Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Fachbereich Informatik

Kronecker-factored Approximate Curvature for Linear Weight-Sharing Layers

Master's Thesis

in partial fulfillment of the requirements for the degree
Master of Science (M.Sc.) in Machine Learning

Runa Eschenhagen
January 2023

1. Examiner: Prof. Dr. Philipp Hennig
2. Examiner: Prof. Dr. Andreas Geiger

ABSTRACT

One of the driving forces behind the recent advances in deep learning is the development of modern neural network architectures. These often rely on components from Transformers and graph neural networks, which use linear layers with *weight-sharing*. An issue of modern deep learning is the cost associated with training increasingly large models. Improving the efficiency of the training algorithms is one approach to decrease these costs, e.g., by using second-order methods for deep learning, often motivated by Newton's method or natural gradient descent. The quantities needed for second-order methods are typically intractable and approximations are necessary. A popular approximation of the Fisher information and generalized Gauss-Newton matrix, both commonly used for second-order methods, is Kronecker-factored Approximate Curvature (K-FAC). While K-FAC has been used with many model architectures, there does not exist a framework for applying it to linear weight-sharing layers of the type used in Transformers and graph neural networks.

Hence, here we focus on K-FAC in the context of linear weight-sharing layers to enable methods using K-FAC for modern neural network architectures and to contribute to a better understanding of their properties. We identify two different settings, the *expand* and the *reduce* setting, which motivate two different flavors of the K-FAC approximation – *K-FAC-expand* and *K-FAC-reduce*. Moreover, we show that they are exact in the same simple settings as K-FAC is for regular linear layers and discuss the usage of the two approximations for Transformers and graph neural networks. Finally, we provide a proof of concept showing that both variations used for training a Vision Transformer on ImageNet can decrease the number of steps to a target validation accuracy compared to a well-tuned baseline. We hope that this work lays a foundation for future empirical and theoretical work to improve the efficiency of training algorithms and other methods which rely on K-FAC approximations for modern neural network architectures.

ACKNOWLEDGEMENTS

First and foremost, I am extremely grateful to Philipp Hennig for giving me the chance to work on research throughout my whole master's and always supporting me in every possible way and situation. I am very thankful to Frank Schneider for the enjoyable supervision of my master's thesis and of my work in the MLCommons Algorithms working group. Special thanks to Alexander Immer and Kazuki Osawa for many insightful discussions and feedback on the work in this thesis. Also, thank you to Andreas Geiger for serving as my second examiner.

Beyond this thesis, I want to thank Agustinus Kristiadi for supervising my first research project in my master's and for all the following great discussions and collaborations. Also, thank you to all collaborators I have directly worked with during my time in the Methods of Machine Learning (MoML) group, who I have not mentioned yet: Erik Daxberger, Matthias Bauer, Emilia Magnani, Nicholas Krämer, and Zachary Nado and the rest of the MLCommons Algorithms working group. Moreover, thank you to the whole MoML group for an amazing and fun research environment and the deep learning meeting group for many interesting discussions. I would also like to thank Franziska Weiler and the MLCloud team for the helpful administrative support.

My time in Tübingen would not have been the same without the friends and flatmates I spent my time with: Alexander Immer, Moritz Reiber, Christian Schlarmann, Leander Kurscheidt, Tina Keller, and Tjorve Pierau. Finally, nothing would have been possible without my parents, thank you for your unconditional support.

Contents

Chapter 1
Introduction

Recently, many applications of deep learning have received a lot of attention, from generative models which can produce complex images from text prompts (Rombach et al., 2021; Ramesh et al., 2022), to human-like text generation with language models (Brown et al., 2020) and advances in protein structure prediction (Jumper et al., 2021). Many of the models used for these applications rely on components of Transformer architectures, i.e. attention mechanisms, and graph neural networks to perform well. These modern elements of neural network architectures are therefore arguably one of the driving forces behind the recent dominance of deep learning for all kinds of machine learning tasks. One thing Transformers and graph neural networks have in common is that they use linear layers with weight-sharing, which means that the same weights are applied across multiple input dimensions. For example in language translation tasks, the same weights are applied to the features of each word in a sentence. weight-sharing is also used in convolutional neural networks layers, which are a special instance of a linear weight-sharing layer; however, they will not be the focus of this work. Overall, it seems fair to say that weight-sharing is a crucial technique of almost *all* state-of-the-art neural network architectures. The typical motivation to use weight-sharing is (i) increased efficiency, as fewer parameters have to be trained, (ii) to enforce an invariance to the position of an element within the weight-sharing dimension, and (iii) to allow for inputs with weight-sharing dimensions of varying size.

One issue with these modern model architectures is that they are typically expensive to train due to their size. The cost of training and tuning these models goes into the millions of dollars and requires training for weeks to months on large infrastructure (Strubell et al., 2020). The increasing environmental impact of this is also a commonly raised concern. One approach to decrease the costs of training large modern deep learning models is increasing the efficiency of the training algorithms. Usually (adaptive) first-order methods are used to optimize neural networks, but there is some evidence that using structured curvature estimates can speed up training in terms of steps or even wall-clock time (Martens & Grosse, 2015; Ren & Goldfarb, 2021; Osawa et al., 2022). Typically Newton's method, which uses the Hessian, or natural gradient descent (Amari, 1998), which uses the Fisher information matrix, are used to motivate these kinds of algorithms. The Hessian as well as the Fisher cannot directly be used as a curvature estimate for large models since they are too expensive to compute, store, and invert. Moreover, the Hessian is not guaranteed to be positive semi-definite (p.s.d.). Hence, in practice approximations of the Hessian are often used, like the generalized Gauss-Newton matrix (GGN). Moreover, for losses like the cross-entropy loss, the GGN is equivalent to the Fisher, connecting Newton's method to natural gradient descent. While the GGN/Fisher are guaranteed to be p.s.d., they are still generally intractable for large models. One specific approach to estimating the GGN/Fisher more practically is Kronecker-factored Approximate Curvature (Heskes, 2000; Martens & Grosse, 2015, K-FAC). It uses a block-diagonal approximation, where each block

can be written as a Kronecker product of two smaller matrices. This approximation was developed for linear layers and later extended to convolutional and recurrent neural networks (Grosse & Martens, 2016; Martens et al., 2018).

While some people have applied K-FAC to Transformers for natural language processing tasks (Zhang et al., 2019; Pauloski et al., 2021; Osawa et al., 2022), to the best of our knowledge, no previous work justifies or even just explicitly expresses the approximation implemented in these cases; there seems to be no framework or derivation of K-FAC for linear weight-sharing layers of the type used in Transformers and graph neural networks. Since K-FAC can potentially be useful to improve the efficiency of training modern neural network architectures, the goal of this work is to investigate the application of K-FAC to this type of linear weight-sharing layers. Here, we focus on the use-case of improving the efficiency of optimization and learning, but the K-FAC approximation of the GGN/Fisher is also useful for a other applications. In Bayesian deep learning, the GGN is typically used as a Hessian approximation for Laplace approximations (MacKay, 1992a; Daxberger et al., 2021) and natural gradient variational inference (Khan et al., 2018; Zhang et al., 2018; Osawa et al., 2019). These methods can be leveraged for improving predictive uncertainty quantification (Ritter et al., 2018; Kristiadi et al., 2020), online learning (Kirkpatrick et al., 2017; Pan et al., 2020), and model selection (MacKay, 1992b; Immer et al., 2021a). Additionally, the GGN/Fisher is used for model pruning and compression (LeCun et al., 1990; Singh & Alistarh, 2020). Therefore, we see the extension of K-FAC to models with linear weight-sharing layers as the first step in enabling these methods for modern deep learning architectures.

To provide sufficient context to the main part of this work, in Chapter 2 we introduce deep learning in general and motivate the idea of second-order optimization for deep learning, with a focus on the methods relevant to this work. Moreover, we introduce the benchmarks which we consider as examples for our more abstract framework and which we use for the experiments; they are a subset of the AlgoPerf benchmark by MLCommons. In Chapter 3 we try to develop a framework for thinking about K-FAC in the context of linear weight-sharing layers. Specifically, we identify two different base cases, the expand and the reduce case, which each motivates a variation of the K-FAC approximation – K-FAC-expand and K-FAC-reduce. We also look at two more concrete cases, i.e. how linear weight-sharing layers are used within a simplified dot-product attention mechanism and how to apply the framework of the two base cases to a specific graph neural network. Besides the theoretical extension of K-FAC to these cases, we also consider practical considerations, like the actual implementation and computational details. Finally, in Chapter 4 we provide some experimental data as a proof of concept of the potential usefulness of the here presented K-FAC approximations for optimizing neural networks, and conclude with a discussion of the implications of the results in this work in Chapter 5.

Chapter 2
Background

In this chapter, we provide the necessary background to follow this work. This includes a general introduction to deep learning (Section 2.1), which covers common neural network architectures (Section 2.1.1), the empirical risk minimization framework (Section 2.1.2), loss functions (Section 2.1.3), and gradient-based optimization methods (Section 2.1.4). Moreover, we cover second-order optimization in deep learning (Section 2.2) and the MLCommons AlgoPerf benchmark (Section 2.3).

2.1 Deep Learning

In this work, we focus on the standard supervised deep learning setup. It consists of a dataset $\mathcal{D}$ of N independently and identically distributed (i.i.d.) samples $\{\boldsymbol{x}_n, y_n\}_{n=1}^N$, a (deep) neural network ((D)NN) architecture, a loss function, and an optimization algorithm. In the following, these concepts will be introduced in a brief, but self-contained manner.

2.1.1 Neural Network Architectures

A deep neural network is a nonlinear function $f_{\boldsymbol{\theta}} : \mathbb{R}^D \to \mathbb{R}^C$, parameterized with $\boldsymbol{\theta} \in \mathbb{R}^P$. It typically has a layer-wise structure, i.e. it can be written as

$$f_{\boldsymbol{\theta}} = f_{\boldsymbol{\theta}_L} \circ \ldots f_{\boldsymbol{\theta}_\ell} \circ \cdots \circ f_{\boldsymbol{\theta}_1}, \tag{2.1}$$

with $\boldsymbol{\theta} = \operatorname{concat}\big(\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_\ell, \ldots, \boldsymbol{\theta}_L\big)$ and L is the number of layers of the NN; $\operatorname{concat}(\cdot, \ldots, \cdot)$ concatenates arbitrarily many vector inputs to a larger vector or arbitrarily many matrices with the same number of rows into a larger matrix with the same number of rows and as many columns as all input matrices combined.

