimes I_N) \frac{\mathrm{d}L}{\mathrm{d}a^2} \mathrm{vec}(\sigma'(s^2)) \right] \left( \underbrace{\frac{\mathrm{d}}{\mathrm{d}\,\mathrm{vec}(W^1)} \mathrm{vec}(\sigma'(s^1))}_{=D_2} \right) \end{aligned} \tag{3.120}$$

where the parts $D_1$ and $D_2$ are:

$$D_1 = \sigma''(((W^2)^\top \otimes I_N) \,\mathrm{vec}(\sigma(s^1))) \left[ ((W^2)^\top \otimes I_N) \,\mathrm{diag}(\sigma'(\mathrm{vec}(a^1)))(I \otimes X) \right] \tag{3.121}$$

$$D_2 = \mathrm{diag}(\sigma'(\mathrm{vec}(s^1)))(I \otimes X) \tag{3.122}$$

Again, this expression can be modified to correspond to a two layer GCN. All occurrences of $X$ as well as all occurrences of $s^1$ and $s^2$ need to be multiplied from the left with $\tilde{A}$, such that the message passing layer will be considered. Furthermore, the gradient would additionally contain the transpose of the adjacency matrix, which also needs to be accounted for in the Hessian block.

# 4 Problem Statement

Graph Neural Networks (GNNs) incorporate the graph structure using message passing layers, which can be implemented by multiplying the normalized adjacency matrix $\tilde{A}$ with the input $X$ as described in Equation 4.1:

$$Y = \tilde{A}X \tag{4.1}$$

GNNs can be trained with first-order optimization algorithms like Stochastic Gradient Descent, which use gradients to adapt the model parameters. The first proposed neural network to incorporate message passing is called Graph Convolutional Network (GCN)[KW17].

**GNNs and Preconditioning** The application of preconditioning to GCNs has been explored in [IFSL20]. The Kronecker-Factored Approximate Curvature (K-FAC) algorithm for preconditioning improved the accuracy of GCNs trained on the Planetoid dataset [IFSL20], indicating that better minima (parameters) have been found. Furthermore, the authors claim that K-FAC can enhance the training convergence. The K-FAC algorithm approximates the Fisher Information matrix, whose inverse is used for preconditioning. Multiplying the gradients by the inverse Fisher matrix transforms them into so-called natural gradients [IFSL20]. This transformation makes the gradients invariant to smooth and invertible reparameterizations of the neural network.

However, there have been no studies to date investigating the effects of preconditioning on other GNN architectures than GCN. This thesis aims to address this gap in research. Furthermore, sparse gradients can be a problem in the optimization of GNNs as mentioned by [ZLM+23] and are discussed in the following.

**Sparsity** In this context, sparsity refers to a property of matrices and vectors. The more zero entries a matrix or a vector has, the sparser it is. Figure 4.1 shows the sparsity of the weight gradients and output gradients (the gradient of the loss with respect to a layer output) during the training of a six-layer Graph Convolutional Network (GCN) on the Cora dataset with ReLU activation functions. A large number of layers was chosen to effectively illustrate the sparsity problem. The learning rate was set to 0.01, the number of hidden channels was 64 (size of the features in the hidden layers) and cross entropy loss (combination of softmax and NLL loss) was used. The sparsity for each layer is presented separately.

The left bottom plot in Figure 4.1 shows the weight matrices sparsity ratio, defined as the ratio of zero elements to the total number of elements in the weight matrix

gradient during training[1]. In the early stages, the ratio is very low for the first three layers. However, for all layers, the sparsity ratio converges to a constant value as training progresses. The right plot on the bottom in Figure 4.1 shows the norm of the weight matrix gradient for each layer during training. The norm is high for all layers in the initial epochs but degrades quickly thereafter. It is important to consider both the sparsity ratio and the norm when investigating sparsity, as a gradient might also contain many entries that are nearly zero.

The sparsity results from multiple factors:

- In node classification tasks, like in the experiment presented in Figure 4.1, the loss is calculated with respect to the training nodes only. Therefore, the gradients of all other nodes in the last layer have an output gradient of zero.
- ReLU activations: this activation function does not suffer from the vanishing gradient problem, which the Sigmoid activation function[DSC22] does. The vanishing gradient problem refers to gradients that are very small or zero. It is also faster computationally, but it can cause sparse gradients when many input features are negative, because the derivative is zero, when the input is negative.

This first issue regarding node classification tasks was also described by [ZLM+23] and is clearly visible in the plots for the output gradients in the top row of Figure 4.1. The sparsity decreases from the last layer to the first layer because the information from the non-training nodes is propagated by message passing to the training nodes. Approximately 5% of the nodes are used for training in the Cora datset, which explains the roughly 95% sparsity in the last layer of the output gradient in Figure 4.1. However, the graphs used for training GNNs are usually, sparse, meaning, that the average number of edges per node is much lower than the total count of nodes. Therefore some nodes may not be reachable by the training nodes at all, leading to a constant sparsity on the output gradient of the first two layers in Figure 4.1.

Note that the sparsity could also be considered a result of a completed training process, i.e. when a global optimum is reached. However, the loss did not reach a global optimum in this experiment because a later experiment (see Figure 6.1) achieved a lower loss. Therefore, the optimization process cannot be considered complete. In essence, the observed sparsity is not the result of reaching global optimum.

Less sparsity could be assumed to positively affect the training process. Sparse gradients are undesirable because they only update parts of the weight matrix. In contrast, a dense gradient updates all entries of a weight matrix, potentially leading to faster convergence.

The sparsity issue can be compared to a mechanic optimizing a machine. If the mechanic only tunes some parts, it might still be possible to improve the machine's performance by later tuning the untouched parts. However, if the mechanic tunes all available parts, the machine can be perfectly optimized. This is summarized in the first research question:

**Research Question 1:** How does reducing the sparsity of the gradient update lead to

[1]Code for this experiment is available at `https://github.com/sfetzel/GNN-Sparsity/blob/main/gcn_grad.py`.

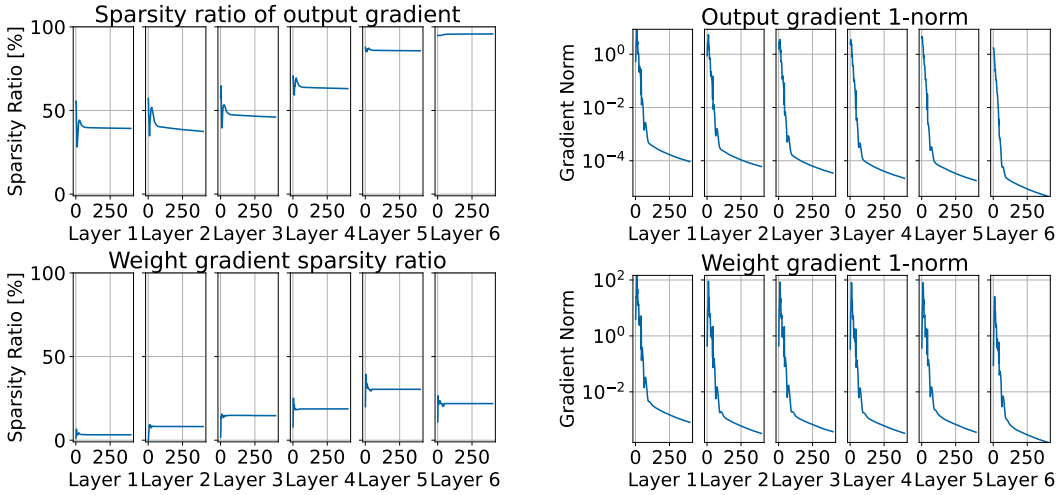



Figure 4.1: Plot created by training a Graph Convolutional Network on the Cora dataset with 64 hidden channels using the Adam optimizer and a learning rate of 0.1 for 400 epochs. The x axes indicate the epochs for each layer. The left hand plot shows the sparsity of the gradients of the loss with respect to the weight matrices of the GCN layers for each epoch. The right hand plot shows the norm of these gradients.

faster convergence and better minimum points in the optimization of GNNs?

Alleviating the sparsity of gradients is a nontrivial problem that can be approached in different ways. One simple solution for graph classification tasks, where the whole graph is used for training, is to use a different activation function. However, ReLU is currently the computationally most efficient activation function and works well in practice. Therefore this thesis sticks with ReLU activation functions.

Another way to address sparsity is to use more training nodes in node classification tasks. However this may lead to models which do not generalize well.

This thesis will focus on architectures with message passing layers. The chosen approach to reduce gradient sparsity is to apply preconditioning. Specifically, the inverse of an approximation to the Hessian matrix $P^{-1}$ will be multiplied with the gradient before applying first-order techniques, as shown in Equation GD-2. Since the Hessian approximation is non-sparse, this reduces the sparsity of the gradient.

Applying K-FAC can improve the convergence speed of the training process but also adds additional costs. For instance, calculating the inverse of a matrix is computationally expensive. Therefore, another research question is whether applying K-FAC to the optimization of GNNs is worthwhile:

**Research Question 2:** How do Natural Gradients improve the efficiency of gradient-based optimization in GNNs?

Efficiency can be compared simply by evaluating average runtimes until convergence. It is likely, that an optimization iteration with K-FAC will take more time per iteration, but fewer iterations will be needed to achieve convergence.

# 5 Methodology

Preconditioning may improve the convergence of the training of neural networks and lead to higher accuracies. Furthermore, it may resolve the issue of sparse gradients. Various preconditioning methods are discussed below and later applied. First, Kronecker-Factored Approximate Curvature (K-FAC) is briefly explained following [MG15]. Subsequently, other preconditioning methods that utilize the Hessian matrix and the Generalized-Gauss-Newton (GGN) matrix are described.

## 5.1 The Fisher Matrix

The basic idea of K-FAC is to derive an expression for a block of the Fisher matrix that can be inverted efficiently with a simple assumption.

First, the motivation and definition of the Fisher matrix are provided. Consider a model $f(x)$ with parameters $\Theta = (\Theta_1, \ldots, \Theta_k) \in \mathbb{R}^k$ trained with samples $(x_1, y_1), \ldots, (x_N, y_N)$, where $x_i \in \mathbb{R}^F$ and $y_i \in \{1, \ldots, C\}$. The model $f$ is trained by maximizing the joint probability that each training sample is classified with the correct label [GBC16, p.131][Hel15, p.198]. This method is also known as the Maximum Likelihood principle, as it selects the model $f$ with the highest likelihood of reproducing the training data.

The joint probability can be calculated by multiplying the probabilities $P(f(x_i) = y_i)$ for each training sample as in Equation 5.1. The parameter $\Theta$ which maximizes this likelihood determines the best model:

$$\Theta = \text{argmax}_\Theta \prod_{i=1}^{N} P\left(f(x_i) = y_i\right) \tag{5.1}$$

In practice it is easier to solve this problem by applying the logarithm function to the joint probability, such that the expression can be rewritten as a sum as in Equation 5.2:

$$\log\left(\prod_{i=1}^{N} P_\Theta(f(x_i) = y_i)\right) = \sum_{i=1}^{N} \log(P(f(x_i) = y_i)) \tag{5.2}$$

The logarithm is a monotonically increasing function, i.e. if $x < y \Rightarrow \log(x) < \log(y)$, therefore a solution maximizing the logarithmic joint probability also maximizes the joint probability. In order to turn the maximization problem into a minimization problem, the negative logarithm is used:

$$L = -\sum_{i=1}^{N} \log(P(f(x_i) = y_i)) \qquad \text{(Negative-Log-Likelihood)}$$

$$\theta = \text{argmin}_\Theta L \tag{5.3}$$

The expression denoted by $L$ is called the Negative Log-Likelihood (NLL), a common loss used in the training of neural networks. To ensure the the output of a neural network is interpretable as a probability, functions such as softmax are used to rescale the output $(z_1, \ldots, z_n)$ of the last layer to conform to the rules of a probability [GBC16, p.57]:

$$\sum_{i=1}^{n} z_i = 1 \qquad\qquad \forall i : z_i \geq 0$$

The softmax function for $z_1, \ldots, z_n$ is defined as [GBC16, p.183]:

$$\hat{z}_i = \exp(z_i) / \left( \sum_{j=1}^{N} \exp(z_j) \right) \tag{Softmax}$$

After the softmax function is applied, the output clearly satisfies the conditions for probabilities. Assume that the softmax function is integrated into a neural network $f$, as the last layer. The second derivative $H_{i,j}$ of a neural network with negative log-likelihood loss can be calculated using the following expression:

$$H_{i,j} = \frac{\partial}{\partial \Theta_i} \frac{\partial}{\partial \Theta_j} \overbrace{\sum_{m=1}^{N} \underbrace{-\log(f(x_m))}_{=:L_m}}^{=:L} = \frac{\partial}{\partial \Theta_i} \sum_{m=1}^{N} \underbrace{-\frac{1}{f(x_m)} \left( \frac{\partial f(x_m)}{\partial \Theta_j} \right)}_{= \frac{\partial}{\partial \Theta_j} L_m}$$

To evaluate the second derivative, the product rule needs to be applied:

$$\begin{aligned} H_{i,j} &= \sum_{m=1}^{N} \left( \frac{\partial}{\partial \Theta_i} \frac{-1}{f(x_m)} \right) \left( \frac{\partial f(x_m)}{\partial \Theta_j} \right) + \sum_{m=1}^{N} \frac{-1}{f(x_m)} \left( \frac{\partial}{\partial \Theta_i} \frac{\partial f(x_i)}{\partial \Theta_j} \right) \\ &= \underbrace{\sum_{m=1}^{N} \left( \frac{1}{f(x_m)^2} \frac{\partial}{\partial \Theta_i} f(x_m) \right) \left( \frac{\partial f(x_m)}{\partial \Theta_j} \right)}_{=:\hat{H}_{i,j}} + \underbrace{\sum_{m=1}^{N} \frac{-1}{f(x_m)} \left( \frac{\partial}{\partial \Theta_i} \frac{\partial f(x_m)}{\partial \Theta_j} \right)}_{=:T} \end{aligned}$$

It can be shown, that the expected value of the expression defined as $T$ is 0 [IFSL20]. The expression $\hat{H}_{i,j}$ can be rewritten to the sum of products of the first order gradients of $L_m$ as shown in Equation 5.4:

$$\hat{H}_{i,j} = \sum_{m=1}^{N} \left( \frac{1}{f(x_m)} \frac{\partial}{\partial \Theta_i} f(x_m) \right) \left( \frac{1}{f(x_m)} \left( \frac{\partial f(x_m)}{\partial \Theta_j} \right) \right) \tag{5.4}$$

$$= \sum_{m=1}^{N} \frac{\partial L_m}{\partial \Theta_i} \frac{\partial L_m}{\partial \Theta_j}$$

$$\hat{H} = \sum_{m=1}^{N} (\nabla_\Theta L_m)(\nabla_\Theta L_m)^\top = (\nabla_\Theta L)(\nabla_\Theta L)^\top =: \hat{F} \tag{Fisher-1}$$

in accordance with [IFSL20]. In summary, the expected value of the Hessian of $L$ is $\mathbb{E}[\hat{F}]$. The matrix $F = \mathbb{E}[\hat{F}]$ is called the Fisher information matrix. Please note, that the definition of this matrix is not unique among different authors. The definition $F = \mathbb{E}[\hat{F}]$ agrees with [MG15].