In a fully-connected feed-forward network, we have $f_{\boldsymbol{\theta}_\ell}(\boldsymbol{x}) = \phi(\boldsymbol{W}_\ell \boldsymbol{x} + \boldsymbol{b}_\ell)$, where $\boldsymbol{x} \in \mathbb{R}^{P_{\ell,\text{in}}}$, $\boldsymbol{W}_\ell \in \mathbb{R}^{P_{\ell,\text{out}} \times P_{\ell,\text{in}}}$, $\boldsymbol{b} \in \mathbb{R}^{P_{\ell,\text{out}}}$, $\boldsymbol{\theta}_\ell = \operatorname{concat}\big(\operatorname{vec}(\boldsymbol{W}_\ell), \boldsymbol{b}_\ell\big) \in \mathbb{R}^{P_\ell}$, and $P_\ell = P_{\ell,\text{out}} P_{\ell,\text{in}} + P_{\ell,\text{out}}$; the operation $\operatorname{vec}(\cdot)$ vectorizes a matrix my concatenating its column vectors. This inner affine function is typically called *linear layer*. Additionally, ϕ is an element-wise nonlinear function, called activation function. One of the most commonly used examples for ϕ is $\text{ReLU}(\boldsymbol{x}) := \max(0, \boldsymbol{x})$, where the $\max$ is defined element-wise. A fully-connected feed-forward network is typically also called multilayer perceptron (MLP).

In this work, we are interested in linear layers with weights shared across an additional input dimension, i.e. we now have inputs $\boldsymbol{X} \in \mathbb{R}^{R \times D}$, where we have an additional dimension of size R. Ignoring the bias, the linear layer is applied to this input as $\boldsymbol{X}\boldsymbol{W}^T$, so the weight matrix $\boldsymbol{W}$ is shared across the additional first dimension, in the same way it is typically applied over a

mini-batch of data. We call linear layers applied to such inputs *linear layers with weight-sharing* or *linear weight-sharing layers*. This type of linear layer appears in many increasingly important neural network architectures, such as Transformer models and graph neural networks.

2.1.1.1 Transformers

The Transformer architecture (Vaswani et al., 2017) relies solely on the attention mechanism, which has originally been used in conjunction with recurrent structures in sequence modelling (Bahdanau et al., 2015). It allows modeling dependencies between all tokens in a sequence, independent of their distance from each other. In contrast, in classical recurrent networks, it is hard to model long-ranging dependencies (Hochreiter, 1991; Bengio et al., 1994).

Attention. An attention operation can generally be defined as a function of a query $\boldsymbol{q} \in \mathbb{R}^D$ and a collection of R pairs of keys $\boldsymbol{K} \in \mathbb{R}^{R \times D}$ and values $\boldsymbol{V} \in \mathbb{R}^{R \times C}$, both stacked in matrices, where $\boldsymbol{v}_r \in \mathbb{R}^C$ denotes the rth row of V. Since attention operations are defined for sequences, R is the sequence length, e.g. the number of words in a sentence.

$$\text{Attention}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V}) := \sum_{r=1}^{R} \alpha_r(\boldsymbol{q}, \boldsymbol{K}) \boldsymbol{v}_r, \tag{2.2}$$

with attention weights $\alpha_r(\boldsymbol{q}, \boldsymbol{K}) \in [0, 1]$ and it holds that $\sum_{r=1}^{R} \alpha_r(\boldsymbol{q}, \boldsymbol{K}) = 1$; many different functions have been proposed for these attention weights. Through this way of writing the attention mechanism, it can be seen as a parametric version of the weighted sum used in Gaussian process prediction. The query $\boldsymbol{q}$ corresponds to the input we want to predict the target for, $\boldsymbol{K}$ to the collection of training inputs, so $R = N$, and $\boldsymbol{V}$ are the stacked training targets (Tsai et al., 2019). Notably, from this formulation, it also becomes visible that an attention mechanism can be seen as a graph neural network (c.f. Section 2.1.1.2) with a dense adjacency matrix (Joshi, 2020).

We can also stack M queries in a matrix $\boldsymbol{Q} \in \mathbb{R}^{M \times D}$ and compute the attention operation via matrix multiplication as

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \alpha(\boldsymbol{Q}, \boldsymbol{K}) \boldsymbol{V}. \tag{2.3}$$

The attention mechanism used in the Transformer architecture is called scaled dot-product attention and is defined as

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}_{\text{row}}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{D}}\right)\boldsymbol{V}, \tag{2.4}$$

where $\text{softmax}_{\text{row}}(\cdot)$ is the softmax function (c.f. Equation (2.17)) which is applied row-wise on a matrix. The scaling by $1/\sqrt{D}$ is applied to avoid vanishing gradients due to input values to the softmax function with large magnitude for large D. Moreover, the Transformer uses *self-attention*, i.e. $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$ are all linear transformations of the same input matrix $\boldsymbol{X} \in \mathbb{R}^{R \times D_{\text{in}}}$; note, that we learn separate linear transformations $\boldsymbol{W}^Q \in \mathbb{R}^{D \times D_{\text{in}}}$, $\boldsymbol{W}^K \in \mathbb{R}^{D \times D_{\text{in}}}$, and $\boldsymbol{W}^V \in \mathbb{R}^{C \times D_{\text{in}}}$ for $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$. In the Transformer architecture, self-attention is combined with *multi-head attention*, which is defined as

$$\text{MultiHead}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) := \text{concat}(\boldsymbol{H}_1, \ldots, \boldsymbol{H}_H)\boldsymbol{W}^O, \tag{2.5}$$

with $\boldsymbol{H}_h = \text{Attention}(\boldsymbol{Q}\boldsymbol{W}_h^{Q^R}, \boldsymbol{K}\boldsymbol{W}_h^{K^T}, \boldsymbol{V}\boldsymbol{W}_h^{V^T})$ and H heads. When combined with self-attention, we would have $\text{MultiHead}(\boldsymbol{X}, \boldsymbol{X}, \boldsymbol{X})$, i.e. $\boldsymbol{Q} = \boldsymbol{K} = \boldsymbol{V}$. To avoid confusion, note that self-attention does not mean that all three inputs to the attention operation are the same, but they are linear combinations of the same input.

Crucially, here we can observe that the weight matrices are shared across the sequence dimension of size R: since the weight matrices are multiplied to the input, $\boldsymbol{Q}$, $\boldsymbol{K}$ or $\boldsymbol{V}$, from the right side, we can interpret this as a batched matrix-vector product of the weight matrix with the rows of the input, i.e. calculating $\boldsymbol{W}_h^Q \boldsymbol{q}_r$ for all $r \in \{1, \dots, R\}$ and stacking the results as rows of a matrix is equivalent to calculating $\boldsymbol{Q}\boldsymbol{W}_h^{Q^T}$.

Transformer architectures and their applications. The Transformer model introduced in Vaswani et al. (2017) stacks multiple multi-head attention layers together with fully-connected layers and has an encoder and decoder structure. Additionally, normalization layers, positional encodings, and input masks are used.

While the Transformer has originally been introduced for natural language processing, e.g. for translation tasks, they have also been applied to image classification tasks (Parmar et al., 2018; Dosovitskiy et al., 2021). Since the attention operation scales quadratically in the sequence length R, interpreting the pixels of an image as a sequence does not scale to realistic image sizes. As a solution, Cordonnier et al. (2020) and Dosovitskiy et al. (2021) propose to reshape the input images into sequences of R flattened 2d patches. The Vision Transformer (ViT) architecture (Dosovitskiy et al., 2021) then uses a typical Transformer encoder to process the embedded sequence of image patches and classifies the images with a fully-connected head based on the encoder outputs. Before the encoder output is passed to the head network, global average pooling, i.e. a mean operation, is applied to reduce the sequence dimension. This reduction is crucial for our results in Chapter 3.

2.1.1.2 Graph Neural Networks

A different class of model architectures that use linear layers with weight-sharing are graph neural networks (GNNs).

Graph convolutional network for node classification. A popular type of GNNs is called graph convolutional network (GCN). A GCN defines a convolution operation on graph structures, by repeatedly aggregating feature information over the neighborhood of a node. As a regular convolutional neural network, it also uses weight-sharing; see Liu et al. (2020) for a comprehensive discussion on weight-sharing in GCNs. However, in contrast to the other models presented here, they utilize a slightly different type of weight-sharing, which will become apparent in Equation (2.7). Nevertheless, we briefly mention this case here, since the only work on K-FAC for GNNs has been on this model architecture and we will explicitly show how K-FAC was applied in this case in Section 3.3; this relies on the notation introduced here.

A graph is defined as $\mathcal{G} := (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of N nodes and $\mathcal{E}$ the set of edges. The edges can be encoded relative to the nodes in an adjacency matrix $\boldsymbol{C} \in \mathbb{R}^{N \times N}$ with $C_{ij} = 0$ if there is no edge and $C_{ij} = 1$ if there is one. Typically, they are used for node and graph classification tasks. Here, we focus on node classification, e.g. classifying scientific publications which are represented as nodes in a citation network into topics (Sen et al., 2008).

The ℓth GCN layer is defined as

$$f_{\boldsymbol{\theta}_\ell}(\boldsymbol{X}) = \phi(\hat{\boldsymbol{C}}\boldsymbol{X}\boldsymbol{W}_\ell^T) \tag{2.6}$$

which is identical to a regular dense linear layer from Section 2.1.1, but the input matrix $\boldsymbol{X} \in \mathbb{R}^{N \times P_{\ell,\text{in}}}$, which has the N node features $\boldsymbol{x}_n$ of size $P_{\ell,\text{in}}$ stacked in the rows, is first transformed by the normalized adjacency matrix $\hat{\boldsymbol{C}} := (\boldsymbol{D} + \mathbf{I}_N)^{-\frac{1}{2}}(\boldsymbol{C} + \mathbf{I}_N)(\boldsymbol{D} + \mathbf{I}_N)^{-\frac{1}{2}}$, where $\boldsymbol{D}$ is the diagonal node degree matrix of the graph and $\mathbf{I}_N$ is the $N \times N$ identity matrix.

Defining

$$\tilde{\boldsymbol{x}}_n := \sum_{j=1}^{N} \hat{\boldsymbol{C}}_{nj} \boldsymbol{x}_j = \sum_{j \in \mathcal{N}(n)} \hat{\boldsymbol{C}}_{nj} \boldsymbol{x}_j, \tag{2.7}$$

we can express the forward pass for a single node and layer as

$$f_{\boldsymbol{\theta}_\ell}(\tilde{\boldsymbol{x}}_n) = \phi(\boldsymbol{W}_\ell \tilde{\boldsymbol{x}}_n), \tag{2.8}$$

where $\mathcal{N}(n) := \{j \in \{1, \dots, N\} | \hat{\boldsymbol{C}}_{nj} \neq 0\}$ is the neighborhood of the node with index n. Notably, the forward pass for a single node $\boldsymbol{x}_n$ depends on its neighborhood, i.e. we cannot express the forward pass for the node without access to the feature information of the nodes in its neighborhood $\mathcal{N}(n)$. Moreover, we can now see that the forward pass through the linear layer, i.e. the matrix multiplication of the weight matrix $\boldsymbol{W}_\ell$ with the transformed input $\tilde{\boldsymbol{x}}_n$, does not need the notion of weight-sharing anymore, in the sense, that we do not need a batched matrix-vector product over a weight-sharing dimension. This is because we aggregate over each node's neighborhood, over which the weights are shared, *before* the matrix-vector product. Hence, in contrast to the Graph network introduced in the next paragraph, this model does not require special consideration when applying K-FAC (c.f. Section 3.3).