## 5.2 Kronecker-Factored Approximate Curvature

The Fisher matrix is defined using gradients, which are already computed during backpropagation. Therefore, various preconditioning methods utilize the Fisher matrix (or its variants). In the following discussion, a neural network consisting of linear layers and activation layers is considered, following the notation of the example in Section 3.3.1 (which is similar to [MG15]). The Hessian matrix will be a block matrix, where the block $i, j$ corresponds to the linear layers $i$ and $j$ with weight matrices $W^i$ and $W^j$.

In [MG15], K-FAC is described for linear layers of the form $Y = W\tilde{X}$, where each column of $\tilde{X} \in \mathbb{R}^{F \times N}$ corresponds to a sample. Consider the transpose of this format as outlined in Equation 5.5:

$$(W\tilde{X})^\top = \underbrace{\tilde{X}^\top}_{=X} W^\top = XW^\top, \tag{5.5}$$

where each row of $X \in \mathbb{R}^{N \times F} = \tilde{X}^\top$ corresponds to a sample. In the context of GNNs, the latter format is usual [VCC+18, KW17].

In the experimental part of this thesis, the framework PyTorch is used. It uses the definition $Y = XW^\top$ for linear layers. Therefore the derivations are presented for this format. As a transpose is just a reordering of elements, the backpropagation Equation 3.51 is the same, but transposed:

$$\text{for } Y = XW: \qquad \frac{\partial L}{\partial W} = X^\top \frac{\partial L}{\partial Y} \tag{5.6}$$

$$\text{for } Y = XW^\top: \qquad \frac{\partial L}{\partial W} = \left(X^\top \frac{\partial L}{\partial Y}\right)^\top = \left(\frac{\partial L}{\partial Y}\right)^\top X \tag{5.7}$$

The Fisher matrix $F$ in equation (F) is the expected value of the gradient of the logarithm of the probability density $p_{x,y}(\Theta)$ of the model with parameters $\Theta$ multiplied by its transpose. Since the parameters $\Theta$ of the model are the concatenation of the parameters of all layers, the Fisher matrix can be decomposed into blocks. The block $F_{i,j}$ corresponds to the block calculated by taking the gradient with respect to the parameter $W^i$ of the $i$-th layer and $W^j$ of the $j$-th layer, as shown in equation 5.8.

$$F_{i,j} = \mathbb{E}[\operatorname{vec}(\nabla_{W^i} p) \operatorname{vec}(\nabla_{W^j} p)^\top] \tag{5.8}$$

Note, that the vectorization operations need to be added, such that the second derivatives can be calculated (as explained in Section 3.3.5). Assuming the $i$-th layer and the $j$-th layer are linear, the gradients in 5.8 can be replaced with the backpropagation expression

for linear layers from Equation 5.7. In the following, $a^{i-1}$ is the output of the $(i-1)$-th layer and $g_i = \nabla_{a^i} L$ is the gradient of the loss with respect to the output of the $i-$th layer.

$$F_{i,j} = \mathbb{E}\left[\operatorname{vec}(g_i^\top a^{i-1}) \operatorname{vec}(g_j^\top a^{j-1})^\top\right] \tag{5.9}$$

The expression in equation 5.9 can be rewritten using $\operatorname{vec}(uv) = v^\top \otimes u$ as well as $(A \otimes B)(C \otimes D) = AC \otimes BD$ [MG15]:

$$\begin{aligned}
F_{i,j} &= \mathbb{E}\left[((a^{i-1})^\top \otimes g_i^\top)((a^{j-1})^\top \otimes g_j^\top)^\top\right] && (5.10)\\
&= \mathbb{E}\left[((a^{i-1})^\top \otimes g_i^\top)((a^{j-1}) \otimes g_j)\right] && (5.11)\\
&= \mathbb{E}[\underbrace{((a^{i-1})^\top a^{j-1})}_{=:A_{i-1,j-1}} \otimes \underbrace{(g_i^\top g_j)}_{=:G_{i,j}}] && (5.12)\\
&\approx \mathbb{E}[A_{i-1,j-1}] \otimes \mathbb{E}[G_{i,j}] && \text{(K-FAC)}
\end{aligned}$$

The matrices $A_{i-1,j-1}$ and $G_{i,j}$ can be interpreted as covariance matrices. They are always positive definite and symmetric. In the last step of the derivative, it is assumed that $A_{i-1,j-1}$ and $G_{i,j}$ are independent. According to [MG15], this approximation seems in practice fairly accurate.

One last step is needed in order to apply this approximation to the gradient in an optimization process. As the inverse of this approximation needs to be multiplied with the gradient, any gradient needs to be vectorized first, multiplied with the inverse matrix and then be unvectorized.

Assuming the diagonal blocks of the Fisher matrix are calculated using $A_{i-1,i-1}$ and $G_{i,i}$. The overall approximation for $H$ is $P = \operatorname{diag}((A_{0,0} \otimes G_{1,1})^{-1}, \ldots, (A_{l-1,l-1} \otimes G_{l,l})^{-1})$ where $l$ is the count of linear layers of the neural network. If $V_i$ is the vectorized gradient corresponding to the parameter of the $i-$th layer, then the preconditioned gradient can be calculated using

$$(A_{i-1,i-1} \otimes G_{i,i})^{-1} \operatorname{vec}(V_i) = \operatorname{vec}(G_{i,i}^{-1} V_i A_{i-1,i-1}^{-1})$$

where the rule $(A \otimes B) \operatorname{vec}(X) = \operatorname{vec}(BXA^\top)$ was used [MG15].

## 5.3 Applying K-FAC to GNNs

As already mentioned in Section 3.1, Graph Convolutional networks can be decomposed into message passing layers and linear layers. Therefore, K-FAC can be applied directly to the linear layers. The package `pytorch-geometric` implements a GCN layer by first applying the linear layer, then message passing and lastly adding a bias. To maintain this order, a bias layer needed to be created. The bias layer is defined by a parameter

$b \in \mathbb{R}^{1\times F}$, which will be added to the input sample $x \in \mathbb{R}^{1\times F}$:

$$y_i = x_i + b_i \qquad\qquad Y = X + \begin{pmatrix} b \\ \vdots \\ b \end{pmatrix} \tag{5.13}$$

If the input is augmented with a one, the bias layer can be considered as a linear layer where the weight matrix $W$ is composed of the identity matrix and the bias vector $b$, as described in Equation 5.14:

$$y = (x_1, \ldots, x_F, 1) \underbrace{\begin{pmatrix} I_F \\ b \end{pmatrix}}_{=:W} \tag{5.14}$$

Therefore the backpropagation equations for linear layers can be used for the bias layer:

$$\frac{\partial L}{\partial W}_{i,j} = (X^\top)_{i,m} \left(\frac{\partial L}{\partial Y}\right)_{m,j} \tag{5.15}$$

As the bias layer only contains $b = W_{(F+1),:}$ as parameter, Equation 5.15 can be simplified, such that the resulting expression has the same form as the backpropagation equation for a linear layer as shown in Equation 5.16:

$$\left(\frac{\partial L}{\partial W}\right)_{(F+1),j} = (X^\top)_{F+1,m} \left(\frac{\partial L}{\partial Y}\right)_{m,j} = \underbrace{\begin{pmatrix} 1 & \ldots & 1 \end{pmatrix}}_{=:\tilde{X}\in\mathbb{R}^{1\times N}} \left(\frac{\partial L}{\partial Y}\right)_{:,j} \rightarrow \frac{\partial L}{\partial b} = \tilde{X}\frac{\partial L}{\partial Y} \tag{5.16}$$

In summary, the backpropagation equation for a bias layer is of the same form as for a linear layer (Equation 3.51). However, as the matrix $\tilde{X} \in \mathbb{R}^{1\times N}$ only consists of ones, the resulting sample covariance matrix $A_{i-1,i-1}$ is the identity matrix $I_1$. Therefore only the matrix $G_{i,i}$ is needed for preconditioning a bias layer.

### 5.3.1 Specifics of Node Classification

GNNs can be trained for node classification or for graph classification. In graph classification tasks, the whole graphs are used to calculate the loss. In the case of node classification, the entire graph is passed through the neural network, but only a subset of the nodes, referred to as the training nodes, is used to calculate the loss. The set of training nodes can be determined in various ways. One approach is to take a random sample, but it is also possible consider the distribution of the labels when sampling.

If the entire graph data is used to calculate the covariance matrices $A_{i-1,i-1}$ and $G_{i,i}$, the optimization process might be misguided by the data of the non-training nodes. Consequently, the optimization process could converge more slowly than it would without K-FAC. Therefore, it makes sense, to only use the data from the training nodes to

calculate the covariance matrices. In other words, the non-training nodes are filtered before calculating the covariance matrices.

To verify if filtering the non-training nodes before calculating the covariance matrices $A_{i,i}$ and $G_{j,j}$ in a node classification task has a significant effect on the model, an experiment was conducted. A two-layer Graph Convolutional Network was trained for 200 epochs on the Cora dataset using the ADAM optimizer and K-FAC with different variants. The count of hidden channels was set to 64, the learning rate to 0.01 and the K-FAC damping to 0.1.

Before training began, the filter for the data used to calculate the covariance matrices was set to include the training nodes plus an additional set of nodes sampled from the training graph. The number of nodes $k$ added was varied to observe the effect on the best test accuracy and the best loss. For each $k$, the training was run 20 times and averages have been calculated.

The results are shown in table 5.1. The Cora dataset has 2708 nodes of which 140 are used for training. The last row, with $k = 2400$, uses nearly all the data to calculate the covariance matrices. It can be seen that the loss is lowest when using $k = 2400$. The loss is higher when $k$ is lower, and this is also true for the standard deviation. However, the test accuracy remains nearly the same across all variants of $k$, with the largest difference between the highest and lowest test accuracy being 0.004.

In summary, while the loss is lower when using more nodes to calculate the covariance matrices, the test accuracy remains about the same. For the sake of calculation speed, it is preferable to calculate the covariance matrices with a smaller sample size. The complexity of calculating the sample covariance matrix of a sample matrix $A \in \mathbb{R}^{N \times F}$ is $NF^2$, showing a linear dependence on the count of samples $N$. Experiments on the PubMed and the CiteSeer dataset showed similar results regarding test accuracy.

| **Added nodes** $k$ | **Test Accuracy** [%] | **Loss** |
|---|---|---|
| 0 | 0.824± 0.006 | 0.508± 0.210 |
| 400 | 0.820± 0.005 | 0.568± 0.253 |
| 800 | 0.820± 0.004 | 0.422± 0.242 |
| 1600 | 0.822± 0.006 | 0.396± 0.224 |
| 2000 | 0.822± 0.005 | 0.424± 0.216 |
| 2400 | 0.822± 0.007 | 0.324± 0.167 |

Table 5.1: Results from the experiment investigating the effect of node filtering for the data to calculate the covariance matrices.

### 5.3.2 Graph Isomorphism Networks (GINs)

In [XHLJ19], the expressivity of Graph Neural Networks is investigated. GCNs may not have the same expressivity as the Weisfeiler-Lehman (WL) graph isomorphism test. Graph Isomorphism Networks, on the other hand, are as expressive as the WL test [XHLJ19].

A layer of a GIN network can be represented with the following equation (following [XHLJ19]):

$$Y = \mathrm{MLP}^{(k)}\left((A + (1 + \varepsilon^{(k)})I)X\right)$$

where $\varepsilon^{(k)} \in \mathbb{R}$ can be a fixed number or a learnable parameter, $X$ is the input matrix with each row representing the feature vector of a node, and $A$ is the adjacency matrix of the graph. The function $\mathrm{MLP}^{(k)}$ denotes an arbitrary Multi Layer Perceptron. Note that this architecture is equivalent to a GCN if $\varepsilon = -1$, the normalized adjacency matrix $\tilde{A}$ is used and a one layer MLP is chosen. In some papers the GIN architecture with $\varepsilon = 0$ is referred to as GIN-0. Experiments from [XHLJ19] suggest that setting $\varepsilon = 0$ is preferable.

Since a GIN utilizes an ordinary MLP, K-FAC preconditioning can be applied directly. GINs are typically used for graph classification tasks, where filtering the training data for the calculation of the covariance matrices is not necessary. In graph classification tasks, the features of all nodes are aggregated to make a prediction for the entire graph, often in combination with a MLP. Commonly used aggregation functions include the sum and the mean.

### 5.3.3 Graph Attention Networks

Another common variant of GNNs is Graph Attention Networks (GATs), which incorporate an attention mechanism. Similar to GCNs, first a linear transformation is applied to each node [VCC+18]:

$$\tilde{X} = XW^\top \in \mathbb{R}^{N \times F'} \tag{5.17}$$

Then the message passing step and an activation function is applied, but with attention coefficients $\alpha_{i,j}$:

$$X_{i:} = \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} \tilde{X}_{j:}\right)$$

The attention coefficients $\alpha_{ij}$ are calculated with a linear layer, followed by a LeakyReLU function denoted by $R$ and a softmax layer

$$\alpha_{ij} = \frac{\exp(R(a^\top(\tilde{X}_{i:} || \tilde{X}_{j:}))}{\sum_{k \in \mathcal{N}_i} \exp(R(a^\top(\tilde{X}_{i:} || \tilde{X}_{k:})))}$$

where $||$ is the concatenation operation and $a \in \mathbb{R}^{2F'}$. The authors of [VCC+18] found, that using multiple attention heads is beneficial. The output from multiple heads is concatenated, such that the count of output features is $H \cdot F'$ if there are $H$ attention heads. With regard to K-FAC, this means that preconditioning can be applied to all parameters of GAT layers: $W$ and $a$.

Please note, that GATs can be used for graph classification and node classification tasks. GATs have a higher number of parameters than GCNs, therefore have a tendency to be more difficult to optimize. Thus, applying K-FAC to GATs can be a good benchmark.