Graph network for graph classification. One more general formulation of a GNN is an instance of the graph network introduced in Battaglia et al. (2018). The graph network in its general form takes a graph $\mathcal{G} = (\boldsymbol{u}, \mathcal{V}, \mathcal{E})$, where $\boldsymbol{u} \in \mathbb{R}^{D_u}$ are the global features of the graph, and $\mathcal{V}$ and $\mathcal{E}$ are the sets of nodes and edges, respectively, just as before. We can also write the ith graph of a dataset of N graphs as a 5-tuple $\mathbb{X}_n^G := (\boldsymbol{x}_n^u, \boldsymbol{X}_n^V, \boldsymbol{X}_n^E, \boldsymbol{r}_n, \boldsymbol{s}_n)$, with global features $\boldsymbol{x}_n^u \in \mathbb{R}^{D_u}$, node features $\boldsymbol{X}_n^V \in \mathbb{R}^{N_n^V \times D_V}$, and edge features $\boldsymbol{X}_n^E \in \mathbb{R}^{N_n^E \times D_E}$ for all $n = 1, \dots, N$. The two vectors $\boldsymbol{r}_n \in \mathbb{R}^{N_n^E}$ and $\boldsymbol{s}_n \in \mathbb{R}^{N_n^E}$ contain the indices of the receiving and sending nodes of each edge, respectively. Using these indices, we define $\boldsymbol{X}_{n,\boldsymbol{r}_n}^V \in \mathbb{R}^{N_n^E \times D_V}$ and $\boldsymbol{X}_{n,\boldsymbol{s}_n}^V \in \mathbb{R}^{N_n^E \times D_V}$ which contain the node features $\boldsymbol{X}_n^V$ at indices $\boldsymbol{s}_n$ and $\boldsymbol{r}_n$, respectively. Note, that these graph inputs unfortunately cannot trivially be batched by stacking them, since the number of nodes N_n^V or edges N_n^E are not necessarily the same for all $n \in \{1, \dots, N\}$.

A graph network block updates the 3-tuple $(\boldsymbol{x}_n^u, \boldsymbol{X}_n^V, \boldsymbol{X}_n^E)$ by using three update functions ϕ,

$$\begin{aligned} \boldsymbol{X}_n^E &\leftarrow \phi^E(\boldsymbol{X}_n^E, \boldsymbol{X}_{n,\boldsymbol{r}_n}^V, \boldsymbol{X}_{n,\boldsymbol{s}_n}^V, \boldsymbol{x}_n^u) \\ \boldsymbol{X}_n^V &\leftarrow \phi^V(\boldsymbol{X}_n^V, \tilde{\boldsymbol{X}}_n^E, \boldsymbol{x}_n^u) \\ \boldsymbol{x}_n^u &\leftarrow \phi^u(\boldsymbol{x}_n^u, \bar{\boldsymbol{X}}_n^V, \bar{\boldsymbol{X}}_n^E), \end{aligned} \tag{2.9}$$

and three permutation-invariant aggregation functions ρ

$$\begin{aligned} \tilde{\boldsymbol{X}}_n^E &\leftarrow \rho^{E \to V}(\boldsymbol{X}_n^E) \\ \bar{\boldsymbol{X}}_n^E &\leftarrow \rho^{E \to u}(\boldsymbol{X}_n^E) \\ \bar{\boldsymbol{X}}_n^V &\leftarrow \rho^{V \to u}(\boldsymbol{X}_n^V). \end{aligned} \tag{2.10}$$

Examples of these aggregation functions include element-wise summation, mean, or maximum.

One forward pass through a graph network block corresponds to the following steps, where each step is executed for all $n \in \{1, \dots, N\}$:

1. Update edges $\boldsymbol{X}_n^E$ with $\phi^E(\boldsymbol{X}_n^E, \boldsymbol{X}_{n,\boldsymbol{r}_n}^V, \boldsymbol{X}_{n,\boldsymbol{s}_n}^V, \boldsymbol{x}_n^u)$.
2. Aggregate updated edges over all nodes in $\tilde{\boldsymbol{X}}_n^E \in \mathbb{R}^{N_n^V \times D_E}$ using $\rho^{E\to V}(\boldsymbol{X}_n^E)$.
3. Update nodes $\boldsymbol{X}_n^V$ using $\phi^V(\boldsymbol{X}_n^V, \tilde{\boldsymbol{X}}_n^E, \boldsymbol{x}_n^u)$.
4. Aggregate updated edges over all graphs in $\bar{\boldsymbol{X}}_n^E \in \mathbb{R}^{D_E}$ using $\rho^{E\to u}(\boldsymbol{X}_n^E)$.
5. Aggregate updated nodes over all graphs in $\bar{\boldsymbol{X}}_n^V \in \mathbb{R}^{D_V}$ using $\rho^{V\to u}(\boldsymbol{X}_n^V)$.
6. Update global features $\boldsymbol{x}_n^u$ with $\phi^u(\boldsymbol{x}_n^u, \bar{\boldsymbol{X}}_n^V, \bar{\boldsymbol{X}}_n^E)$.

In this work, we consider graph classification; for example, molecules can be represented as graphs and we could classify them according to some chemical property (c.f. the OGBG workload in Section 2.3.1). We specifically consider a graph network instance with simple MLPs for all update functions ϕ, and an element-wise sum for the aggregation functions ρ. Moreover, multiple of these graph network blocks can be stacked on top of each other. To classify the input graphs, an MLP is applied to the global features $\boldsymbol{x}_n^u$ after they are updated by the last graph network block.

To be more precise, the update functions are in this case specified as

$$\begin{aligned}
\phi^E(\boldsymbol{X}_n^E, \boldsymbol{X}_{n,\boldsymbol{r}_n}^V, \boldsymbol{X}_{n,\boldsymbol{s}_n}^V, \boldsymbol{x}_n^u) &:= \mathrm{concat}(\boldsymbol{X}_n^E, \boldsymbol{X}_{n,\boldsymbol{r}_n}^V, \boldsymbol{X}_{n,\boldsymbol{s}_n}^V, \mathrm{repeat}_{N_n^E}(\boldsymbol{x}_n^u))\boldsymbol{W}^{E^T} \\
\phi^V(\boldsymbol{X}_n^V, \tilde{\boldsymbol{X}}_n^E, \boldsymbol{x}_n^u) &:= \mathrm{concat}(\boldsymbol{X}_n^V, \tilde{\boldsymbol{X}}_n^E, \mathrm{repeat}_{N_n^V}(\boldsymbol{x}_n^u))\boldsymbol{W}^{V^T} \\
\phi^u(\boldsymbol{x}_n^u, \bar{\boldsymbol{X}}_n^V, \bar{\boldsymbol{X}}_n^E) &:= \boldsymbol{W}^u \mathrm{concat}(\boldsymbol{x}_n^u, \bar{\boldsymbol{X}}_n^V, \bar{\boldsymbol{X}}_n^E)
\end{aligned} \tag{2.11}$$

with $\boldsymbol{W}^E \in \mathbb{R}^{D_E \times (D_E + 2D_V + D_u)}$, $\boldsymbol{W}^V \in \mathbb{R}^{D_V \times (D_V + D_E + D_u)}$, and $\boldsymbol{W}^u \in \mathbb{R}^{D_u \times 3D_u}$.

Note, that this is a simplification, since in reality the update functions ϕ are MLPs with ReLU activations, layer normalization (Ba et al., 2016), and dropout (Hinton et al., 2012b). Also, we omit the potential bias vectors. However, these components are not relevant for deriving K-FAC for the linear layers within these networks, which is why we can omit them here for simplicity.

More importantly, we can observe that this type of GNN shares its weights over each graph's edges and nodes: just as for the Transformer models, we apply the (transposed) weight matrices from the right side of the input of the layers of type ϕ^E and ϕ^V, i.e. for updating the edge and node features. However, since the number of edges N_n^E and the number of nodes N_n^V is not necessarily the same for all N graphs, we now have a weight-sharing dimension of size R_n, which depends on the nth input. We have specifically chosen this notation of the inputs to show that this GNN indeed uses this type of weight-sharing. This is also necessary to easily express our result in Section 3.3.

2.1.2 Empirical Risk Minimization

Supervised neural network training is typically formulated as *empirical risk minimization*. The goal is to minimize a loss function $\ell : \mathbb{R}^C \times \mathbb{R}^C \to \mathbb{R}$ by adjusting the parameters $\boldsymbol{\theta} \in \mathbb{R}^P$ of a neural network $f_{\boldsymbol{\theta}} : \mathbb{R}^D \to \mathbb{R}^C$ on an i.i.d. training dataset $\mathcal{D} = \{\boldsymbol{x}_n, y_n\}_{n=1}^N$ with $\boldsymbol{x}_n \in \mathbb{R}^D$ and $y_n \in \mathbb{R}$. The *true risk* is defined as

$$\mathcal{L}_{\text{true}}(f_{\boldsymbol{\theta}}) = \mathbb{E}_{\boldsymbol{x},y\sim p_{\text{data}}(\boldsymbol{x},y)}\big[\ell(y, f_{\boldsymbol{\theta}}(\boldsymbol{x}))\big]. \tag{2.12}$$

Since we usually do not have access to $p_{\text{data}}(\boldsymbol{x}, y)$, but only to our dataset $\mathcal{D}$, we minimize the *empirical risk*

$$\mathcal{L}_{\text{emp}}(f_{\boldsymbol{\theta}}, \mathcal{D}) = \frac{1}{N} \sum_{n=1}^{N} \ell(y_n, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)), \tag{2.13}$$

to find $\boldsymbol{\theta}^* = \operatorname{argmin}_{\boldsymbol{\theta}} \mathcal{L}_{\text{emp}}(f_{\boldsymbol{\theta}}, \mathcal{D})$.

Since we are interested in learning and not just optimization, we want $\boldsymbol{\theta}^*$ to generalize to unseen data, i.e. yield low empirical risk on unseen data points. Hence, we typically optimize a regularized empirical risk

$$\mathcal{L}_{\text{reg}}(f_{\boldsymbol{\theta}}, \mathcal{D}) = \frac{1}{N} \sum_{n=1}^{N} \ell(y_n, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) + r(\boldsymbol{\theta}), \tag{2.14}$$

where $r(\boldsymbol{\theta})$ is an *explicit* regularizer, and use other *implicit* regularization methods such as data augmentation (Baird, 1993; Wong et al., 2016) or dropout (Hinton et al., 2012b).