## 5.4 Hessian-free Preconditioning

Another technique for preconditioning is discussed in [Mar10]. This approach leverages the fact, that Hessian-vector products $Hv$ can be computed much easier than the full Hessian matrix directly [Pea94]. This allows the application of the conjugate gradient algorithm to efficiently calculate $c = H^{-1}v$.

To explain the technique for calculating $Hv$, it is first necessary to clarify the calculation of a Jacobian-vector product, as this is essential for implementing the approach from [Mar10]. The chain rule is used to compute the derivatives of a function of the form $f = f_l \circ f_{l-1} \circ \cdots \circ f_1$. The resulting expression in Equation 5.19 can be evaluated from left to right or from right to left:

$$f_i(x_{i-1}) = x_i \qquad x_0 = x \tag{5.18}$$

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\mathrm{d}x_l}{\mathrm{d}x_{l-1}} \frac{\mathrm{d}x_{l-1}}{\mathrm{d}x_{l-2}} \cdots \frac{\mathrm{d}x_1}{\mathrm{d}x_0} \tag{5.19}$$

$$= \left( \left( \frac{\mathrm{d}x_l}{\mathrm{d}x_{l-1}} \right) \frac{\mathrm{d}x_{l-1}}{\mathrm{d}x_{l-2}} \right) \cdots \frac{\mathrm{d}x_1}{\mathrm{d}x_0} \tag{RMAD}$$

$$= \frac{\mathrm{d}x_l}{\mathrm{d}x_{l-1}} \left( \frac{\mathrm{d}x_{l-1}}{\mathrm{d}x_{l-2}} \cdots \left( \frac{\mathrm{d}x_1}{\mathrm{d}x_0} \right) \right) \tag{FMAD}$$

The backpropagation algorithm first calculates $x_l, \ldots, x_1$ in a forward pass through the function $f$ as well as the quantities $\frac{\mathrm{d}x_i}{\mathrm{d}x_{i-1}}$. Then the overall derivative $\frac{\mathrm{d}f}{\mathrm{d}x}$ is calculated from left to right, starting by the last layer and ending at the first layer. Due to this ordering, this calculation of the derivative is also called reverse mode autodifferentation (RMAD) [GW08].

The overall derivative can also be evaluated from right to left, starting from the first layer to the last layer. It is not necessary to calculate the quantities $x_l, \ldots, x_1$ first and then do a backward pass. The derivative can be calculated directly in one pass [GW08]. This mode of calculation is also called forward mode autodifferentation (FMAD).

For neural networks trained with a loss function like the root-mean-square-error or a negative log-likelihood, RMAD is usually preferable, because the count of floating point operations is lower [BPRS18]. Furthermore, RMAD calculates products, by repeatedly multiplying the $\frac{\mathrm{d}x_i}{\mathrm{d}x_{i-1}}$ from the left, while in FMAD the quantities $\frac{\mathrm{d}x_i}{\mathrm{d}x_{i-1}}$ are repeatedly multiplied from the right.

The order of the multiplication is fixed in implementations of RMAD and cannot be changed, therefore it is not possible to calculate a Jacobian vector product with RMAD, which is implemented by a lot of machine learning frameworks. It is only possible to calculate vector Jacobian products. However, a trick described by [Tow17] can solve this issue. The following derivation is taken from [Tow17] with a similar notation. Assuming $f : \mathbb{R}^m \to \mathbb{R}^n$, then the Vector-Jacobian product can be defined as:

$$\mathrm{vjp}(f, v, x) = v^\top J_f(x)$$
$$(\mathrm{vjp}(f, v, x))^\top = J_f(x)^\top v =: g(v)$$

The function $g(v)$ is linear, therefore the Jacobian is $J_f(x)^\top$. Applying the Vector-Jacobian product again to $g$ yields:

$$\text{vjp}(g, u, v) = u^\top J_g(v) = u^\top J_f(x)^\top = u^\top \nabla f = (J_f(x)u)^\top, \tag{5.20}$$

which is the transpose of the Jacobian vector product. With this tool at hand, the Hessian can be calculated as the gradient of the gradient in RMAD frameworks. The Hessian of a real valued function $f : \mathbb{R}^m \to \mathbb{R}$ is:

$$(H_f)_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial}{\partial x_i}(\nabla f)_j \to H_f = (J_g)^\top = \nabla g \qquad \text{where} \qquad g := \nabla f \tag{5.21}$$

Note, that this expression might be written differently depending on the conventions used to define the Jacobian and the gradient. The Hessian vector product can therefore be calculated using the gradient of the gradient and the trick shown in Equation 5.20.

The Hessian-vector product $Hv$ can then be used to compute the inverse Hessian-vector product $H^{-1}v$ using the conjugate gradient algorithm [Meu23]. Note that the conjugate gradient algorithm assumes the matrix $H$ remains constant during the calculations. Since neural networks often include dropout layers that produce different outputs on each forward pass, these layers must be disabled during the application of the conjugate gradient algorithm.

In [Mar10], this method demonstrated improvements over standard stoachstic gradient descent algorithms, with a slight modification. Instead of using the Hessian matrix, the Generalized-Gauss-Newton matrix (GGN) was employed. The GGN is always positive definite and can be derived for an arbitrary loss $l$. First, the gradient is calculated using the chain rule as shown in Equation 5.23:

$$\mathcal{L} = l(y) \qquad y = f(x) \tag{5.22}$$

$$\frac{\partial \mathcal{L}}{\partial x_i} = \frac{\partial l}{\partial y_j}\frac{\partial y_j}{\partial x_i} \tag{5.23}$$

To calculate the second derivative, the product rule is applied to the gradient as shown in Equation 5.24:

$$\frac{\partial}{\partial x_k}\frac{\partial \mathcal{L}}{\partial x_i} = \left(\frac{\partial}{\partial x_k}\frac{\partial l}{\partial y_j}\right)\frac{\partial y_j}{\partial x_i} + \frac{\partial l}{y_j}\left(\frac{\partial}{\partial x_k}\frac{\partial y_j}{\partial x_i}\right) \tag{5.24}$$

$$= \underbrace{\left(\frac{\partial y_l}{\partial x_k}\frac{\partial l}{\partial y_l \partial y_j}\right)\frac{\partial y_j}{\partial x_i}}_{=:G_{ik}} + \frac{\partial l}{y_j}\left(\frac{\partial}{\partial x_k}\frac{\partial y_j}{\partial x_i}\right) \tag{5.25}$$

where in the last step the chain rule was applied in the form of $\frac{\partial}{\partial x_k} = \frac{\partial y_l}{\partial x_k}\frac{\partial}{\partial y_l}$.

The Generalized-Gauss-Newton matrix $G$ refers to the first expression in Equation 5.25:

$$G = (J_f)^\top H_l J_f \tag{5.26}$$

If the loss is the sum of squared errors of the samples $\sum(y_i - x_i)^2$, then the second term in Equation 5.25 will go to zero in the training process.

## 5.5 Summary

An introduction into the Negative Log-Likelihood and K-FAC has been provided. A bias layer was described, including how it can be preconditioned using K-FAC. For node classification tasks with GNNs, a performance enhancement for K-FAC has been described. An according experiment showed, that the test accuracy was not affected by the proposed changes. Additionally, Hessian-free preconditioning including the Generalized Gauss-Newton matrix was discussed.

# 6 Experiments

This section experimentally investigates how preconditioning affects the training and the performance of GNN models. For node classification, the datasets Cora, CiteSeer and PubMed from Planetoid [YCS16] will be used, as well as the arXiv dataset from the Open Graph Benchmark (OGB). Details on these datasets can be found in table 6.1. For graph classification, the datasets from the TUDataset [MKB+20] will be used, which details are provided in table 6.2. All experiments used Cross Entropy as loss, which is equal to applying softmax to the output of the model and applying a negative log-likelihood loss.

The experiments were implemented using the PyTorch framework and the PyTorch-Geometric package. Preconditioning using K-FAC is based on an existing implementation of the K-FAC algorithm called `pytorch-kfac` [1]. This package was chosen due to its well-designed code and adherence to software development principles. The correctness of this implementation was verified using unit tests and compared with a reference implementation from [IFSL20][2]. The implementation was modified[3] such that GCN and GAT convolution layers can be preconditioned (GIN Convolution layers did not require any changes).

Some experiments will use the Hessian or the Generalized Gauss-Newton matrix for preconditioning. The implementation relies on the package `PyTorchHessianFree`[4], which was adapted to support preconditioning[5]. Instead of calculating the Hessian directly, the implementation computes Hessian-vector products. This approach uses the conjugate gradient method to calculate inverse Hessian vector products as explained in Section 5.4.

The experiments were run on an Intel Xeon Gold 6130 CPU with a NVIDIA Tesla P100 GPU with 12GB of memory, which can compute 10.6 TFLOPs in single precision.

| **Dataset** | **Nodes** | **Edges** | **Features** | **Classes** | **Edges/(Nodes$^2$)** |
|---|---|---|---|---|---|
| Cora | 2708 | 10556 | 1433 | 7 | 0.14% |
| CiteSeer | 3327 | 9104 | 3703 | 6 | 0.082% |
| PubMed | 19717 | 88648 | 500 | 3 | 0.022% |
| OGBN-Arxiv | 169343 | 1166243 | 128 | 40 | $4.07 \cdot 10^{-5}$% |

Table 6.1: A statistical overview of the datasets used for the node classification experiments.

[1] `https://github.com/n-gao/pytorch-K-FAC`
[2] `https://github.com/russellizadi/ssp`
[3] `https://github.com/sfetzel/pytorch-kfac/tree/gnns`
[4] `https://github.com/ltatzel/PyTorchHessianFree/tree/main`
[5] `https://github.com/sfetzel/PyTorchHessianFree`

| Dataset | Graphs | Avg. Nodes | Avg. Edges | Features | Classes | $\frac{\text{Edges}}{\text{Nodes}^2}$ |
|---|---|---|---|---|---|---|
| MUTAG | 188 | 17.9 | 39.6 | 7 | 2 | 12,.4% |
| ENZYMES | 600 | 32.6 | 124.3 | 3 | 6 | 11.70% |
| PROTEINS | 1113 | 39,1 | 145,6 | 3 | 2 | 9.52% |
| IMDB-BINARY | 1000 | 19,8 | 193,1 | 0 | 2 | 49.25% |
| REDDIT-BINARY | 2000 | 429,6 | 995,5 | 0 | 2 | 0.54% |
| IMDB-MULTI | 1500 | 13 | 10.1 | 0 | 3 | 5.98% |
| DD | 1178 | 284.3 | 5 | 82 | 2 | $6.18 \cdot 10^{-5}$ % |

Table 6.2: A statistical overview of the datasets used for the graph classification experiments.

In general, the data is split into three distinct sets: the training set, the validation set and the test set. The training set is used to train the model and compute the gradients. The validation set is used to evaluate a metric that indicates how well the model performs. In this thesis, accuracy is used for all experiments (the number of correctly predicted samples divided by the count of samples). During training, the model with the highest validation accuracy is selected. The test set is then used to assess the overall performance of the model by calculating its accuracy, hence also called the model's test accuracy.

## 6.1 Graph Convolutional Networks

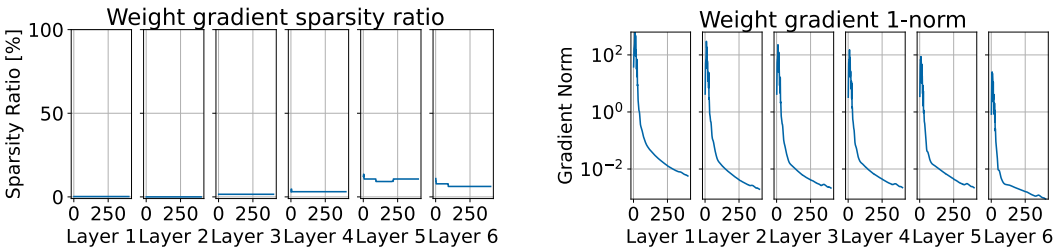



Figure 6.1: A repetition of the same experiment as explained in the problem statement but with K-FAC applied: a 6 layer GCN was trained for 400 epochs on the Cora dataset.

In this section, experiments using Graph Convolutional Networks (GCN) will be presented. To address the first research question – how preconditioning affects the sparsity of the gradients – the experiment on gradient sparsity is repeated with K-FAC preconditioning applied. The damping rate of K-FAC was set to 0.1, all other hyperparameters were left unchanged. Figure 6.1 shows the sparsity of the weight gradients for each layer, using both the sparsity ratio (the ratio of zero elements to the number of elements) and the 1-norm.

The maximum sparsity ratio is about 15%, while the minimum is 0%. In comparison to Figure 4.1, which shows the same experiment without preconditioning, the sparsity has improved significantly with K-FAC preconditioning. Without preconditioning the maximum sparsity ratio was approximately 30% and the minimum was about 3%. Regarding the 1-norm, the difference between the plots in Figures 4.1 and 6.1 is low. In some layers, the norm increased with preconditioning; for instance, the lowest 1-norm was about $10^{-3}$ without preconditioning for the first layer while with preconditioning, it increased to approximately $10^{-2}$. Overall, the discussed experiment from Figure 6.1 demonstrates that preconditioning is an effective approach for addressing sparse weight gradients.

Next, the training of a two-layer GCN with and without K-FAC is investigated. In [KW17], experiments on the Planetoid dataset used 16 hidden features, a dropout rate of 0.5, a weight decay (L2 regularization) of $5 \cdot 10^{-4}$ and the ADAM optimizer with a learning rate of 0.01. The same parameters are used for the following experiment. The K-FAC damping factor was set to 0.1, and the covariance matrix decay was set to 0.25. The network consists of two layers. Figure 6.2 shows the average loss over 100 runs for training with the ADAM optimizer, both with and without K-FAC preconditioning, over 200 epochs.

It is evident that the loss decreases more rapidly with K-FAC preconditioning and the final loss is also lower. However, the overall accuracy does not improve with preconditioning. While the final training accuracies with and without preconditioning are comparable, the average best test accuracy is 0.5-1% higher without preconditioning for the Cora, PubMed and CiteSeer datasets (details in table 6.6). These results indicate overfitting. In contrast, for the ogbn-arxiv dataset, the validation accuracy is higher with preconditioning. Despite this, the test accuracy remains approximately the same, with a slight increase of about 0.5% when preconditioning is not used. Additionally, training is notably faster in terms of epochs with K-FAC; it achieves its best training accuracy around epoch 200, while ADAM requires approximately 400 epochs to reach its best training accuracy. Although the improved convergence seems desirable, it led to overfitting.

## 6.2 Graph Attention Networks

Compared to GCNs, Graph Attention Networks (GAT) have a higher number of parameters. It is hypothesized that overparameterized models benefit more from preconditoning, so GATs might show better results with preconditioning compared to GCNs. The following experiments were conducted using the same hyperparameters as those optimized for the Cora dataset in [VCC+18]: 8 attention heads with 8 features each (64 features in total), $L_2$-regularization with $\lambda = 0.0005$, dropout probability 0.6 and a learning rate of 0.005. The network architecture follows the one described in [VCC+18] and is detailed in Section 3.3 on transductive learning. For the PubMed dataset, some adjustments were made, including a higher weight decay and a learning rate of 0.01, consistent with the parameters

used in [VCC+18]. Figure 6.3 presents the results of these experiments. As with GCNs, the loss decreases more rapidly with preconditioning. However, both test and validation accuracies do not show improvement compared to training without preconditioning and are slightly lower with K-FAC preconditioning.

The experiment using ogbn-arxiv could not run on the GPU due to insufficient memory. The plot shows the average results from 10 runs. The improvement in training with preconditioning is not as pronounced as it was for ogbn-arxiv when trained on a GCN.

## 6.3 Graph Classification

The TUDataset contains data for graph classification tasks, which will be used in the following analysis. Since GCNs are not suitable for this task, only GIN and GAT architectures will be considered. To ensure a fair comparison, the methodology from [EPBM20] is employed. This approach uses a 10-fold split (N=10) of the dataset and identifies the best-performing model for each fold through a grid search.

For each fold, after the best hyperparameters are determined, the model is trained with these hyperparameters for $R = 3$ runs. The average test accuracy of these three runs is

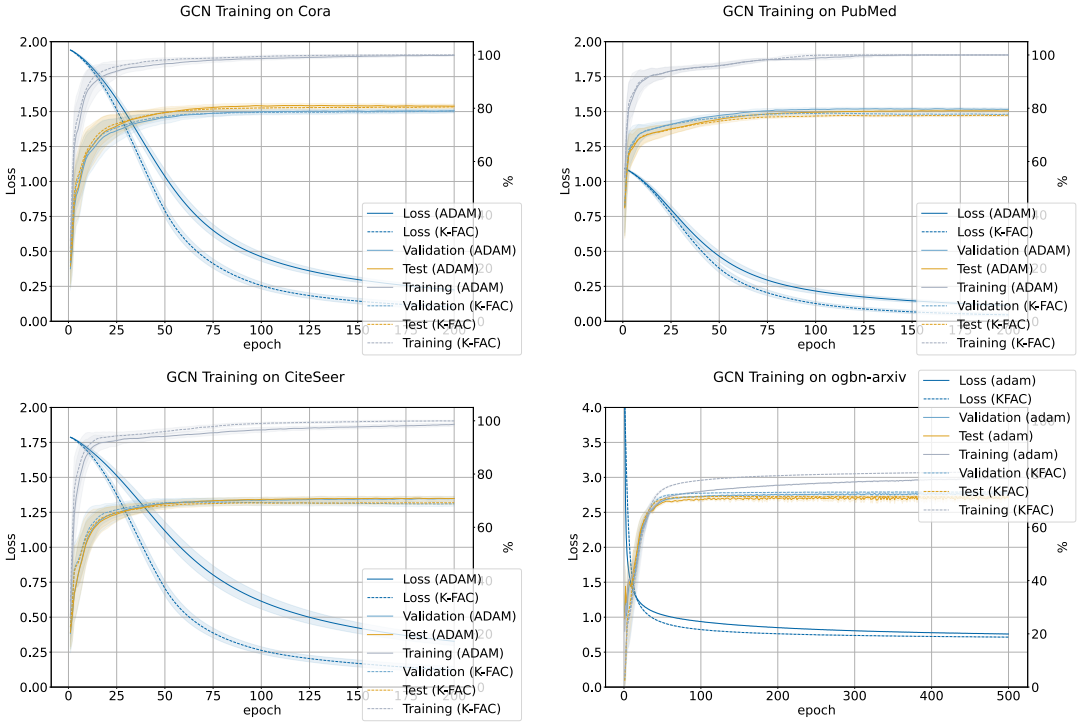



Figure 6.2: Comparison of the training of a GCN on the Planetoid dataset with the same hyperparameters as in [KW17]. Furthermore, the training of ogbn-arxiv is shown. The y-axis on the right hand side shows the accuracy in percent.

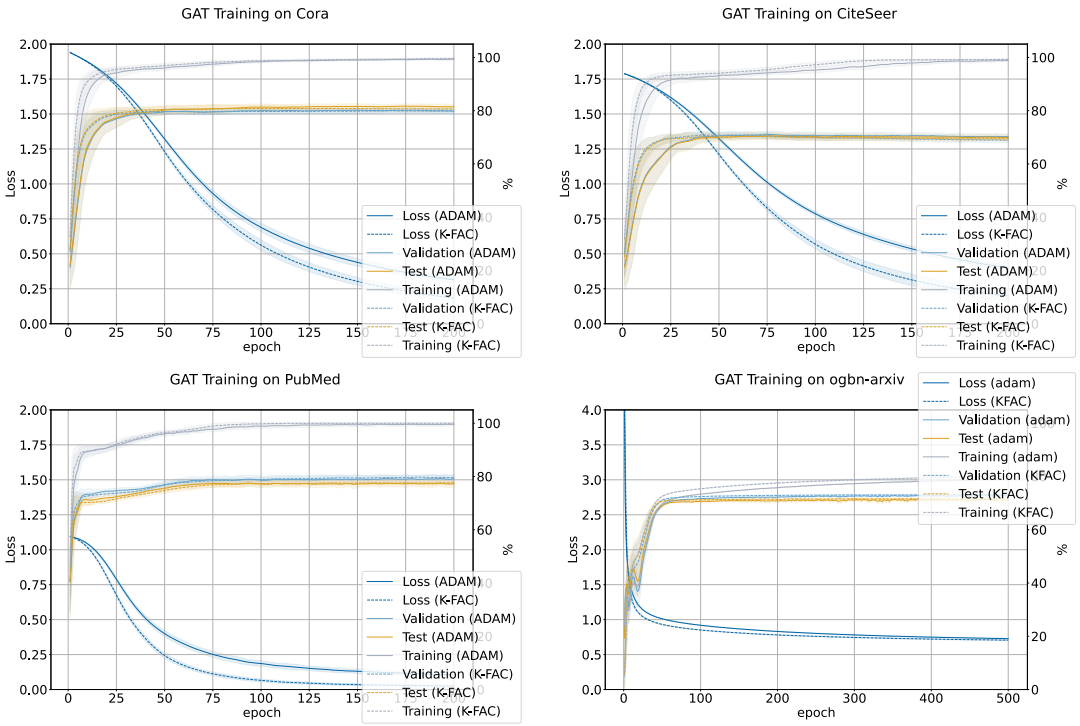



Figure 6.3: Comparison of the training of a GAT on the Planetoid dataset with the same hyperparameters as in [VCC+18]. The y-axis on the right hand side shows the accuracy in percent.

stored for each fold. Finally, the average test accuracy of the folds is calculated. The procedure is detailed in [EPBM20].

A pre-defined split by the authors of [EPBM20] was used. Since MUTAG was not considered by [EPBM20] and there was no split provided, a random stratified split will be used to create the 10-fold split.

To investigate whether K-FAC improves test accuracies in graph classification tasks, one of the hyperparameters considered is the use of K-FAC for preconditioning. When K-FAC is enabled, a damping of 0.1 is used. If K-FAC preconditioning is often selected as the best model during gridsearch for a specific dataset, it might be beneficial to apply it in general for this dataset.

**Configuration** For GIN, the other hyperparameters explored during grid search include the number of hidden channels (32, 64), the number of layers (2,3,5), weight decay (0 or 0.0005) and dropout (0 or 0.5). First, results from simple models are discussed. The simple GIN model consisted of GIN convolution layers with trainable $\varepsilon$-parameter, which used a two layer MLP including batch normalization and dropout. The output was determined by a sum aggregation of the node features followed by another two layer MLP (without dropout and batch normalization). The architecture of the GAT models is the

| Dataset | Model | Loss K-FAC | Loss ADAM |
|---|---|---|---|
| CiteSeer | GAT | 1.29±0.278 | 1.108±0.34 |
| CiteSeer | GCN | 0.549±0.354 | 0.509±0.242 |
| Cora | GAT | 0.956±0.566 | 0.84±0.439 |
| Cora | GCN | 0.215±0.17 | 0.378±0.188 |
| PubMed | GAT | 0.035±0.014 | 0.139±0.039 |
| PubMed | GCN | 0.139±0.055 | 0.186±0.057 |

Table 6.3: An overview of the loss values of the epoch with the maximum validation accuracy.

same as in Section 6.2, but with a sum aggregation.

Additionally, the GIN model from [EPBM20] was used in the experiments[6]. This GIN model implemented jumping knowledge[XLT+18], which allows each layer to contribute directly to the prediction. Adding jumping knowledge connections is expected to increase the expressivity of a model. In other words, the model is able to learn a larger class of functions. Furthermore, the aggregation type (sum or mean) was added as a hyperparameter in these experiments.

**Results (Chemical datasets)** Table 6.4 presents the results for the chemical datasets from the TUDataset. A small improvement of about 2% was observed for the ENZYMES dataset for the simple GIN trained with K-FAC compared to the baseline results for GIN from [EPBM20] (shown in the first row). K-FAC preconditioning was selected as the best option for the simple GIN in 8 out of 10 folds for PROTEINS, MUTAG and ENZYMES dataset, which indicates that the optimization process achieved a higher validation accuracy K-FAC. However, K-FAC preconditioning was not selected as the optimal choice for any fold when the DD dataset was used.

GAT was also trained on the chemical datasets, although [EPBM20] did not consider GATs. The test results were always lower than those from GIN. Preconditioning was used 9 times for the PROTEINS dataset, 10 times for MUTAG, 6 times for ENZYMES and 2 times for the DD dataset.

The GIN model from [EPBM20] trained with K-FAC performed best for the ENZYMES dataset with a gain of about 7%. For all other datasets, the results are lower than the baseline from [EPBM20] (seen in the first row of Table 6.4). Preconditioning was chosen 8 times for PROTEINS, 10 times for ENZYMES and 5 times for MUTAG and DD.

**Results (Social datasets)** The results for the social datasets from the TUDataset collection are shown in table 6.5. Improvements were observed in all datasets except for REDDIT-BINARY for the simple GIN model compared to the baseline GIN results from [EPBM20]. K-FAC preconditioning was selected 7 times for IMDB-MULTI, 5 times for IMDB-BINARY, 9 times for the COLLAB dataset and 8 times for REDDIT-BINARY.

[6]Taken from `https://github.com/diningphil/gnn-comparison/tree/master`

The improvement for the COLLAB dataset, at about 4%, is notably significant. Due to the poor results on chemical datasets and the large amount of time for these experiments, GAT was not trained on the social datasets.

The results for the GIN model from [EPBM20] trained with K-FAC improved over the baseline GIN. However, the test accuracy is lower compared to the simple GIN for all datasets but for IMDB-MULTI. K-FAC preconditioning was chosen 2 times for IMDB-BINARY, 5 times for IMDB-MULTI and 10 times for the COLLAB dataset. Due to insufficient time, this model was not trained on the REDDIT-BINARY dataset.

In conclusion, K-FAC has shown the potential to enhance the results in graph classification in some cases. While there is a noticeable tendency for K-FAC to improve results for the simple GIN model in chemical datasets, this effect is less pronounced for social datasets. For some datasets like PROTEINS, ENZYMES and COLLAB, K-FAC was chosen in the best configuration for the majority of the folds, indicating that preconditioning improved the validation accuracy. However, the higher validation accuracy did not always result in a higher test accuracy.

To date, there are no other known publications on applying K-FAC to graph classification tasks with GNNs to which these results could be compared. It could be observed, that the test accuracies in Tables 6.4 and 6.5 are sometimes lower than the baseline results from [EPBM20], indicating that the validation accuracies are higher with K-FAC, but the resulting test accuracies may be lower due to overfitting.

| | **PROTEINS** | **ENZYMES** | **MUTAG** | **DD** |
|---|---|---|---|---|
| GIN (baseline) [EPBM20] | 73.3± 4.0 | 59.6± 4.5 | 93.43± 5.99* | **75.3 ± 2.9** |
| GIN (simple) with K-FAC | **73.77±2.74** (8) | 61.94±5.20 (8) | **93.60±4.44** (8) | 75.10±2.58 (0) |
| GAT with K-FAC | 66.88 ± 3.27 (9) | 22.33 ± 1.84 (6) | 85.82 ± 4.47 (10) | 70.45 ± 4.08 (2) |
| GIN from [EPBM20] with K-FAC | 71.13±5.00 (8) | **66.78±2.72 (10)** | 92.73±4.74 (5) | 74.73±3.15 (7) |

Table 6.4: Results of the fair comparison algorithm with K-FAC preconditioning on the chemical datasets contained in the TUDataset collection. Numbers in parenthesis indicate how often K-FAC was chosen in the best configuration for a fold. The result from MUTAG marked with * was not contained in [EPBM20].

## 6.4 Prevent overfitting

Previous experiments on node classification tasks in Sections 6.2 and 6.1 with the Planetoid dataset indicated that training with K-FAC tends to lead to overfitting, resulting in lower test and validation accuracies. However, the graph classification experiments described in

| | **IMDB-BINARY** | **IMDB-MULTI** | **REDDIT-BINARY** | **COLLAB** |
|---|---|---|---|---|
| GIN (baseline) [EPBM20] | 71.2 ± 3.9 | 48.5 ± 3.3 | **89.9±1.9** | 75.9 ± 1.9 |
| GIN (simple) with K-FAC | **72.6±3.65** (5) | 50.36±3.87 (7) | 81.76±2.22 (8) | **79.66±2.28** (9) |
| GIN from [EPBM20] with K-FAC | 72.67±3.32 (2) | **51.09±4.37 (5)** | - | 76.33±2.12 (10) |

Table 6.5: Results of the fair comparison algorithm with K-FAC preconditioning on the social datasets contained in the TUDataset collection. Numbers in parenthesis indicate how often K-FAC was chosen in the best configuration for a fold.

Section 6.3 showed improvements in test accuracy for some datasets and models, but also lower test accuracies for some other cases. This indicates that overfitting may also be a problem in graph classification tasks.

There are several techniques to prevent overfitting, with dropout and weight decay being two commonly used methods.