Moreover, we can connect learning via regularized empirical risk minimization to probabilistic inference for losses $\ell(y_n, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))$ which correspond to a valid negative log-likelihood $-\log p(y_n | f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))$ and a regularizer which is also a valid negative log density. In these cases, empirical risk minimization is equivalent to maximum likelihood estimation and maximum a-posteriori estimation (MAP) for the regularized case:

$$\mathcal{L}_{\text{MAP}}(f_{\boldsymbol{\theta}}, \mathcal{D}) = \underbrace{-\sum_{n=1}^{N} \log p(y_n | f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))}_{N\mathcal{L}_{\text{emp}}} \underbrace{- \log p(\boldsymbol{\theta})}_{Nr(\boldsymbol{\theta})}, \tag{2.15}$$

where we can see that $\boldsymbol{\theta}^*$ coincides for $\mathcal{L}_{\text{reg}}(f_{\boldsymbol{\theta}}, \mathcal{D})$ and $\mathcal{L}_{\text{MAP}}(f_{\boldsymbol{\theta}}, \mathcal{D})$, since $N\mathcal{L}_{\text{reg}}(f_{\boldsymbol{\theta}}, \mathcal{D}) = \mathcal{L}_{\text{MAP}}(f_{\boldsymbol{\theta}}, \mathcal{D})$. Hence, in these cases $\boldsymbol{\theta}^*$ is the argmax of the intractable posterior distribution

$$p(\boldsymbol{\theta} | \mathcal{D}) = \frac{p(\mathcal{D} | \boldsymbol{\theta}) p(\boldsymbol{\theta})}{p(\mathcal{D})} \propto p(\mathcal{D} | \boldsymbol{\theta}) p(\boldsymbol{\theta}) = p(\boldsymbol{\theta}) \prod_{n=1}^{N} p(y_n | f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)). \tag{2.16}$$

This correspondence holds, among others, for the common cross-entropy (CE) loss for classification and the mean-square error (MSE) loss for regression, which are the only losses considered here. Also, the popular weight decay or L2 regularizer corresponds to an isotropic Gaussian prior over the weights (Hinton, 1987; Krogh & Hertz, 1991; MacKay, 1992c); we assume this prior because its Hessian takes a simple (constant) form and we do not explicitly consider it from now on.

2.1.3 Loss Functions for Classification and Regression

We consider the cross-entropy loss since it is used in all the benchmarks we discuss in this work (c.f. Section 2.3.1 and Chapter 4) and the mean-square error loss since we use it for the theoretical statements in Section 3.1. Also, we state the gradients and Hessians of the losses w.r.t. the model outputs, because they appear in the derivation and definition of the generalized Gauss-Newton matrix (Section 2.2.2), which we will use for our results in Chapter 3.

Cross-entropy loss. To use the cross-entropy loss, we first have to map the outputs $\boldsymbol{f} \in \mathbb{R}^C$ of the neural network to probabilities via the softmax function

$$p(\boldsymbol{f})_i = \frac{\exp(f_i)}{\sum_{c=1}^{C} \exp(f_c)}. \tag{2.17}$$

The cross-entropy loss is equivalent to using a categorical likelihood, i.e.

$$p(\mathcal{D}|f_{\boldsymbol{\theta}}) = \prod_{n=1}^{N} \prod_{c=1}^{C} p(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))_c^{\tilde{y}_{nc}}, \tag{2.18}$$

where $\tilde{\boldsymbol{y}}_n = \text{onehot}(y_n) \in \{0, 1\}^C$. Hence, the corresponding loss or the negative log-likelihood is

$$\begin{aligned} \mathcal{L}_{\text{CE}}(f_{\boldsymbol{\theta}}, \mathcal{D}) &= -\sum_{n=1}^{N} \sum_{c=1}^{C} \tilde{y}_{nc} \log p(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))_c \\ &= \sum_{n=1}^{N} \log \big(\sum_{c=1}^{C} \exp(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_c) \big) - f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_*, \end{aligned} \tag{2.19}$$

where $f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_* := \{f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_i | \tilde{y}_{ni} = 1\}$. For this loss, the gradient w.r.t. $f_{\boldsymbol{\theta}}$ is sum of the residuals $\boldsymbol{r}_n$

$$\nabla_{f_{\boldsymbol{\theta}}} \mathcal{L}_{\text{CE}}(f_{\boldsymbol{\theta}}, \mathcal{D}) = \sum_{n=1}^{N} \underbrace{p(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) - \tilde{\boldsymbol{y}}_n}_{=:\boldsymbol{r}_n} \in \mathbb{R}^C. \tag{2.20}$$

and the Hessian w.r.t. the model outputs is

$$\boldsymbol{H}_{f_{\boldsymbol{\theta}}} \mathcal{L}_{\text{CE}}(f_{\boldsymbol{\theta}}, \mathcal{D}) = \sum_{n=1}^{N} \text{diag}(p(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))) - p(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) p(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))^T \in \mathbb{R}^{C \times C}; \tag{2.21}$$

the $\text{diag}(\cdot)$ operation constructs a diagonal matrix with the input vector on its diagonal.

Mean-square error loss. The mean-square error loss is equivalent to using a (multivariate) Gaussian likelihood, i.e.

$$p(\mathcal{D}|f_{\boldsymbol{\theta}}) \propto \prod_{n=1}^{N} \exp\bigg(-\frac{1}{2} (f_{\boldsymbol{\theta}}(\boldsymbol{x}_n) - \boldsymbol{y}_n)^T \boldsymbol{\Sigma}^{-1} (f_{\boldsymbol{\theta}}(\boldsymbol{x}_n) - \boldsymbol{y}_n) \bigg), \tag{2.22}$$

where $\boldsymbol{\Sigma}^{-1} \in \mathbb{R}^{C \times C}$ is the noise precision and the targets are now in $\boldsymbol{y} \in \mathbb{R}^C$; we disregard factors constant w.r.t. $\boldsymbol{\theta}$. The corresponding loss or the negative log-likelihood is then

$$\mathcal{L}_{\text{MSE}}(f_{\boldsymbol{\theta}}, \mathcal{D}) = \frac{1}{2} \sum_{n=1}^{N} (f_{\boldsymbol{\theta}}(\boldsymbol{x}_n) - \boldsymbol{y}_n)^T \boldsymbol{\Sigma}^{-1} (f_{\boldsymbol{\theta}}(\boldsymbol{x}_n) - \boldsymbol{y}_n), \tag{2.23}$$

the gradient w.r.t. the model outputs is the sum over the (weighted) residuals $\boldsymbol{r}_n$

$$\nabla_{f_{\boldsymbol{\theta}}} \mathcal{L}_{\text{MSE}}(f_{\boldsymbol{\theta}}, \mathcal{D}) = \sum_{n=1}^{N} \underbrace{\boldsymbol{\Sigma}^{-1}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n) - \boldsymbol{y}_n)}_{=\boldsymbol{r}_n} \in \mathbb{R}^C, \tag{2.24}$$

and the Hessian w.r.t. the model outputs is simply

$$\boldsymbol{H}_{f_{\boldsymbol{\theta}}} \mathcal{L}_{\text{MSE}}(f_{\boldsymbol{\theta}}, \mathcal{D}) = N\boldsymbol{\Sigma}^{-1} \in \mathbb{R}^{C \times C}. \tag{2.25}$$

Note that for both, the categorical and the Gaussian likelihood, the loss Hessian w.r.t. the model outputs does not depend on the labels at all, and for the Gaussian likelihood not even on the inputs. This is relevant for the content of Section 2.2 and Chapter 3.

2.1.4 Gradient-Based Optimization

We want to learn from data via empirical risk minimization, or in our case equivalently MAP inference, as introduced in Section 2.1.2. Typically, iterative gradient-based algorithms are used to optimize these objectives. We simplify notation and write $\mathcal{L}(\boldsymbol{\theta})$ for our loss function $\mathcal{L}(f_{\boldsymbol{\theta}}, \mathcal{D})$, to emphasize the loss as a function of the parameters. Gradient descent-based algorithms can be derived by finding the minimizer to a local approximation to the actual problem. We use a first-order Taylor expansion of the loss $\mathcal{L}(\boldsymbol{\theta}_t)$ around the current iterate of the parameters $\boldsymbol{\theta}_t$ at time step t, and add a quadratic term weighted by $1/2\alpha$ with curvature $\boldsymbol{C}(\boldsymbol{\theta}_t) \in \mathbb{R}^{P \times P}$,

$$m_t(\boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{\theta}_t) + \boldsymbol{g}(\boldsymbol{\theta}_t)^T(\boldsymbol{\theta} - \boldsymbol{\theta}_t) + \frac{1}{2\alpha}(\boldsymbol{\theta} - \boldsymbol{\theta}_t)^T \boldsymbol{C}(\boldsymbol{\theta}_t)(\boldsymbol{\theta} - \boldsymbol{\theta}_t), \tag{2.26}$$

where $\boldsymbol{g}(\boldsymbol{\theta}_t) = \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_t)$ is the (empirical) gradient evaluated at $\boldsymbol{\theta}_t$. The simple gradient descent algorithm arises from setting $\boldsymbol{C}(\boldsymbol{\theta}_t) = \mathbf{I}_P$. The local problem then becomes

$$m_t(\boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{\theta}_t) + \boldsymbol{g}(\boldsymbol{\theta}_t)^T(\boldsymbol{\theta} - \boldsymbol{\theta}_t) + \frac{1}{2\alpha}\|\boldsymbol{\theta} - \boldsymbol{\theta}_t\|_2^2, \tag{2.27}$$

which can be seen as a first-order Taylor approximation with an added regularization term to discourage large steps in parameter space. To find the next iterate, $\boldsymbol{\theta}_{t+1}$, we now minimize this approximate local model,

$$\boldsymbol{\theta}_{t+1} = \operatorname*{argmin}_{\boldsymbol{\theta}} m_t(\boldsymbol{\theta}) = \boldsymbol{\theta}_t - \alpha \boldsymbol{g}(\boldsymbol{\theta}_t), \tag{2.28}$$

which is the classic gradient descent update with step size or learning rate α.

Since we are often interested in large datasets and models, it is prohibitively expensive to calculate the gradient for the whole training set at each iteration. As a stochastic approximation, we draw random subsets of M samples, called mini-batches $\mathcal{M}_t$ of the training data at each training step and calculate the mini-batch gradient $\hat{\boldsymbol{g}}(\boldsymbol{\theta}_t) = \nabla_{\boldsymbol{\theta}} \hat{\mathcal{L}}(\boldsymbol{\theta}_t) = \nabla_{\boldsymbol{\theta}} \mathcal{L}(f_{\boldsymbol{\theta}_t}, \mathcal{M}_t)$. If we replace the gradient with the mini-batch gradient, the update step in Equation (2.28) then becomes stochastic gradient descent (Robbins & Monro, 1951, SGD).

Many additional heuristics have been proposed to further improve the properties of SGD. *Momentum* methods, such as heavy ball (Polyak, 1964) and Nesterov (Nesterov, 1983) momentum, take the gradient information at different points into account. *Adaptive* methods, such as Adam

(Kingma & Ba, 2015), AdaGrad (Duchi et al., 2011), and RMSProp (Hinton et al., 2012a), adopt the learning rate for each parameter separately by leveraging quantities which change during the training process. The most common quantity used for this purpose is the element-wise squared mini-batch gradient $\hat{\boldsymbol{g}}(\boldsymbol{\theta}_t) \odot \hat{\boldsymbol{g}}(\boldsymbol{\theta}_t)$. This can be seen as choosing a different $\boldsymbol{C}(\boldsymbol{\theta}_t)$ (c.f. Section 2.2). Hence, there is only a blurry line between first- and second-order methods, e.g. people also refer to methods using structured approximations to the uncentered (average) mini-batch gradient covariance $\hat{\boldsymbol{g}}(\boldsymbol{\theta}_t)\hat{\boldsymbol{g}}(\boldsymbol{\theta}_t)^T$ as second-order methods (Anil et al., 2020). However, methods using a diagonal $\boldsymbol{C}(\boldsymbol{\theta}_t)$, like Adam, are typically considered as first-order methods.