Dropout randomly sets a subset of the input features to zero during training, making the model more robust against disturbances in the input. In other words, a predefined percentage of the entries in the input matrix $X$ will be set to zero.

Weight decay, on the other hand, adds a penalty proportional to the norm of the model parameters to the objective function. This encourages the optimizer to prefer models with smaller parameter norms, which are considered more lightweight and therefore leading to simple, more generalized models.

A grid search was conducted to find the best configurations for dropout and weight decay for the Planetoid dataset. The count of hidden channels was set to 64 for this experiment. The dropout values tested were 0.4, 0.5 and 0.6, while the weight decay values included 0.05, 0.005 and 0.0005. All models trained had two layers and each configuration was run for 5 times. The results are summarized in table 6.6.

Some configurations showed improvements over the standard settings, though the gains were modest. For instance, the mean test accuracy of a GCN trained on Cora with K-FAC was 80,49% as shown in Figure 6.2. The best configuration from table 6.6 for a GCN trained on Cora with K-FAC achieved a test accuracy of 82,0%. For GAT, the accuracy improved from 81,21% to 81,6%, but this is still lower than the mean test accuracy of 82,31% achieved using only the ADAM optimizer.

The results for the other datasets show more noticeable improvements. For example, the mean test accuracy for a GCN trained on PubMed with ADAM is 78,83%. The best configuration from table 6.6 achieved a mean test accuracy of 79,24%, representing a small improvement. The gains are more significant for a GCN trained on the CiteSeer

dataset, where the accuracy improved from 69,06% to 72,18% with the grid search.

Overfitting was also a problem in some cases in the graph classification experiments in Section 6.3. These experiments are time-consuming, therefore it was not feasible to run another grid search on all datasets. As an illustration, the gridsearch was extended to include a weight decay of 0.005 for the PROTEINS dataset for the GIN model from [EPBM20]. K-FAC as preconditioner was fixed in this experiment. The resulting mean test accuracy of $72.39 \pm 2.66$ is slightly higher than the previous result of $71.13 \pm 5.0$. The weight decay of 0.005 was chosen for the best model in all 10 folds, suggesting that the higher weight decay could prevent overfitting to some extent.

In conclusion, while K-FAC consistently finds a model with lower loss, the test and validation accuracies may still be lower compared to models trained with the ADAM optimizer. Fine-tuned weight decay and dropout can lead to slightly better results than those achieved with ADAM alone. The lower loss found by K-FAC is consistent with previous results from [MG15] on other datasets. The results reported in [IFSL20] on the Planetoid dataset found by a grid search on the K-FAC damping parameter (captioned as "Adam-KFAC$_\varepsilon$") differ slightly (e.g. $81.68 \pm 0.25$ for Cora on a GCN, $79.48 \pm 0.28$ for PubMed) from the grid search results in table 6.6, which might due to the different hyperparameters and implementation related differences.

| **Dataset** | **Model** | **Dropout** | **Weight decay** | **Accuracy gridsearch** | **Accuracy K-FAC** | **Accuracy ADAM** |
|---|---|---|---|---|---|---|
| CiteSeer | GAT | 0.5 | 0.05 | 71.62 ± 0.522 | 70.853±0.788 | 71.662±0.9 |
| CiteSeer | GCN | 0.6 | 0.005 | 72.18 ± 0.205 | 69.059±0.897 | 70.859±0.869 |
| Cora | GAT | 0.5 | 0.05 | 81.6 ± 0.367 | 81.205±1.14 | 82.312±0.782 |
| Cora | GCN | 0.6 | 0.005 | 82.0 ± 0.255 | 80.488±0.843 | 81.087±0.836 |
| PubMed | GAT | 0.6 | 0.005 | 78.06 ± 0.627 | 77.698±0.56 | 77.941±0.608 |
| PubMed | GCN | 0.4 | 0.05 | 79.24 ± 0.702 | 77.122±0.568 | 78.825±0.669 |

Table 6.6: Test accuracy results from a grid search on the dropout and the weight decay for each model when trained with K-FAC and ADAM. The column "Accuracy gridsearch" shows the results when the respective dropout and weight decay from the table were used. The column "Accuracy K-FAC" shows the baseline results when a standard weight decay of 0.0005 and a dropout of 0.5 were used, trained with K-FAC and ADAM. The last column shows the test accuracy when trained only with ADAM.

## 6.5 K-FAC and Hessian-free optimization

To gain better insight into how well K-FAC approximates the second-order information, the algorithm can be compared to existing approaches like Hessian-free optimization, as explained in Section 5.4. Hessian-free methods do not calculate the Hessian directly; instead, they compute Hessian-vector products exactly and use the conjugate gradient

method to approximate the inverse matrix-vector product.

Since the Hessian is not always positive definite, damping can be used as a compensatory measure. It also makes the optimization process more reliable [MG15]. Specifically, $\beta$ times the identity matrix is added to the Hessian before executing the conjugate gradient algorithm, with $\beta = 1.0$. The Generalized-Gauss-Newton matrix can also be employed for preconditioning, as it is always positive definite. Therefore, a lower damping can be applied when using the GGN matrix. In the following experiments, a damping factor of 0.1 was used for K-FAC and the GGN matrix.

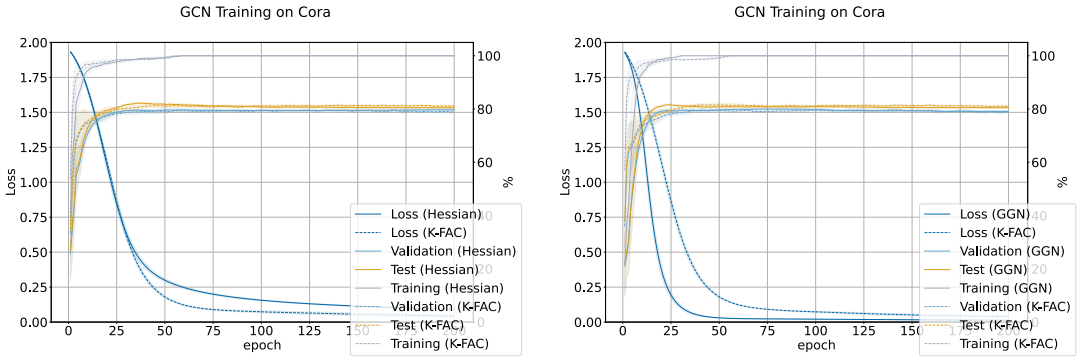



Figure 6.4: Comparison of the training of a GCN with preconditioning with the Hessian (left plot) and the Generalized-Gauss-Newton matrix (right plot).

Preconditioning with the Hessian matrix and the Generalized Gauss-Newton (GGN) matrix was compared to preconditioning with K-FAC in an experiment. The covariance matrix update frequency was set to $f = 1$, so it was updated in every epoch. The maximum count of iterations for the conjugate gradient algorithm was set to 1000, since this value resulted in the lowest loss values. The ADAM optimizer was used to apply the gradients to the parameters. The left plot in Figure 6.4 presents a comparison between the Hessian and K-FAC showing the average of 10 runs each. Both methods perform nearly the same initially, but K-FAC begins to outperform preconditioning with the Hessian around epoch 27. This might be due to the large damping factor used with the Hessian matrix. The `PyTorchHessianFree` library supports adaptive damping, which reduces the damping during training. However, adaptive damping was disabled due to errors regarding the definiteness and unreliable convergence. This could also explain why the experiment using the Hessian did not achieve a lower loss.

The right plot in Figure 6.4 compares preconditioning with the GGN matrix to K-FAC. The variant using the GGN matrix clearly outperforms K-FAC, reaching 100% training accuracy approximately 20 epochs earlier. Although the loss converges faster for GGN, the test accuracy improves quicker for K-FAC in the first 20 epochs. Additionally, the difference between the mean test accuracies between K-FAC and the Hessian-free methods ranged from 0.2% to 2.2% (see Table 6.7). However, no preconditioner did consistently outperform the others. Interestingly, the Hessian free preconditioners tend to achieve the highest validation accuracy in an earlier stage of the training than K-FAC,

as the corresponding loss in Table 6.7 is larger than the one from K-FAC in nearly all experiments. The mean running times using Hessian-free methods were significantly larger than the K-FAC runtimes (for example, the training time was approximately 3 times the K-FAC training time for a GCN on Cora). The overhead could be reduced by lowering the maximum count of conjugate gradient iterations. However these experiments focused on comparing K-FAC with preconditioning methods that approximate the true Hessian and the GGN matrix very well.

| **Dataset** | **Model** | **Type** | **Loss** | **Test Accuracy [%]** | **K-FAC Test Accuracy [%]** | **K-FAC Loss** |
|---|---|---|---|---|---|---|
| CiteSeer | GAT | GGN | 0.807±0.201 | 69.84 ± 0.443 | 69.67±0.868 | 0.661±0.526 |
| CiteSeer | GAT | Hessian | 1.136±0.184 | 70.26 ± 0.636 | 70.48±1.241 | 0.759±0.386 |
| CiteSeer | GCN | GGN | 0.467±0.141 | 68.18 ± 0.797 | 69.72±0.561 | 0.822±0.356 |
| CiteSeer | GCN | Hessian | 0.391±0.207 | 70.57 ± 0.677 | 69.71±0.446 | 0.742±0.276 |
| Cora | GAT | GGN | 0.9±0.536 | 79.39 ± 1.069 | 81.58±0.676 | 0.217±0.331 |
| Cora | GCN | GGN | 0.059±0.065 | 80.94 ± 0.735 | 81.6±0.604 | 0.185±0.132 |
| Cora | GCN | Hessian | 0.451±0.201 | 81.82 ± 0.656 | 81.28±0.388 | 0.248±0.237 |
| PubMed | GAT | GGN | 0.023±0.036 | 76.3 ± 1.404 | 77.8±0.638 | 0.006±0.002 |
| PubMed | GAT | Hessian | 0.087±0.068 | 77.58 ± 0.822 | 78.08±0.514 | 0.006±0.002 |
| PubMed | GCN | GGN | 0.074±0.027 | 77.18 ± 0.29 | 77.53±0.86 | 0.063±0.042 |
| PubMed | GCN | Hessian | 0.136±0.047 | 79.28 ± 0.789 | 77.07±0.56 | 0.088±0.022 |

Table 6.7: Results from training two-layer GNNs with preconditioning using the GGN and Hessian matrix. The loss values are taken from the epoch, where the model achieved the highest validation accuracy.

## 6.6 Efficiency of K-FAC

The execution times of the previously presented experiments on node classification (Figures 6.3 and 6.2) have been recorded to assess the efficiency of K-FAC. Note, that the covariance matrix update frequency was set to $f = 50$, so the covariance matrices were updated at every 50th epoch. The number of operations per epoch was determined with the Python package `torch-operations-counter`. This package facilitates efficiency calculations by collection all operations dispatched to the C++ part of PyTorch and computing the corresponding operation counts. Note that some operations might not be captured by this package; therefore, the efficiency values are estimates.

Table 6.8 shows the time per epoch (T/E) of the Planetoid dataset experiments discussed in the previous sections. Additionally, the time per epoch with K-FAC preconditioning is divided by the time per epoch without preconditioning to calculate the K-FAC factor, which indicates how much slower K-FAC is.

Table 6.8 shows that GAT models consistently have a higher time per epoch than GCN

models. This is because a GAT model needs two linear layers per GAT convolution layer, whereas a GCN only needs one linear layer per GCN convolution layer. Consequently, the overall time per epoch is always higher for GAT models than for GCN models. Furthermore, the GAT architecture always has a lower K-FAC factor than the GCN architecture, regardless of the dataset. The lower factor of GAT models in table 6.8 suggest, that more complex models, like GAT, have a lower computational time overhead in this implementation.

Additionally, it is notable that the maximum K-FAC factor is 2.004 and the lowest is 1.322. The lowest factor of 1.332 was measured using a GAT model with the PubMed dataset, which is the largest of all Planetoid datasets. This indicates that preconditioning has a lower overhead when using larger datasets. The last column "K-FAC Factor normalized" displays how much more time K-FAC needed to find the best epoch compared to ADAM. For some datasets, the overhead becomes much lower. For example the overhead dropped from 1.804x to 1.094x for CiteSeer. The largest overhead measured with CiteSeer on GAT dropped from 2.004x to 1.069x. The overall maximum normalized factor is 1.868x for Cora on a GCN.

| **Dataset** | **Model** | **ADAM T/E [s]** | **K-FAC T/E [s]** | **K-FAC Factor** | **K-FAC Factor normalized** |
|---|---|---|---|---|---|
| CiteSeer | GAT | 0.013 | 0.023 | 1.804x | 1.094x |
| CiteSeer | GCN | 0.006 | 0.012 | 2.004x | 1.069x |
| Cora | GAT | 0.011 | 0.021 | 1.818x | 1.49x |
| Cora | GCN | 0.005 | 0.01 | 1.864x | 1.868x |
| PubMed | GAT | 0.023 | 0.03 | 1.322x | 1.464x |
| PubMed | GCN | 0.006 | 0.011 | 1.751x | 1.416x |

Table 6.8: Comparison of the time per epoch (T/E) without preconditioning (labeled ADAM) and with preconditioning (labeled K-FAC). The column "K-FAC Factor" indicates how much more time per epoch is needed when preconditioning with K-FAC is applied. The last column "K-FAC Factor normalized" shows how much longer K-FAC took to find the best model compared to ADAM.

Table 6.9 shows the GFLOPs count for training GNNs with preconditioning (labeled K-FAC) and without preconditioning (labeled ADAM). While K-FAC might have a higher overhead, it can converge earlier. Therefore, it is important to incorporate the epoch at which the best validation accuracy was found into the efficiency considerations. For each run, the epoch with the best validation accuracy was recorded. The average of these recorded epochs is referred to as the mean best epoch. The GFLOPs in table 6.9 are calculated as the GFLOPs per iteration multiplied by the mean best epoch. Despite K-FAC's earlier convergence, the total number of operations remains higher with K-FAC preconditioning, as indicated by the higher FLOPs in all experiments.

Furthermore, table 6.9 compares estimates of the efficiency $E$ of training a GNN with and without preconditioning. In this context, efficiency refers to an implementation-specific efficiency, defined as the ratio of the actual execution time $T_a$ to the theoretical

| Dataset | Model | ADAM E [%] | ADAM GFLOPs |
|---|---|---|---|
| CiteSeer | GAT | 1.918 | 185.32 |
| CiteSeer | GCN | 1.171 | 105.051 |
| Cora | GAT | 0.684 | 83.332 |
| Cora | GCN | 0.409 | 32.349 |

| Dataset | Model | K-FAC E [%] | K-FAC GFLOPs | K-FAC GFLOPs Factor |
|---|---|---|---|---|
| CiteSeer | GAT | 2.182 | 230.49 | 1.244x |
| CiteSeer | GCN | 2.159 | 207.008 | 1.971x |
| Cora | GAT | 0.581 | 105.454 | 1.265x |
| Cora | GCN | 0.536 | 79.159 | 2.447x |

Table 6.9: Comparison of the efficiency of training a GNN with preconditioning (labeled K-FAC) and without preconditioning (labeled ADAM), including the count of Giga floating point operations (GFLOPs) per iteration multiplied with the mean best epoch (the epoch where the highest validation accuracy was achieved).

lowest execution time $T_c$ as defined in Equation 6.1:

$$E = \frac{T_a}{T_c} \tag{6.1}$$

The theoretical execution time $T_c$ is calculated as the count of floating point operations (FLOPs) divided by the maximum count of FLOPs per second, as determined by the GPU. Comparing the efficiency values of ADAM to ADAM with K-FAC preconditioning in table 6.9, it is noticeable that the efficiency of K-FAC is slightly higher. This might be due to the fact that the inverse covariance matrices only change at every 50th epoch, reducing the amount of memory transfers.

In summary, the application of K-FAC incurs significant overhead, and the higher count of floating point operations per epoch is not compensated by the earlier convergence of the training.

## 6.7 M-FAC as preconditioner

The authors of [FKA21] provide an implementation of their Matrix-Free Approximate Curvature algorithm. This implementation has been adapted to support preconditioning, allowing it to be combined with the ADAM optimizer[7].

M-FAC only utilizes gradients for its computations, eliminating the need for further adaptions of any code. The number of gradients for the sliding window was set to the default value of 1024.

Figure 6.5 illustrates an experiment using M-FAC preconditioning with two-layer GCN and GAT models, each with 64 hidden channels. All other parameters were kept

[7] `https://github.com/sfetzel/M-FAC`

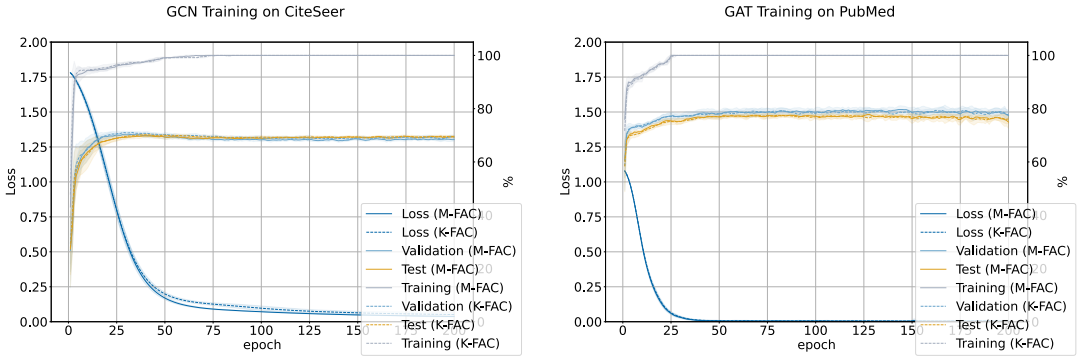



Figure 6.5: The plots show the training with M-FAC preconditioning and with K-FAC preconditioning. The left plot was created using a GCN trained on the CiteSeer dataset, the right plot was created with a GAT and the PubMed dataset.

consistent with those in previous sections. The results displayed represent an average of 10 runs. During these experiments, the covariance matrices of K-FAC were updated in each iteration. Regarding the convergence of the loss and the accuracies in figure 6.5, there is no significant difference between K-FAC and M-FAC. Specifically, the best test accuracy achieved by M-FAC for the PubMed dataset trained on GAT models does not significantly differ from that achieved by K-FAC. For the CiteSeer dataset trained on GCN models, the mean test accuracy is slightly higher with K-FAC. But overall, the differences in convergence and test accuracies between K-FAC and M-FAC are not significant. K-FAC was slower in these experiments, but it is consistently faster when the covariance update frequency is set to $f = 50$.

## 6.8 Ablation study on K-FAC hyperparameters

Although the K-FAC approach does not specify any hyperparameters directly, implementations use several additional parameters.

For example, since the Fisher matrix is defined using an expected value, it is approximated in practice using a moving average. This involves calculating a weighted average between the estimated matrices. In most K-FAC implementations, the estimate for the covariance sample matrix $\hat{A}$ is calculated using an exponential moving average [MG15] [IFSL20]. This method multiplies the previous estimate $\hat{A}_p$ (from the previous training iteration) by a coefficient $\gamma$ and adds the new estimated covariance matrix $\hat{A}_n$ multiplied by $1 - \gamma$, as described in Equation 6.2:

$$\hat{A} = \hat{A}_p\gamma + \hat{A}_n(1 - \gamma) \tag{6.2}$$

The parameter $\gamma$ is referred to as decay factor.

Furthermore, damping is applied to make the optimization process more reliable. Damping is needed in the early iterations of the optimization to compensate for the local

quadratic approximation of the objective, but also to balance out the approximation used for the Fisher matrix [MG15]. The derivation and description of the damping technique used in K-FAC are detailed in [MG15]. The algorithm was refined through trial and error. Effectively, the approximation of the Fisher matrix $\hat{F}$ is damped by adding a multiple of the identity matrix $\beta I$ to it:

$$\hat{F} \to \hat{F} + \beta I$$

In K-FAC, multiples of the identity matrix are added to both the activations covariance matrix $A_{i,i}$ and the sensitivities covariance matrix $G_{j,j}$.

To illustrate the effects of these hyperparameters, experiments with different damping values and decay values are presented in Figure 6.6. The experiments follow an ablation study approach, where only one parameter is varied at a time. A two-layer GCN was trained on the Cora dataset using the ADAM optimizer with K-FAC preconditioning and 64 channels. Each configuration was run for 100 times.

The left plot in Figure 6.6 shows training results with varying damping values and a decay of zero. Lower damping values of 0.01 and 0.001 result in an overall lower loss. However, a damping value of 0.1 achieved the best test and validation accuracy at the 50th epoch.

The right plot in Figure 6.6 shows the training of a two-layer GCN with a damping value of 0.1 and varying decay values. There is little to no difference between the training plots, indicating that the influence of the decay hyperparameter is marginal.

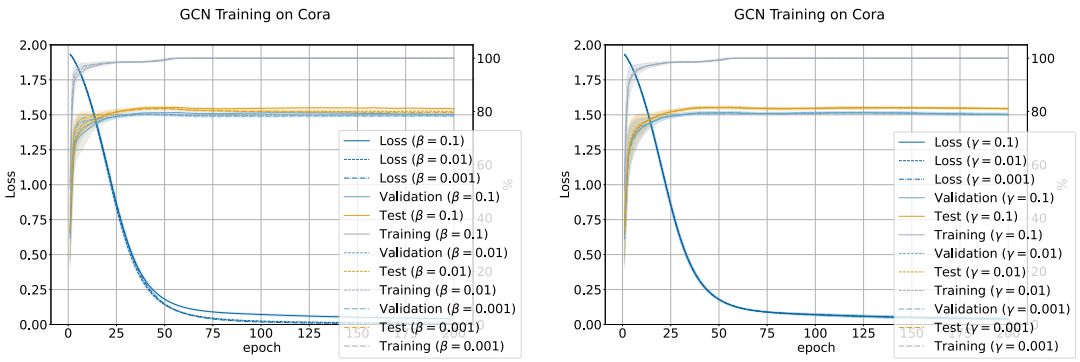



Figure 6.6: The left plot shows the training of a two layer GCN on the Cora dataset with varying damping values. The right plot shows the same GCN trained with varying decay values.

According to [IFSL20], it is possible to reduce the update frequency of the covariance matrices without significantly affecting performance. In their experiments, the covariance matrices were updated every 50th epoch. Figure 6.7 presents the results of experiments conducted on the CiteSeer and Cora datasets using a two layer GCN. The update frequencies were 1 (updating the covariance matrices in each iteration), 25 and 50 (updating every 50th epoch). The convergence and final loss achieved were similar

across these different update frequencies. The best test accuracies for the various update frequencies differ by less than 0.05 for each dataset. Thus, the efficiency of K-FAC can be significantly increased by reducing the frequency of covariance matrix updates.

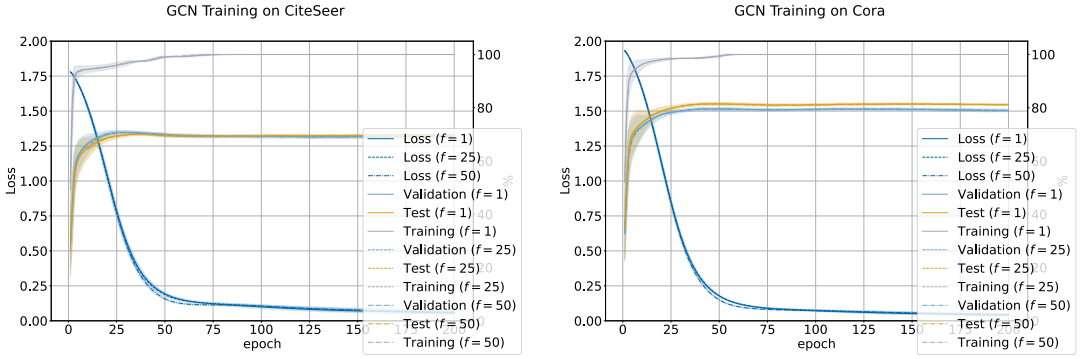



Figure 6.7: Experiments on the covariance matrix update frequency $f$. For $f = 1$, the matrix was updated in every epoch, for $f = 25$ in every 25th epoch.

In summary, a higher damping value was preferred in the experiments to achieve better test accuracy. Since the decay parameter had minimal impact on the results, the values zero and 0.25 were used. The update frequency was set to 50 in the Planetoid and ogbn-arxiv experiments.

# 7 Future Ideas

K-FAC preconditioning could also be applied to various more recent GNN architectures and techniques. For example, the quantization method described in [ZLM+23] uses a separate learning rate for different parameters. Preconditioning has the potential to eliminate the need for these fine-tuned learning rates.

In practice, applying K-FAC to newer architectures beyond GCN, GIN or GAT is often challenging because implementations use non-standard techniques and customized layers. For example, the package `pytorch_geometric` implements a GCN convolution by combining a linear layer, message passing and bias into a single module. It was necessary to modify the original code to apply K-FAC preconditioning.

On the other hand, preconditioning methods that only require the gradients of a parameter, such as [FKA21], are preferable because they require minimal adaptation of existing code.

## 7.1 Ideas for preconditioning algorithms

It seems natural to assume that the gradients encountered during a training process lie in a low-dimensional subspace because the changes from one iteration to the next are small. Therefore applying a dimensionality reduction technique to the gradients, such as orthonormal projections, makes sense.

Each row of $A \in \mathbb{R}^{N \times M}$ corresponds to one vectorized gradient of size $M$ from a training process. The number $N$ is the count of gradients and $M$ is the length of a gradient. Using QR decomposition, the matrix $A^\top$ can be written as the product of $Q \in \mathbb{R}^{M \times N}$ and $R \in \mathbb{R}^{N \times N}$, where the columns of $Q$ form an orthogonal basis (so $Q^\top Q = I$) and the columns of $R$ can be interpreted as coordinates. The decomposition of $A^\top$ is preferable because the QR decomposition orthogonalizes the columns. Therefore, applying the QR decomposition to $A^\top$ yields an orthonormal basis in the vector space of the gradients.

The Fisher matrix is essentially the covariance matrix of the gradients [MG15], which can be estimated using a set of samples. The transpose of the matrix $A$ multiplied with $A$, divided by the count of samples, provides an estimate of the covariance matrix $\tilde{F}$ (the sample covariance matrix). Using $QR$ decomposition of $A^\top$, the estimate $\tilde{F}$ can be written as a product of the matrices $Q$, $R$ and their transposes:

$$\tilde{F} = \frac{1}{N-1} A^\top A = \frac{1}{N-1} QR(QR)^\top = \frac{1}{N-1} Q \underbrace{RR^\top}_{=:S} Q^\top = \frac{1}{N-1} QSQ^\top \tag{7.1}$$

where $S \in \mathbb{R}^{M \times M}$ is the matrix $R$ multiplied by it transpose $R^\top$. The matrix $S$ can be interpreted as the covariance matrix of the coordinates of the gradients with respect to

the orthonormal basis. The expression $QSQ^\top$ is a similarity transform of $S$.

The basic idea of the QR-Inverse is to replace $S$ in Equation 7.1 with its inverse. In summary, an estimation for the inverse covariance matrix is:

$$\tilde{F}^{-1} = (N-1)QS^{-1}Q^\top. \tag{QR-Inverse}$$

Note, that if $N = M$ and $Q^{-1} = Q^T$, the QR-Inverse can be derived as shown in Equation 7.2:

$$\begin{aligned}\left(\frac{1}{N-1}QSQ^\top\right)^{-1} &= (N-1)(QSQ^\top)^{-1} \\ &= (N-1)((Q^\top)^{-1}S^{-1}Q^{-1} \\ &= (N-1)((Q^{-1})^\top S^{-1}Q^{-1} \\ &= (N-1)(QS^{-1}Q^T)\end{aligned} \tag{7.2}$$

Multiplying $\tilde{F}^{-1}$ with a vector $v$ can be interpreted as projecting $v$ into the lower dimensional space ($Q^\top v$), multiplying it with the inverse of $S$ and the projecting it back to the original space (by multiplying with $Q$). During an optimization process, the matrix $Q$ could be built up iteratively to save computation costs, adding the gradients as they are encountered during the training.

A numerical experiment was conducted, where gradient vectors were generated randomly. The samples followed a multivariate normal distribution with mean $\mu$ and random covariance matrix $\Sigma$. A total of $N = 16$ samples have been taken from this distribution with $M = 2048$ parameters. An image of the logarithm of the difference between the pseudoinverse of $\tilde{F}$ and the QR-Inverse of $\tilde{F}$ can be seen in Figure 7.1 on the left. Although the QR-Inverse takes much less time to calculate, the difference between the QR-Inverse and the pseudoinverse is very small. The QR-Inverse needs to orthogonalize $N$ gradients, but only needs to invert a $N \times N$ matrix, instead of a $M \times M$ matrix. Usually, $N >> M$, which leads to a faster computation time. For example, the count of epochs used in Section 6.1 is $N = 200$. The length of the gradient considered in the next experiment is $M = 11464$, which is approximately 60 times larger. Therefore the dominant part of the computation time, the calculation of the inverse, is greatly reduced.