2.2 Second-Order Optimization

Second-order methods choose a different curvature matrix $\boldsymbol{C}(\boldsymbol{\theta}_t)$ to increase the quality of the local approximation and therefore, of the update steps. Minimizing Equation (2.26), assuming that $\boldsymbol{C}(\boldsymbol{\theta}_t)$ is positive definite (p.d.), we get the update step

$$\boldsymbol{\theta}_{t+1} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, m_t(\boldsymbol{\theta}) = \boldsymbol{\theta}_t - \alpha \boldsymbol{C}(\boldsymbol{\theta}_t)^{-1}\boldsymbol{g}(\boldsymbol{\theta}_t). \tag{2.29}$$

Since $\boldsymbol{C}(\boldsymbol{\theta}_t)$ is quadratic in the number of parameters P, and its inversion of cubic computational complexity, a full matrix is typically not a feasible choice in practice.

In the following, we will introduce different choices for $\boldsymbol{C}(\boldsymbol{\theta}_t)$.

2.2.1 Newton's Method

The second-order Taylor expansion might be the obvious choice for a local quadratic approximation. We have

$$m_t(\boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{\theta}_t) + \boldsymbol{g}(\boldsymbol{\theta}_t)^T(\boldsymbol{\theta} - \boldsymbol{\theta}_t) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_t)^T \boldsymbol{H}(\boldsymbol{\theta}_t)(\boldsymbol{\theta} - \boldsymbol{\theta}_t), \tag{2.30}$$

where we set $\alpha = 1$ and $\boldsymbol{C}(\boldsymbol{\theta}_t) = \boldsymbol{H}(\boldsymbol{\theta}_t) := \nabla^2_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}_t) \in \mathbb{R}^{P\times P}$ is the Hessian of the loss w.r.t. the parameters. The resulting update step,

$$\boldsymbol{\theta}_{t+1} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, m_t(\boldsymbol{\theta}) = \boldsymbol{\theta}_t - \boldsymbol{H}(\boldsymbol{\theta}_t)^{-1}\boldsymbol{g}(\boldsymbol{\theta}_t), \tag{2.31}$$

together with a stopping condition and a rule to choose the step size, such as backtracking line search, is well-known in convex optimization as Newton's method (Boyd & Vandenberghe, 2004). Notably, we cannot generally assume that the Hessian is p.s.d., since we consider non-convex problems in deep learning. This motivates p.s.d. approximations to the Hessian.

2.2.2 Generalized Gauss-Newton

We can decompose the Hessian into two terms by applying the chain rule to the split $\ell \circ f_{\boldsymbol{\theta}}$[1],

$$\boldsymbol{H}_{\boldsymbol{\theta}} \ell(y, f_{\boldsymbol{\theta}}(\boldsymbol{x})) = \underbrace{\boldsymbol{J}_{\boldsymbol{\theta}} f_{\boldsymbol{\theta}}(\boldsymbol{x})^T \boldsymbol{H}_{f_{\boldsymbol{\theta}}} \ell(y, f_{\boldsymbol{\theta}}(\boldsymbol{x})) \boldsymbol{J}_{\boldsymbol{\theta}} f_{\boldsymbol{\theta}}(\boldsymbol{x})}_{=:\mathbf{GGN}(\boldsymbol{\theta})} + \sum_{c=1}^{C} \underbrace{(\nabla_{f_{\boldsymbol{\theta}}} \ell(y, f_{\boldsymbol{\theta}}(\boldsymbol{x})))_c}_{=\boldsymbol{r}} \boldsymbol{H}_{\boldsymbol{\theta}} f_{\boldsymbol{\theta}}(\boldsymbol{x})_c, \tag{2.32}$$

where $\boldsymbol{J}_{\boldsymbol{\theta}} f_{\boldsymbol{\theta}}(\boldsymbol{x}) \in \mathbb{R}^{C \times P}$ is the model's Jacobian matrix, $\boldsymbol{H}_{f_{\boldsymbol{\theta}}} \ell(y, f_{\boldsymbol{\theta}}(\boldsymbol{x})) \in \mathbb{R}^{C \times C}$ the Hessian of the loss function, $\nabla_{f_{\boldsymbol{\theta}}} \ell(y, f_{\boldsymbol{\theta}}(\boldsymbol{x})) \in \mathbb{R}^C$ is the residual $\boldsymbol{r}$, and $\boldsymbol{H}_{\boldsymbol{\theta}} f_{\boldsymbol{\theta}}(\boldsymbol{x})_c \in \mathbb{R}^{P \times P}$ the Hessian of the model. We can observe that we have curvature information of the loss function in the first, and of the model in the second term of the equation. Since our loss functions are convex in $f_{\boldsymbol{\theta}}$, the first term is always p.s.d. and we call it the generalized Gauss-Newton matrix (GGN). Writing the empirical GGN for a dataset $\mathcal{D}$, we have

$$\mathbf{GGN}(\boldsymbol{\theta}) = \sum_{n=1}^{N} \boldsymbol{J}_{\boldsymbol{\theta}} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)^T \boldsymbol{H}_{f_{\boldsymbol{\theta}}} \ell(y_n, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \boldsymbol{J}_{\boldsymbol{\theta}} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n). \tag{2.33}$$

The Hessian $\boldsymbol{H}(\boldsymbol{\theta}_t)$ in Equation (2.31) can be replaced by $\mathbf{GGN}(\boldsymbol{\theta}_t)$ to approximate Newton's method in a non-convex setting.

There are two cases when the GGN coincides with the exact Hessian. First, if we assume that our model $f_{\boldsymbol{\theta}}$ is linear in the parameters $\boldsymbol{\theta}$, the Hessian $\boldsymbol{H}_{\boldsymbol{\theta}} f_{\boldsymbol{\theta}}(\boldsymbol{x})_c = \mathbf{0}_P$ becomes a $P \times P$ zero matrix. Hence, for a linear model the second term in Equation (2.32) vanishes and the Hessian is thus equal to the first term – the GGN. Conversely, using the GGN as a p.s.d. approximation to the Hessian for nonlinear models can be seen as implicitly linearizing $f_{\boldsymbol{\theta}}$ at iterate $\boldsymbol{\theta}_t$ via a first-order Taylor expansion. In deep learning, the GGN is a common approximation of the Hessian (Schraudolph, 2002; Martens, 2010; Botev et al., 2017; Khan et al., 2018; Ritter et al., 2018). In most cases, however, the model is not linearized for predicting even when the GGN was used for optimization or approximate inference; it has been shown to be beneficial to be consistent and predict with the linearized model when using the GGN as a Hessian replacement for a Laplace approximation (Immer et al., 2021b).

Second, the gradient $\nabla_{f_{\boldsymbol{\theta}}} \ell(y, f_{\boldsymbol{\theta}}(\boldsymbol{x}))$ is the residual $\boldsymbol{r}$ for the two losses considered here, as presented in Equation (2.20) and Equation (2.24). Hence, when $p(f_{\boldsymbol{\theta}}(\boldsymbol{x})) = \tilde{\boldsymbol{y}}$ in the classification or $f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{y}$ in the regression case, the second term in Equation (2.32) will vanish. However, this condition does usually also not hold in practice, just as the first one.

2.2.3 Natural Gradient Descent

Natural gradient descent (Amari, 1998, NGD) is typically derived by finding a direction of steepest descent using a specific metric – the Fisher information matrix (FIM), or short, the Fisher. The resulting update uses the FIM as the curvature matrix $\boldsymbol{C}(\boldsymbol{\theta})$ in Equation (2.29). One common motivation for using NGD is its well-known invariance to smooth invertible reparameterizations of the model (Martens, 2014).

[1] In general, the GGN is ambiguous since it depends on the chosen split (Schraudolph, 2002); here, we always refer to the GGN assuming this split.

To introduce the concept of the direction of steepest descent, we first derive the classic gradient descent update (Equation (2.28)) from this alternative perspective. We define a direction of steepest descent as

$$(\delta\boldsymbol{\theta})_* = \lim_{\varepsilon \to 0} \frac{1}{\varepsilon} \operatorname*{argmin}_{\delta\boldsymbol{\theta}:\|\delta\boldsymbol{\theta}\| \leq \varepsilon} \mathcal{L}(\boldsymbol{\theta} + \delta\boldsymbol{\theta}), \tag{2.34}$$

where $\|\cdot\|$ is an unspecified norm on $\mathbb{R}^P$. Intuitively, it represents the direction of a step with fixed infinitesimal length in terms of the chosen norm which results in the smallest possible loss. For the Euclidean norm $\|\cdot\|_2$, we can solve the constrained optimization problem by locally approximating $\mathcal{L}(\boldsymbol{\theta} + \delta\boldsymbol{\theta})$ with a first-order Taylor approximation and using Lagrange multipliers. The result is simply the normalized negative gradient $-\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})/\|\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})\|_2$.

This implies that we implicitly assume a Euclidean metric when optimizing the loss with gradient descent. However, when our loss can be interpreted as a negative log-likelihood, which is the case for our two losses considered here, we are optimizing in the space of probability distributions. It is well known that the Euclidean distance is a bad similarity measure for distributions, e.g. the two mostly overlapping Gaussians $\mathcal{N}(0, 10000)$ and $\mathcal{N}(10, 10000)$ have an Euclidean distance of 10, whereas the mean parameters of the two barely overlapping Gaussians $\mathcal{N}(0, 0.01)$ and $\mathcal{N}(0.1, 0.01)$ have an Euclidean distance of only 0.1.