In order to verify whether this approach is suitable for calculating the inverse of covariance matrices in the context of GNN optimization, a GCN with two layers was trained on the Cora dataset. It is assumed that the gradients follow a more complex probability distribution. The first $N = 16$ gradients from the weight matrix of the first layer were vectorized and used to estimate the covariance matrix. The parameter count was $M = 11464$, given that the first GCN layer had 8 output features and there are 1433 input features. The results comparing the inverse covariance matrix from this experiment calculated by the QR-Inverse and the pseudoinverse are shown in Figure 7.1 on the right. While the pseudoinverse took several minutes to calculate, the QR inverse was calculated in less than a second. The difference between the QR inverse and the pseudoinverse is significantly larger than in the first experiment. One source of this large error may be the

invalidity of the assumption $Q^T = Q^{-1}$ and the difference between the QR-decomposition $QR$ and the actual matrix $A^T$. The median of all elements of the difference between the expected and the actual inverse matrix entries is 0.00244. A theoretical investigation of the maximum error of the QR-Inverse needs is necessary to explore methods for reducing this error. In summary, the QR-Inverse could be a viable option for calculating the inverse

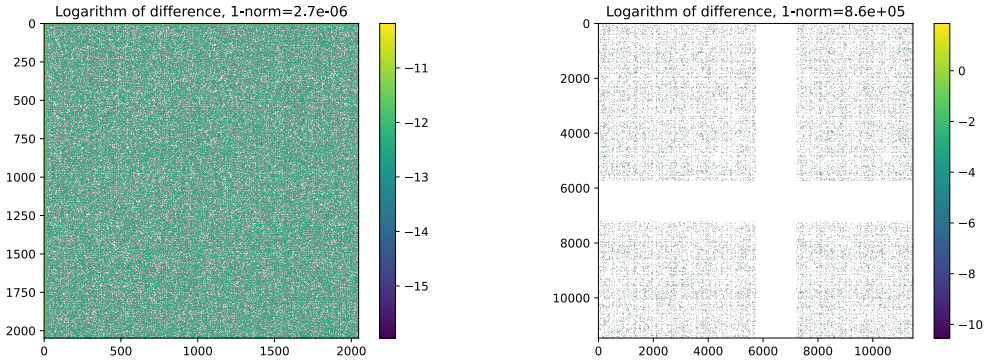



Figure 7.1: Numerical experiments on the QR inverse. The plots show the log10 of the difference between the Pseudoinverse and the QR-Inverse. The left plot is created using a random multivariate normal distribution with $M = 2048$ parameters. The right plot is created using the gradients from the first layer weight matrix of a two layer GCN. The count of parameters is $M = 11464$. A white color indicates a difference of zero.

Fisher matrix with low computational overhead. It might also be used to compute the inverse covariance matrix in scenarios, where the count of samples $N$ is much lower than the count of features $M$. Further experiments are needed to find out if the accuracy is sufficient or is improvable.

Furthermore, one possibility for future preconditioning algorithms is to simplify the analytical expression for the Hessian (Section 3.3.5). For example, this could involve assuming that the second derivative of the loss function is zero or ignoring contributions from other layers. As an illustration, Equation 3.117 could serve as a rough approximation of the Hessian for a single layer.

## 7.2 Prevent overfitting

Another direction for future work is the application of overfitting prevention techniques in combination with K-FAC. While preconditioning accelerates progress from one epoch to the next epoch in terms of the loss, it may also lead to quicker overfitting of the training data compared to first-order methods.

According to the experiments, the best parameters with respect to the validation set are typically found before achieving a very low loss (see Table 6.3 or 6.7). Preventing the

| **Percentage of added edges** [%] | **Test Accuracy** [%] | **Loss** |
|---|---|---|
| 0.0 | 0.820± 0.006 | 0.271± 0.127 |
| 0.1 | 0.817± 0.005 | 0.175± 0.097 |
| 0.5 | 0.819± 0.005 | 0.137± 0.061 |
| 1.0 | 0.815± 0.006 | 0.177± 0.114 |
| 2.0 | 0.810± 0.007 | 0.231± 0.145 |
| 5.0 | 0.796± 0.008 | 0.198± 0.110 |

Table 7.1: An experiment on adding random edges during the training process. The first column is the relative amount of edges added. The test accuracy and loss was taken from the epoch with the best validation accuracy and the best test accuracy of the training process was reported.

optimization algorithm from reaching a too low loss would help to find parameters, which do not overfit the training set.

Inspired by graph editing methods such as [BSAGBG24], a simple overfitting prevention technique was investigated. A two-layer GCN was trained with K-FAC preconditioning on the Cora dataset with a weight decay of 0.0005 for the first layer parameters. During each epoch of the training, a set of random edges was added to the original graph. These additional edges introduce noise into the feature vectors of each node, which should ideally make the neural network more robust to noise and help prevent overfitting by diverting the training from minima, that overly fit the training data. The results of this experiment are shown in table 7.1. Unfortunately, this approach did not improve the neural network's performance. While the loss improved with adding edges, the test accuracy actually declined. Another possibility for this approach could be to select the added edges more carefully.

In summary, overfitting remains a challenge when applying preconditioning techniques and further investigation is needed.

# 8 Conclusion

GNNs such as GCNs, utilize the adjacency matrix to incorporate graph structure through message passing. The gradients in GNNs trained on node classification can be sparse. Preconditioning with K-FAC significantly reduced the sparsity of weight matrix gradients – i.e., the number of zero elements – and improved the convergence of the loss.

Although K-FAC can have positive effects on the training of GNNs, experiments on node classification data from the Planetoid dataset revealed a decrease in mean test and validation accuracies compared to models trained without K-FAC. Although the improved convergence of second-order methods seemed desirable it actually led to overfitting. Adjustments to overfitting prevention techniques, such as weight decay and dropout, could potentially enhance accuracy, but improvements over models trained with the ADAM optimizer were minimal (max. 0.5%).

Additionally, filtering data for covariance matrices of K-FAC in node classification tasks was explored in Section 5.3.1. Experiments on the Planetoid dataset indicated that while filtering has a minimal effect on test accuracy, it results in a lower loss when more data is used. Thus, filtering can be employed to speed up calculations, as the effect on test accuracy is low.

K-FAC preconditioning was also applied to graph classification tasks using data from the TUDataset. Following the approach in [EPBM20], preconditioning with K-FAC was set as a hyperparameter in a grid search. For the PROTEINS, MUTAG and ENZYMES datasets, K-FAC was chosen in the best hyperparameter configuration for a GIN model 8 out of 10 times. The improvements in test accuracy ranged from 0.2% to 7% (ENZYMES). An improvement of about 4% was observed for the COLLAB dataset. These results show that K-FAC preconditioning is beneficial in some experiments. However, it can also be counterproductive, resulting in lower test accuracies due to overfitting.

The efficiency of training with K-FAC preconditioning was assessed. The overhead introduced by K-FAC was not offset by faster convergence. Furthermore, the impact of K-FAC's hyperparameters was studied experimentally. The decay of the covariance matrix did not significantly affect training, while damping had a minor effect on the final loss achieved. Comparisons with Hessian-free methods showed that K-FAC converges similarly to Hessian matrix preconditioning when a damping of one is used. However, preconditioning with the Generalized-Gauss-Newton matrix outperformed K-FAC in terms of training convergence speed.

# Bibliography


[BPRS18] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic Differentiation in Machine Learning: a Survey. *Journal of Machine Learning Research*, 18(153):1–43, 2018.

[BSAGBG24] Maya Bechler-Speicher, Ido Amos, Ran Gilad-Bachrach, and Amir Globerson. Graph Neural Networks Use Graphs When They Shouldn't. *arXiv preprint*, 2024.

[DCJ23] Nikita Doikov, El Mahdi Chayti, and Martin Jaggi. Second-Order Optimization with Lazy Hessians. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

[DHH20] Felix Dangel, Stefan Harmeling, and Philipp Hennig. Modular Block-diagonal Curvature Approximations for Feedforward Architectures. In Silvia Chiappa and Roberto Calandra, editors, *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, volume 108 of *Proceedings of Machine Learning Research*, pages 799–808. PMLR, 26–28 Aug 2020.

[Dhr00] Phoebus J. Dhrymes. *Matrix Vectorization*, pages 117–145. Springer New York, New York, NY, 2000.

[DLZ+22] Shaojun Dong, Fengyu Le, Meng Zhang, Si-Jing Tao, Chao Wang, Yong-Jian Han, and Guo-Ping Guo. Generalization to the Natural Gradient Descent. *arXiv preprint*, 2022.

[DSC22] Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. Activation functions in deep learning: A comprehensive survey and benchmark. *Neurocomputing*, 503:92–108, 2022.

[EPBM20] Federico Errica, Marco Podda, Davide Bacciu, and Alessio Micheli. A Fair Comparison of Graph Neural Networks for Graph Classification. In *International Conference on Learning Representations*, 2020.

[FKA21] Elias Frantar, Eldar Kurtic, and Dan Alistarh. M-FAC: Efficient Matrix-Free Approximations of Second-Order Information. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 14873–14886. Curran Associates, Inc., 2021.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[GL03] Philip E. Gill and Michael W. Leonard. Limited-Memory Reduced-Hessian Methods for Large-Scale Unconstrained Optimization. *SIAM Journal on Optimization*, 14(2):380–401, 2003.

[GLB+18] Thomas George, César Laurent, Xavier Bouthillier, Nicolas Ballas, and Pascal Vincent. Fast approximate natural gradient descent in a kronecker factored eigenbasis. *Advances in Neural Information Processing Systems*, 31, 2018.

[GLZ+21] Yuan Gao, Jianping Li, Yue Zhou, Fei Xiao, and He Liu. Optimization Methods For Large-Scale Machine Learning. In *2021 18th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP)*, pages 304–308, 2021.

[GRB20] Donald Goldfarb, Yi Ren, and Achraf Bahamou. Practical Quasi-Newton Methods for Training Deep Neural Networks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20. Curran Associates Inc., 2020.

[Gro17] Roger Grosse. CSC2541 Lecture 5 Natural Gradient, 2017.

[GW08] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, Second Edition*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, 2008.

[HEAV19] Joao Henriques, Sebastien Ehrhardt, Samuel Albanie, and Andrea Vedaldi. Small Steps and Giant Leaps: Minimal Newton Solvers for Deep Learning. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 4762–4771, 2019.

[Hel15] Tobias Hell. *Einführung in die Stochastik*. Studia Verlag Innsbruck, 2015.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[IFSL20] Mohammad Rasool Izadi, Yihao Fang, Robert Stevenson, and Lizhen Lin. Optimization of Graph Neural Networks with Natural Gradient Descent. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 171–179, 2020.

[KAG+23] Koroko, Abdoulaye, Anciaux-Sedrakian, Ani, Gharbia, Ibtihel Ben, Garès, Valérie, Haddou, Mounir, and Tran, Quang Huy. Efficient approximations

of the Fisher matrix in neural networks using Kronecker product singular value decomposition. *ESAIM: ProcS*, 73:218–237, 2023.

[KB15] Diederik Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

[KL20] Patrick Kidger and Terry Lyons. Universal Approximation with Deep Narrow Networks. In Jacob Abernethy and Shivani Agarwal, editors, *Proceedings of Thirty Third Conference on Learning Theory*, volume 125 of *Proceedings of Machine Learning Research*, pages 2306–2327. PMLR, 2020.

[KL22] Piyush Kaul and Brejesh Lall. Projective Fisher Information for Natural Gradient Descent. *IEEE Transactions on Artificial Intelligence*, PP:1–1, 2022.

[KO20] Ryo Karakida and Kazuki Osawa. Understanding approximate fisher information for fast convergence of natural gradient descent in wide neural networks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20. Curran Associates Inc., 2020.

[Kri18] Agustinus Kristiadi. Natural Gradient Descent, 2018.

[KW17] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations*, 2017.

[Mar10] James Martens. Deep learning via Hessian-free optimization. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 735–742. Omnipress, 2010.

[Mar20] James Martens. New insights and perspectives on the natural gradient method. *J. Mach. Learn. Res.*, 21(1), 2020.

[Meu23] Gérard Meurant. Detection and correction of silent errors in the conjugate gradient algorithm. *Numerical Algorithms*, 92(1):869–891, 2023.

[MG15] James Martens and Roger Grosse. Optimizing Neural Networks with Kronecker-factored Approximate Curvature. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 2408–2417. PMLR, 2015.

[MKB+20] Christopher Morris, Nils M. Kriege, Franka Bause, Kristian Kersting, Petra Mutzel, and Marion Neumann. TUDataset: A collection of benchmark datasets for learning with graphs. In *ICML 2020 Workshop on Graph Representation Learning and Beyond (GRL+ 2020)*, 2020.

[MN19] Jan R. Magnus and Heinz Neudecker. *Matrix Differential Calculus with Applications in Statistics and Econometrics*. Wiley Series in Probability and Statistics. Wiley, 2019.

[Pea94] Barak A. Pearlmutter. Fast exact multiplication by the Hessian. *Neural Comput.*, 6(1):147–160, 1994.

[PP12] Kaare Brandt Petersen and Michael Syskind Pedersen. *The Matrix Cookbook*. Technical University of Denmark, 2012.

[Shu] Jerry Shurman. *Multivariable Calculus*.

[Tow17] James Townsend. A new trick for calculating Jacobian vector products, 2017.

[VCC+18] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *ICLR 2018*, 2018.

[XHLJ19] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How Powerful are Graph Neural Networks? In *International Conference on Learning Representations*, 2019.

[XLT+18] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. Representation Learning on Graphs with Jumping Knowledge Networks. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 5453–5462. PMLR, 10–15 Jul 2018.

[XN09] Dexuan Xie and Qin Ni. An Incomplete Hessian Newton Minimization Method and Its Application in a Chemical Database Problem. *Comput. Optim. Appl.*, 44(3):467–485, 2009.

[YCS16] Zhilin Yang, William W. Cohen, and Ruslan Salakhutdinov. Revisiting semi-supervised learning with graph embeddings. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 40–48. JMLR.org, 2016.

[YGS+21] Zhewei Yao, Amir Gholami, Sheng Shen, Mustafa Mustafa, Kurt Keutzer, and Michael Mahoney. ADAHESSIAN: An Adaptive Second Order Optimizer for Machine Learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(12):10665–10673, 2021.