The key idea of NGD is to replace the Euclidean metric with a better suited alternative for probability distributions – a local approximation to the Kullback-Leibler (KL) divergence between the likelihood and the likelihood after an infinitesimally small step $\delta\boldsymbol{\theta}$. Defining the *model's* joint distribution as $h(\boldsymbol{\theta}) := p_{\boldsymbol{\theta}}(\boldsymbol{x}, y) = p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}))p(\boldsymbol{x})$, we have

$$\begin{aligned} D_{\mathrm{KL}}(h(\boldsymbol{\theta}) \| h(\boldsymbol{\theta} + \delta\boldsymbol{\theta})) &= \mathbb{E}_{\boldsymbol{x},y\sim h(\boldsymbol{\theta})}[\log h(\boldsymbol{\theta}) - \log h(\boldsymbol{\theta} + \delta\boldsymbol{\theta})] \\ &\approx \frac{1}{2}(\delta\boldsymbol{\theta})^T \underbrace{-\mathbb{E}_{\boldsymbol{x},y\sim h(\boldsymbol{\theta})}[\nabla^2_{\boldsymbol{\theta}} \log h(\boldsymbol{\theta})]}_{=:\boldsymbol{F}(\boldsymbol{\theta})} \delta\boldsymbol{\theta}, \end{aligned} \tag{2.35}$$

where we have used a second-order Taylor approximation and simplified it to get to this result which includes the FIM $\boldsymbol{F}(\boldsymbol{\theta})$. Expanding $h(\boldsymbol{\theta}) = p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}))p(\boldsymbol{x})$, we can also express the FIM as

$$\begin{aligned} \boldsymbol{F}(\boldsymbol{\theta}) &= -\mathbb{E}_{\boldsymbol{x},y\sim p_{\boldsymbol{\theta}}(\boldsymbol{x},y)}[\nabla^2_{\boldsymbol{\theta}} \log p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}))] \\ &= \mathbb{E}_{\boldsymbol{x},y\sim p_{\boldsymbol{\theta}}(\boldsymbol{x},y)}[\nabla_{\boldsymbol{\theta}} \log p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}))(\nabla_{\boldsymbol{\theta}} \log p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x})))^T], \end{aligned} \tag{2.36}$$

which implies that the FIM is p.s.d.. In the machine learning literature $p(\boldsymbol{x})$ is usually replaced by the empirical data, i.e.

$$\begin{aligned} \boldsymbol{F}(\boldsymbol{\theta}) &= -\sum_{n=1}^{N} \mathbb{E}_{y\sim p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))}[\nabla^2_{\boldsymbol{\theta}} \log p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))] \\ &= \sum_{n=1}^{N} \mathbb{E}_{y\sim p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))}[\nabla_{\boldsymbol{\theta}} \log p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))(\nabla_{\boldsymbol{\theta}} \log p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)))^T], \end{aligned} \tag{2.37}$$

where we still do not use the empirical labels y_n; this is also what we will refer to as Fisher from now on. In the case of also using the labels y_n, the resulting quantity is called *empirical* Fisher (EF); it is then simply the uncentered covariance of the empirical gradient. While it is commonly

used as a replacement for the Fisher (Graves, 2011; Kingma & Ba, 2015; Chaudhari et al., 2017; Zhang et al., 2018; Khan et al., 2018), they are generally not the same. Kunstner et al. (2019) highlight potential pitfalls of using the Fisher and the EF as interchangeable quantities and offer an alternative hypothetical explanation for the EF's empirical success based on adaption to the gradient noise in stochastic optimization.

While the KL divergence is not a valid metric since it is not symmetric, we can see from Equation (2.35) combined with the insight that the FIM is always p.s.d., that it locally defines the norm $\|\delta\boldsymbol{\theta}\|_{\boldsymbol{F}(\boldsymbol{\theta})} = \sqrt{(\delta\boldsymbol{\theta})^T \boldsymbol{F}(\boldsymbol{\theta})\delta\boldsymbol{\theta}}$. Following the initial motivation to use a more meaningful metric to find a direction of steepest descent, we can now use this norm for our objective in Equation (2.34) and according to Ollivier et al. (2017) have

$$\lim_{\varepsilon \to 0} \frac{1}{\varepsilon} \operatorname*{argmin}_{\delta\boldsymbol{\theta}:\|\delta\boldsymbol{\theta}\|_{\boldsymbol{F}(\boldsymbol{\theta})} \leq \varepsilon} \mathcal{L}(\boldsymbol{\theta} + \delta\boldsymbol{\theta}) = -\frac{\boldsymbol{F}(\boldsymbol{\theta})^{-1}\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})}{\|\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})\|_{\boldsymbol{F}(\boldsymbol{\theta})^{-1}}}; \tag{2.38}$$

note that $\|\delta\boldsymbol{\theta}\|_{\boldsymbol{F}(\boldsymbol{\theta})} \leq \varepsilon \iff \frac{1}{2}\|\delta\boldsymbol{\theta}\|^2_{\boldsymbol{F}(\boldsymbol{\theta})} \leq \varepsilon^2/2$ (with $\varepsilon \geq 0$) and from Equation (2.35) we know that $\frac{1}{2}\|\delta\boldsymbol{\theta}\|^2_{\boldsymbol{F}(\boldsymbol{\theta})} \approx D_{\mathrm{KL}}(h(\boldsymbol{\theta})\|h(\boldsymbol{\theta} + \delta\boldsymbol{\theta}))$. The resulting update sets the curvature matrix $\boldsymbol{C}(\boldsymbol{\theta}_t) = \boldsymbol{F}(\boldsymbol{\theta}_t)$ in Equation (2.29) and is called natural gradient descent.

Even though NGD is motivated by the concept of finding a direction of steepest descent in distribution space, we can connect it to Newton's method through the Fisher's relationship to the GGN. The Fisher and the GGN coincide for both losses we consider here and more generally, for all likelihoods of the exponential family with natural parameterization (Wang, 2010; Martens, 2014). To see this, we rewrite the Fisher as

$$\begin{aligned}
\boldsymbol{F}(\boldsymbol{\theta}) &= \sum_{n=1}^{N} \mathbb{E}_{y \sim p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))}\left[\nabla_{\boldsymbol{\theta}} \log p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))(\nabla_{\boldsymbol{\theta}} \log p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)))^T\right] \\
&= \sum_{n=1}^{N} \boldsymbol{J}_{\boldsymbol{\theta}} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)^T \mathbb{E}_{y \sim p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))}\left[\nabla_{f_{\boldsymbol{\theta}}} \log p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \nabla_{f_{\boldsymbol{\theta}}} \log p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))^T\right] \boldsymbol{J}_{\boldsymbol{\theta}} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n) \\
&= \sum_{n=1}^{N} \boldsymbol{J}_{\boldsymbol{\theta}} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)^T \mathbb{E}_{y \sim p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))}\left[-\nabla^2_{f_{\boldsymbol{\theta}}} \log p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))\right] \boldsymbol{J}_{\boldsymbol{\theta}} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n),
\end{aligned} \tag{2.39}$$

and by substituting $-\nabla^2_{f_{\boldsymbol{\theta}}} \log p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))$ with $\boldsymbol{H}_{f_{\boldsymbol{\theta}}}(\ell(y, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)))$, we have

$$F(\boldsymbol{\theta}) = \sum_{n=1}^{N} \boldsymbol{J}_{\boldsymbol{\theta}} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)^T \mathbb{E}_{y \sim p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))}[\boldsymbol{H}_{f_{\boldsymbol{\theta}}}(\ell(y, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)))] \boldsymbol{J}_{\boldsymbol{\theta}} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n). \tag{2.40}$$

Now the similarity to the GGN becomes apparent since we have the same expression as in Equation (2.33), but with an expectation over $y \sim p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))$. As we have seen in Section 2.1.3, for the cross-entropy and the mean-square error loss $\boldsymbol{H}_{f_{\boldsymbol{\theta}}}(\ell(y, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)))$ does not depend on the labels y at all. Hence, in both cases the Fisher and the GGN are identical.

2.2.4 K-FAC

Kronecker-factored Approximate Curvature (Heskes, 2000; Martens & Grosse, 2015, K-FAC) was proposed as an efficient approximation to a neural network's Fisher information matrix (Section 2.2.3). First, in all of this work, we focus on a layer-wise K-FAC approximation of the Fisher, i.e. it is approximated by a block-diagonal matrix

$$\boldsymbol{F}(\boldsymbol{\theta}) \approx \operatorname{diag}(\boldsymbol{F}(\boldsymbol{\theta}_1), \dots, \boldsymbol{F}(\boldsymbol{\theta}_\ell), \dots, \boldsymbol{F}(\boldsymbol{\theta}_L)) \in \mathbb{R}^{P \times P}, \tag{2.41}$$

where $\boldsymbol{F}(\boldsymbol{\theta}_\ell) \in \mathbb{R}^{P_\ell \times P_\ell}$ and $\operatorname{diag}(\cdot, \dots, \cdot)$ build a block-diagonal matrix with the input matrices as blocks.

To derive K-FAC, we first note that the pre-activation for layer ℓ and the nth data point $\boldsymbol{x}_n$ can be expressed as $\boldsymbol{s}_{\ell,n} = \boldsymbol{W}_\ell \boldsymbol{a}_{\ell,n}$, with $\boldsymbol{W}_\ell \in \mathbb{R}^{P_{\ell,\text{out}} \times P_{\ell,\text{in}}}$ and $\boldsymbol{a}_{\ell,n} \in \mathbb{R}^{P_{\ell,\text{in}}}$, the input to the ℓth layer (or equivalently, the activation of the $\ell - 1$th layer). We have omitted an explicit bias parameter $\boldsymbol{b}_\ell$, since it can always be subsumed in $\boldsymbol{W}_\ell$. Hence, by applying the chain rule, the gradient of the loss w.r.t. the weights of the ℓth layer can be written as $\nabla_{\boldsymbol{W}_\ell} \mathcal{L}(y, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) = \nabla_{\boldsymbol{s}_\ell} \mathcal{L}(y, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \boldsymbol{a}_{\ell,n}^T =: \boldsymbol{g}_{\ell,n} \boldsymbol{a}_{\ell,n}^T \in \mathbb{R}^{P_{\ell,\text{out}} \times P_{\ell,\text{in}}}$.

Using these insights, K-FAC then replaces the sum of expectations over Kronecker products with a Kronecker product of two sums of expectations, i.e.

$$\begin{aligned}
\boldsymbol{F}(\boldsymbol{\theta}_\ell) &= \sum_{n=1}^{N} \mathbb{E}_{y \sim p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))} [\operatorname{vec}(\nabla_{\boldsymbol{W}_\ell} \mathcal{L}(y, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))) \operatorname{vec}(\nabla_{\boldsymbol{W}_\ell} \mathcal{L}(y, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)))^T] && \text{(2.42a)} \\
&= \sum_{n=1}^{N} \mathbb{E}_{y \sim p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))} [\operatorname{vec}(\boldsymbol{g}_{\ell,n} \boldsymbol{a}_{\ell,n}^T) \operatorname{vec}(\boldsymbol{g}_{\ell,n} \boldsymbol{a}_{\ell,n}^T)^T] && \text{(2.42b)} \\
&= \sum_{n=1}^{N} \mathbb{E}_{y \sim p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))} [(\boldsymbol{a}_{\ell,n} \otimes \boldsymbol{g}_{\ell,n})(\boldsymbol{a}_{\ell,n}^T \otimes \boldsymbol{g}_{\ell,n}^T)] && \text{(2.42c)} \\
&= \sum_{n=1}^{N} \mathbb{E}_{y \sim p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))} [\boldsymbol{a}_{\ell,n} \boldsymbol{a}_{\ell,n}^T \otimes \boldsymbol{g}_{\ell,n} \boldsymbol{g}_{\ell,n}^T] && \text{(2.42d)} \\
&\approx \underbrace{\left[\frac{1}{N} \sum_{n=1}^{N} \boldsymbol{a}_{\ell,n} \boldsymbol{a}_{\ell,n}^T \right]}_{=:\boldsymbol{A}_\ell} \otimes \underbrace{\left[\sum_{n=1}^{N} \mathbb{E}_{y \sim p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))} [\boldsymbol{g}_{\ell,n} \boldsymbol{g}_{\ell,n}^T] \right]}_{=:\boldsymbol{G}_\ell}, && \text{(2.42e)}
\end{aligned}$$

where $\boldsymbol{A}_\ell \in \mathbb{R}^{P_{\ell,\text{in}} \times P_{\ell,\text{in}}}$ and $\boldsymbol{G}_\ell \in \mathbb{R}^{P_{\ell,\text{out}} \times P_{\ell,\text{out}}}$. For this derivation, we have used three convenient properties of the Kronecker product (using matrices $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D}$ with appropriate dimensions): $\operatorname{vec}(\boldsymbol{ABC}) = (\boldsymbol{C}^T \otimes \boldsymbol{A}) \operatorname{vec}(\boldsymbol{B})$ and $(\boldsymbol{A} \otimes \boldsymbol{B})^T = \boldsymbol{A}^T \otimes \boldsymbol{B}^T$ for Equation (2.42c), and $(\boldsymbol{A} \otimes \boldsymbol{B})(\boldsymbol{C} \otimes \boldsymbol{D}) = \boldsymbol{AC} \otimes \boldsymbol{BD}$ for Equation (2.42d).