[ZLM+23] Zeyu Zhu, Fanrong Li, Zitao Mo, Qinghao Hu, Gang Li, Zejian Liu, Xiaoyao Liang, and Jian Cheng. $\rm A^2Q$: Aggregation-Aware Quantization for Graph Neural Networks. In *The Eleventh International Conference on Learning Representations*, 2023.

# 9 Appendix

The code for the experiments on Hessian-free preconditioning and K-FAC is available at `https://github.com/sfetzel/pytorch-kfac/tree/gnns`. The following variant of the package `PyTorchHessianFree` is needed: `https://github.com/sfetzel/PyTorchHessianFree`

## Sparsity Experiments

The code for these experiments is available at `https://github.com/sfetzel/GNN-Sparsity`. The plots were created by training a Graph Convolutional Network with 64 hidden channels using the ADAM optimizer and a learning rate of 0.1 for 400 epochs. The x axes indicate the epochs for each layer. The left hand plot shows the sparsity of the gradients of the loss with respect to the weight matrices of the GCN layers for each epoch. The right hand plot shows the norm of these gradients.

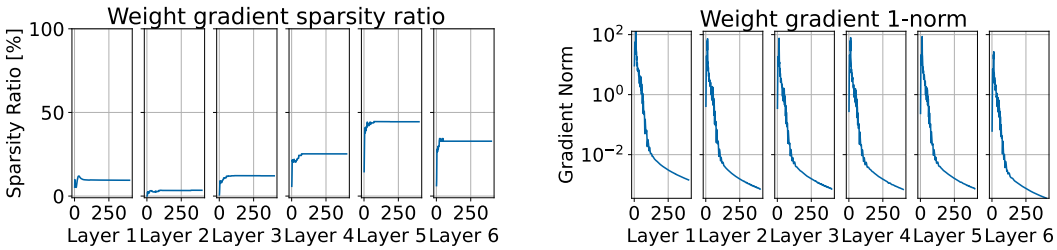



Figure 9.1: Sparsity Experiment on the CiteSeer dataset without K-FAC applied

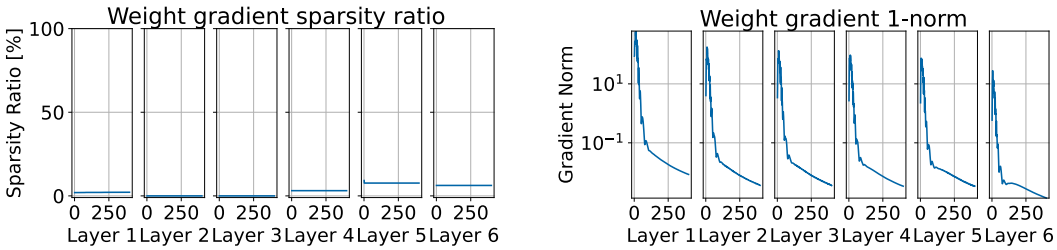



Figure 9.2: Sparsity Experiment on the CiteSeer dataset with K-FAC applied

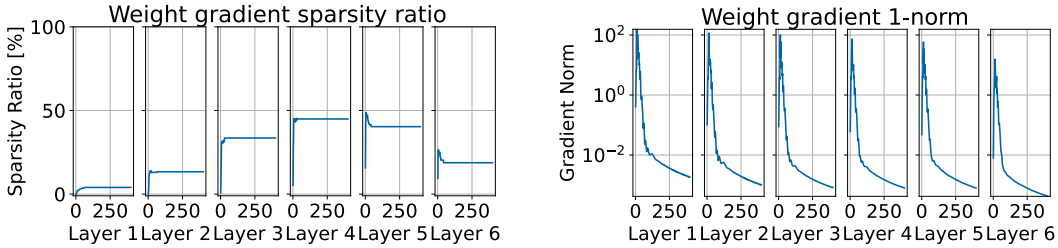



Figure 9.3: Sparsity Experiment on the PubMed dataset without K-FAC applied

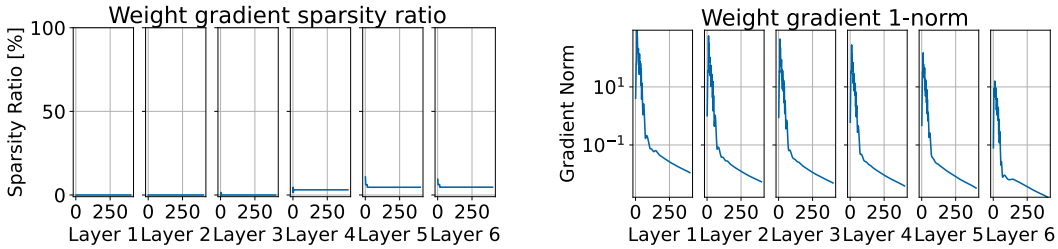



Figure 9.4: Sparsity Experiment on the PubMed dataset with K-FAC applied

## Node filtering Experiment

Additional results from the experiment described in Section 5.3.1 investigating the effect of node filtering for the data to calculate the covariance matrices.

| **Added nodes** $k$ | **Test Accuracy** [%] | **Loss** |
|---|---|---|
| 0 | 0.718± 0.007 | 0.339± 0.198 |
| 400 | 0.719± 0.004 | 0.307± 0.084 |
| 800 | 0.719± 0.004 | 0.325± 0.148 |
| 1600 | 0.719± 0.003 | 0.302± 0.085 |
| 2000 | 0.718± 0.004 | 0.317± 0.157 |
| 2400 | 0.717± 0.005 | 0.413± 0.298 |

Table 9.1: Results for PubMed for the node filtering experiment

| **Added nodes** $k$ | **Test Accuracy** [%] | **Loss** |
|---|---|---|
| 0 | 0.718± 0.007 | 0.339± 0.198 |
| 400 | 0.719± 0.004 | 0.307± 0.084 |
| 800 | 0.719± 0.004 | 0.325± 0.148 |
| 1600 | 0.719± 0.003 | 0.302± 0.085 |
| 2000 | 0.718± 0.004 | 0.317± 0.157 |
| 2400 | 0.717± 0.005 | 0.413± 0.298 |

Table 9.2: Results for CiteSeer for the node filtering experiment

## Hessian-Free experiments

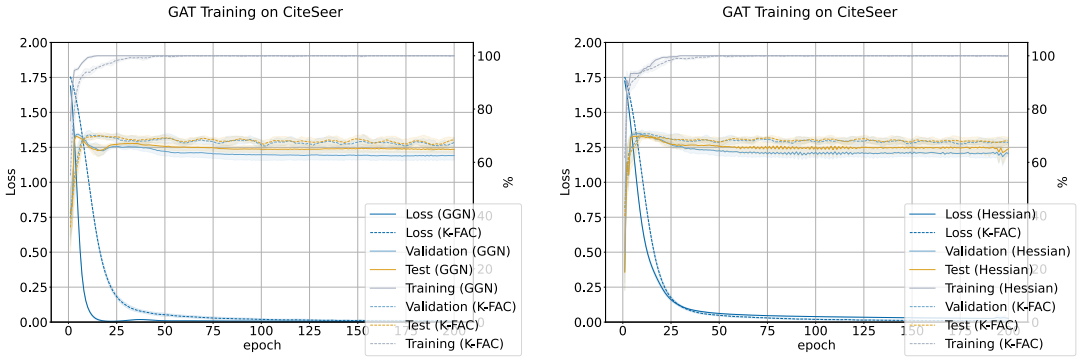



Figure 9.5: Training of a GAT with Hessian-free methods on the CiteSeer dataset. The left plot shows training with the GGN matrix, the right plot with the Hessian matrix.

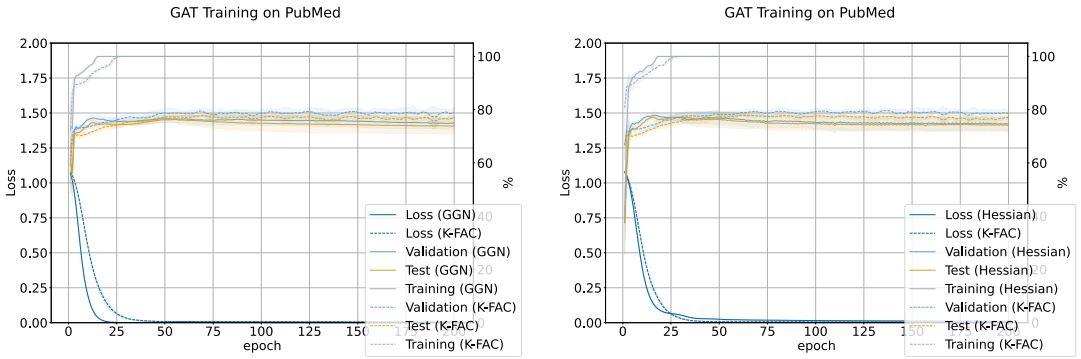



Figure 9.6: Training of a GAT with Hessian-free methods on the PubMed dataset. The left plot shows training with the GGN matrix, the right plot with the Hessian matrix.

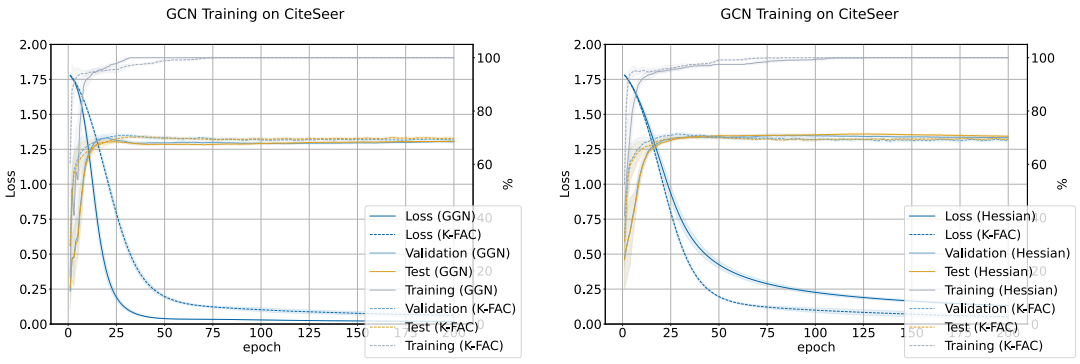



Figure 9.7: Training of a GCN with Hessian-free methods on the CiteSeer dataset. The left plot shows training with the GGN matrix, the right plot with the Hessian matrix.

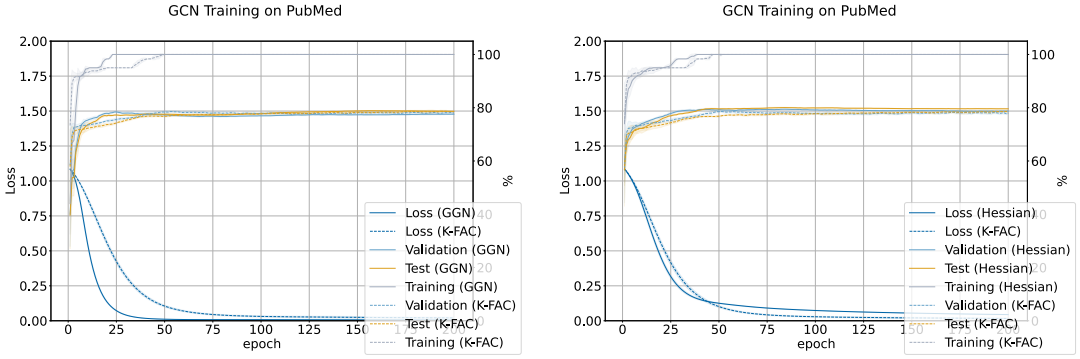



Figure 9.8: Training of a GCN with Hessian-free methods on the PubMed dataset. The left plot shows training with the GGN matrix, the right plot with the Hessian matrix.

## M-FAC experiments

The plots show a comparison between the training with M-FAC preconditioning and with K-FAC preconditioning.

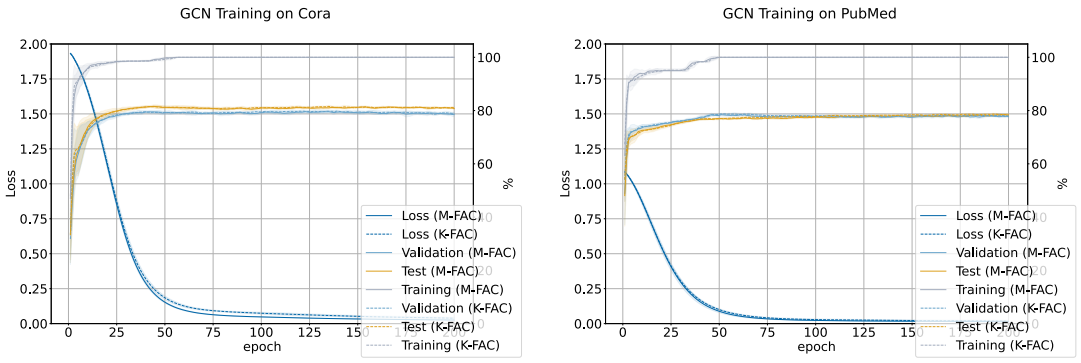



Figure 9.9: The left plot was created with the Cora dataset, the right plot was created with the PubMed dataset.

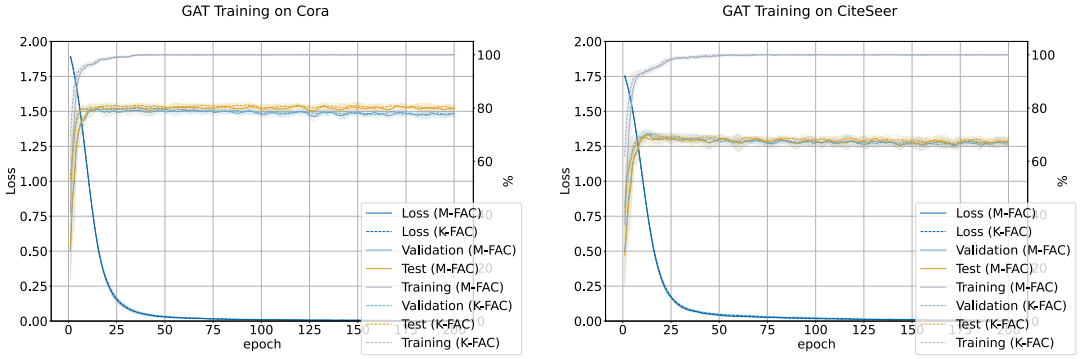



Figure 9.10: The left plot was created with the Cora dataset, the right plot was created with the CiteSeer dataset.