We can see that the approximation is exact in the trivial case of a single data point, i.e. $N = 1$. Moreover, it is also exact in the case of a single linear layer or a deep linear network and a Gaussian likelihood (Bernacchia et al., 2018).

K-FAC is more efficient than a naive block-wise approximation because we only have to store and invert two Kronecker factors instead of a larger dense matrix for each layer, which reduces the memory complexity from $\mathcal{O}(P_{\ell,\text{in}}^2 P_{\ell,\text{out}}^2)$ to $\mathcal{O}(P_{\ell,\text{in}}^2 + P_{\ell,\text{out}}^2)$ and the computational com-

plexity of the preconditioning of the gradient with the approximate Fisher from $\mathcal{O}(P_{\ell,\text{in}}^3 P_{\ell,\text{out}}^3)$ to $\mathcal{O}(P_{\ell,\text{in}}^3 + P_{\ell,\text{out}}^3)$, since

$$\begin{aligned} \boldsymbol{F}(\boldsymbol{\theta}_\ell)^{-1}\boldsymbol{g}(\boldsymbol{\theta}_\ell) &\approx (\boldsymbol{A}_\ell \otimes \boldsymbol{G}_\ell)^{-1}\boldsymbol{g}(\boldsymbol{\theta}_\ell) \\ &= \text{vec}\big(\boldsymbol{G}_\ell^{-1}\nabla_{\boldsymbol{W}_\ell}\mathcal{L}(y, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))\boldsymbol{A}_\ell^{-1}\big) \end{aligned} \tag{2.43}$$

with $\boldsymbol{g}(\boldsymbol{\theta}_\ell) = \text{vec}(\nabla_{\boldsymbol{W}_\ell}\mathcal{L}(y, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)))$ and the property $(\boldsymbol{A} \otimes \boldsymbol{B})^{-1} = \boldsymbol{A}^{-1} \otimes \boldsymbol{B}^{-1}$.

Alternatively, we can derive K-FAC for the GGN (Botev et al., 2017), which will recover the same result as for the Fisher in Equation (2.42) for the losses we consider here, as we have learned in Section 2.2.3. We define $\boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{x}_n) := \boldsymbol{J}_{\boldsymbol{\theta}_\ell} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n) = \boldsymbol{J}_{\boldsymbol{s}_{\ell,n}} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n) \boldsymbol{J}_{\boldsymbol{\theta}_\ell} \boldsymbol{s}_{\ell,n} \in \mathbb{R}^{C \times P_\ell}$ as the Jacobian of the model outputs w.r.t. the parameters of the ℓth layer and $\boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) := \boldsymbol{H}_{f_{\boldsymbol{\theta}}} \mathcal{L}(y_n, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \in \mathbb{R}^{C \times C}$ as the Hessian of the loss w.r.t. the model outputs. Now we can write $\boldsymbol{s}_{\ell,n} = \boldsymbol{W}_\ell \boldsymbol{a}_{\ell,n} = (\boldsymbol{a}_{\ell,n}^T \otimes \mathbf{I}_{P_{\ell,\text{out}}})\text{vec}(\boldsymbol{W}_\ell)$ and with this we have $\boldsymbol{J}_{\boldsymbol{\theta}_\ell} \boldsymbol{s}_{\ell,n} = \boldsymbol{a}_{\ell,n}^T \otimes \mathbf{I}_{P_{\ell,\text{out}}}$. Additionally, by defining $\boldsymbol{b}_{\ell,n} := \boldsymbol{J}_{\boldsymbol{s}_\ell} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)^T \in \mathbb{R}^{P_{\ell,\text{out}} \times C}$ as the transposed Jacobian of the model outputs w.r.t to the pre-activations of the ℓth layer, we note that $\boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{x}_n)^T = (\boldsymbol{a}_{\ell,n}^T \otimes \mathbf{I}_{P_{\ell,\text{out}}})^T \boldsymbol{b}_{\ell,n} = \boldsymbol{a}_{\ell,n} \otimes \boldsymbol{b}_{\ell,n}$.

Replacing the (transposed) Jacobians in the definition of the GGN by this expression, we have

$$\begin{aligned} \mathbf{GGN}(\boldsymbol{\theta}_\ell) &= \sum_{n=1}^N \boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{x}_n)^T \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{x}_n) \\ &= \sum_{n=1}^N (\boldsymbol{a}_{\ell,n} \otimes \boldsymbol{b}_{\ell,n}) \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) (\boldsymbol{a}_{\ell,n} \otimes \boldsymbol{b}_{\ell,n})^T \\ &= \sum_{n=1}^N (\boldsymbol{a}_{\ell,n} \boldsymbol{a}_{\ell,n}^T) \otimes (\boldsymbol{b}_{\ell,n} \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \boldsymbol{b}_{\ell,n}^T) \\ &\approx \underbrace{\left[\frac{1}{N} \sum_{n=1}^N \boldsymbol{a}_{\ell,n} \boldsymbol{a}_{\ell,n}^T\right]}_{=:\boldsymbol{A}_\ell} \otimes \underbrace{\left[\sum_{n=1}^N \boldsymbol{b}_{\ell,n} \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \boldsymbol{b}_{\ell,n}^T\right]}_{=:\boldsymbol{B}_\ell}. \end{aligned} \tag{2.44}$$

This derivation is a bit more convenient for our purposes, as it does not require us to keep track of the expectation over the labels y, while still being equivalent to the Fisher for the losses we consider here. Moreover, it will be useful to have the Jacobians $\boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{x}_n)$ separate from the loss for our derivations in Chapter 3; therefore, we will only explicitly write our results for the GGN.

Finally, K-FAC has also been extended to convolution (Grosse & Martens, 2016) and recurrent neural network layers (Martens et al., 2018).

2.3 Benchmarks

To evaluate and compare optimization methods such as the ones presented in Section 2.2, we need benchmarks. Without them, it is impossible to determine if one optimization algorithm provides any advantage over others; in fact, the lack of a consistent and large-scale benchmark seems to be a problem of the recent deep learning optimization literature (Schmidt et al., 2021). One very recent benchmark which tries to address this issue is the MLCommons Al-

goPerf benchmark. According to our focus, we are using a subset of its workloads which uses models with linear weight-sharing layers for the examples and experiments in this work.

2.3.1 MLCommons AlgoPerf: An Algorithmic Efficiency Benchmark

The Algorithms Working Group of MLCommons wants to "create a set of rigorous and relevant benchmarks to measure neural network training speedups due to algorithmic improvements" (MLCommons, 2022). The benchmark, called *AlgoPerf*, consists of multiple workloads, where one workload is defined as a loss function, a dataset, and a neural network model architecture. A submission to the benchmark is a training algorithm together with a hyperparameter search space. Performance is measured in terms of wall-clock time to reach a fixed target value of a validation metric on fixed hardware. One submission has to run on all workloads and the final score will contain the timing results on all workloads.

Since the benchmark is designed to provide meaningful comparisons between optimization algorithms across contemporary and large-scale deep learning settings, running an algorithm using K-FAC on the full benchmark would be a step towards increasing K-FAC's potential practical relevance. In multiple workloads models with linear weight-sharing layers, i.e. Transformers and GNNs, are used. Hence, extending K-FAC to these types of models is a necessary step towards a valid K-FAC submission to the benchmark. Since we focus on this extension of K-FAC here, we only consider the following three workloads.

Vision Transformer on ImageNet. This workload also uses the cross-entropy loss, a Vision Transformer architecture, and the LSVRC-2012 ImageNet (short: ImageNet) image dataset (Russakovsky et al., 2015). The goal is to classify images into one of 1000 classes. There are about 1.3 million training, 50k validation, and 10k test examples.

While we do not provide empirical results for the next two workloads, we provide all necessary details to also conduct experiments on them.

Transformer on WMT. This workload consists of a the cross-entropy loss (c.f. Section 2.1.3), a Transformer model architecture (c.f. Section 2.1.1.1), and the WMT14 (Bojar et al., 2014) and WMT17 (Bojar et al., 2017) de-en translation datasets. The task is to translate German sentences into English. The datasets contain data from multiple sources, like the Europarl corpus. The WMT17 dataset is used for training and the WMT14 dataset for validation and testing. There are about slightly below six million examples for training and about 3000 examples each for validation and testing. The vocabulary size is 32k.

GNN on OGBG. The workload consists of a (binary) cross-entropy loss, the Graph network instance described in Section 2.1.1.2, and the ogbg-molpcba molecular property prediction dataset (Hu et al., 2020). Each molecule is represented as a graph, where atoms are nodes and edges are chemical bonds. The task is to predict whether or not a molecule has certain chemical properties; there are 128 different properties. For training, we have about 350k examples and almost 44k for validation and testing.

Chapter 3
K-FAC for Linear Weight-Sharing Layers

As introduced in Section 2.1.1, many contemporary neural network architectures, such as Transformers and GNNs, use linear weight-sharing layers where the weights are shared over an additional input dimension of size R. There can be arbitrarily many dimensions like this, but, for simplicity, we will focus on the case of a single additional dimension.

In this chapter, we are trying to answer the question of how K-FAC can be applied to this layer type. We identify two different scenarios of how linear weight-sharing layers can be used within a model and show that each motivates a slightly different K-FAC approximation, which we call *K-FAC-expand* and *K-FAC-reduce*. However, in practice, both approximations can be applied to each of the two settings. Prototypical examples for the two scenarios are a Transformer for language translation and a Vision Transformer for image classification. We also concretely discuss the application of the approximation to a simplification of the attention mechanism within these Transformer architectures. Moreover, the two base cases also provide a recipe for deciding which K-FAC approximation to use for model architectures that are not explicitly considered here. As an example, we will consider an instance of the second case, the reduce setting, a GNN as introduced by Battaglia et al. (2018), which is a bit more complex than the corresponding base case, mostly due to the representation of the graph inputs and implementation considerations.

3.1 The Expand and the Reduce Setting

When we know there exists at least one weight-sharing layer in a model and the final loss is a scalar value, we can deduce that there has to exist an aggregation function z which at some point reduces the weight-sharing dimension of size R. We propose to classify linear layers within a network that contains weight-sharing layers based on the point where the aggregation function is applied. We can distinguish three different aggregation points, which will define the setting for the ℓth layer:

(i) *Before* the ℓth layer, i.e. $\boldsymbol{A}_{\ell,n} \in \mathbb{R}^{R \times P_{\ell,\text{in}}}$ is reduced to $\tilde{\boldsymbol{a}}_{\ell,n} \in \mathbb{R}^{P_{\ell,\text{in}}}$ before being multiplied with the weight matrix of the layer $\boldsymbol{W}_\ell$. $\rightarrow$ The weight matrix is applied like in a regular linear layer, i.e. we are in the setting of a regular linear layer and no particular considerations are necessary when using K-FAC.

(ii) *After the per-example loss*, i.e. there will be $N \cdot R$ outputs of the model and $N \cdot R$ labels. The per-example loss is applied to each of the $N \cdot R$ output-label pairs and summed. $\rightarrow$ This is what we call the *expand* case or setting, as the loss for each of the N data points is expanded with R terms. The reduction function is always a simple sum in this case (at least if we assume our loss corresponds to a valid density, which we do here).

(iii) *In between* the pre-activation of the ℓth layer $\boldsymbol{S}_{\ell,n} \in \mathbb{R}^{R \times P_{\ell,\text{out}}}$ and the final aggregation over the per-example losses. In our case, this implies that the aggregation happens before the model output $f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)$, since we only consider the CE and MSE losses which do not include any additional aggregation of the model outputs. In this case, the labels y_n will not have an additional dimension. $\rightarrow$ This is what we call the *reduce* case or setting, as all weight-sharing dimensions have been reduced before the final aggregation over the per-example losses.

We can see that depending on the setting, the loss will have a different number of terms, i.e. per-example losses. Hence, if we assume a single weight-sharing dimension which will be reduced once, the form of the loss function determines which setting applies to all linear weight-sharing layers within a model. So for our purposes, we can identify the setting we are considering simply by looking at the form of the loss.

There might exist model architectures that reduce the weight-sharing dimension of size R and then recreate it during the forward pass, e.g. by stacking the outputs of two linear layers with the same inputs, in which case we would have to different settings for different layers. Additionally, it is possible that we have multiple weight-sharing dimensions and they are aggregated at different points, leading to different settings for different weight-sharing dimensions. However, we are not aware of such architectures and problems used in practice. In any case, the two base cases presented here still offer a framework for reasoning about these more complex settings.

Starting with the form of the loss, we present the two settings and motivate one approximation each. We explicitly state the derivations for the GGN, but they follow analogously for the Fisher, see Section 2.2.4.

3.1.1 The Expand Setting and K-FAC-expand

The first base setting can be identified by a loss with $N \cdot R$ terms, which corresponds to assuming $N \cdot R$ i.i.d. examples,

The Expand Setting

$$\mathcal{L}_{\text{expand}}(f_{\boldsymbol{\theta}}, \mathcal{D}) := -\sum_{n=1}^{N} \sum_{r=1}^{R} \log p(y_{n,r} | f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r), \tag{3.1}$$

where $f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r$ is the rth row of the model output $f_{\boldsymbol{\theta}}(\boldsymbol{x}_n) \in \mathbb{R}^{R \times C}$; the target of each data point is now a vector $\boldsymbol{y}_n \in \mathbb{R}^R$. A typical example of this type of loss function is language translation, e.g. the WMT workload presented in Section 2.3, where N is the dataset size and R is the sequence length.

Note that we are not assuming our inputs $\boldsymbol{x}_n$ to have an additional weight-sharing dimension, since we only require that the input to the ℓth layer to have this additional dimension, i.e. $\boldsymbol{A}_{\ell,n} \in \mathbb{R}^{R \times D}$. This does not exclude the case where $\boldsymbol{x}_n$ already has this weight-sharing dimension, e.g. sentences in translation tasks.

We can express the Jacobian of the rth row of the model output $f_{\boldsymbol{\theta}}(\boldsymbol{x}_n) \in \mathbb{R}^{R \times C}$ w.r.t. the parameters $\boldsymbol{\theta}_\ell$ as

$$(\boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{x}_n)_r)_{ij} = \sum_{m=1}^{R} \sum_{p=1}^{P_{\ell,\text{out}}} \frac{\partial f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_{ri}}{\partial \boldsymbol{S}_{\ell,n,mp}} \frac{\partial \boldsymbol{S}_{\ell,n,mp}}{\partial \boldsymbol{\theta}_{\ell,j}} \tag{3.2}$$

or in matrix form

$$\boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{x}_n)_r = \sum_{m=1}^{R} \boldsymbol{J}_{\boldsymbol{s}_{\ell,n,m}} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r \boldsymbol{J}_{\boldsymbol{\theta}_\ell} \boldsymbol{s}_{\ell,n,m}. \tag{3.3}$$

Since the weights $\boldsymbol{\theta}_\ell$ are shared across the weight-sharing dimension of size R, we can write the rth row of $\boldsymbol{S}_{\ell,n}$ as $\boldsymbol{s}_{\ell,n,r} = \boldsymbol{W}_\ell \boldsymbol{a}_{\ell,n,r}$ and we have $\boldsymbol{J}_{\boldsymbol{\theta}_\ell} \boldsymbol{s}_{\ell,n,r} = \boldsymbol{a}_{\ell,n,r}^T \otimes \mathbf{I}_{P_{\ell,\text{out}}}$, as for regular K-FAC (c.f. Section 2.2.4). We denote $\boldsymbol{b}_{\ell,n,r,k} := \boldsymbol{J}_{\boldsymbol{s}_{\ell,n,k}} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r^T$. Hence, we have

$$\begin{aligned} \boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{x}_n)_r^T &= \left(\sum_{m=1}^{R} \boldsymbol{b}_{\ell,n,r,m}^T (\boldsymbol{a}_{\ell,n,m}^T \otimes \mathbf{I}_{P_{\ell,\text{out}}}) \right)^T \\ &= \sum_{m=1}^{R} \boldsymbol{a}_{\ell,n,m} \otimes \boldsymbol{b}_{\ell,n,r,m}. \end{aligned} \tag{3.4}$$

On a high level, applying K-FAC to a model trained with this type of loss just requires treating the problem as if we had $N \cdot R$ *independent* examples and derive the approximation in the exact same way as we would with N examples (c.f. Section 2.2.4),

$$\begin{aligned} \mathbf{GGN}(\boldsymbol{\theta}_\ell) &= \sum_{n=1}^{N} \sum_{r=1}^{R} \boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{x}_n)_r^T \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r) \boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{x}_n)_r \\ &= \sum_{n=1}^{N} \sum_{r=1}^{R} \left(\sum_{m=1}^{R} \boldsymbol{a}_{\ell,n,m} \otimes \boldsymbol{b}_{\ell,n,r,m} \right) \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r) \left(\sum_{m=1}^{R} \boldsymbol{a}_{\ell,n,m}^T \otimes \boldsymbol{b}_{\ell,n,r,m}^T \right). \end{aligned} \tag{3.5}$$

However, we cannot directly write each of the $N \cdot R$ loss terms as a Kronecker product without any approximation. One approach could be to use a K-FAC style approximation to the Jacobians $\boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{x}_n)_r$, but then we would have to be able to access $\sum_{m=1}^{R} \boldsymbol{b}_{\ell,n,r,m}$. Moreover, this would not even be exact in the simple settings we consider later. In practice, we only have access to $\sum_{r=1}^{R} \boldsymbol{b}_{\ell,n,r,m}$ (c.f. Section 3.4). In fact, this is what has been used in this setting, in the context of Transformers for natural language processing (Zhang et al., 2019; Pauloski et al., 2021; Osawa et al., 2022). However, the authors do not discuss this extension of K-FAC at all (Pauloski et al., 2021; Osawa et al., 2022) or do not try to derive or justify it (Zhang et al., 2019). What is actually implemented in practice is[1]

$$\begin{aligned} \mathbf{GGN}(\boldsymbol{\theta}_\ell) &\approx \sum_{n=1}^{N} \sum_{m=1}^{R} \left(\boldsymbol{a}_{\ell,n,m} \otimes \underbrace{\left[\sum_{r=1}^{R} \boldsymbol{b}_{\ell,n,r,m} \right]}_{=: \hat{\boldsymbol{b}}_{\ell,n,m}} \right) \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_m) \left(\boldsymbol{a}_{\ell,n,m}^T \otimes \sum_{r=1}^{R} \boldsymbol{b}_{\ell,n,r,m}^T \right) \\ &= \sum_{n=1}^{N} \sum_{r=1}^{R} (\boldsymbol{a}_{\ell,n,r} \boldsymbol{a}_{\ell,n,r}^T) \otimes \left(\hat{\boldsymbol{b}}_{\ell,n,r} \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r) \hat{\boldsymbol{b}}_{\ell,n,r}^T \right) \end{aligned} \tag{3.6}$$

Where we have replaced the exact expression for each of the $N \cdot R$ terms with something else, which allows us to express each term as a Kronecker product. Consequently, we can apply the

[1] Although previous work seems to ignore the scaling by $1/R$ in Equation (3.7).

regular K-FAC approximation over $N \cdot R$ terms instead of just N terms as usual. We call this approximation *K-FAC-expand*:

K-FAC-expand

$$\mathbf{G}\hat{\mathbf{G}}\mathbf{N}_{\boldsymbol{\theta}_\ell}^{\text{expand}} := \underbrace{\left[\frac{1}{NR}\sum_{n=1}^{N}\sum_{r=1}^{R}\boldsymbol{a}_{\ell,n,r}\boldsymbol{a}_{\ell,n,r}^T\right]}_{=\boldsymbol{A}_\ell} \otimes \underbrace{\left[\sum_{n=1}^{N}\sum_{r=1}^{R}\hat{\boldsymbol{b}}_{\ell,n,r}\boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r)\hat{\boldsymbol{b}}_{\ell,n,r}^T\right]}_{=\boldsymbol{B}_\ell}. \tag{3.7}$$

There is one simple case where the exact expression in Equation (3.5) is identical to the approximation in Equation (3.6). When $\boldsymbol{b}_{\ell,n,r,m} = \mathbf{0}$ for all $r \neq m$, both expression are equivalent to

$$\sum_{n=1}^{N}\sum_{r=1}^{R}(\boldsymbol{a}_{\ell,n,r} \otimes \boldsymbol{b}_{\ell,n,r,r})\boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r)(\boldsymbol{a}_{\ell,n,r}^T \otimes \boldsymbol{b}_{\ell,n,r,r}^T). \tag{3.8}$$

With other words, when $f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r$ is independent from all pre-activations $\boldsymbol{s}_{\ell,n,m}$ with $m \neq r$ the two expressions coincide. This is the case for a network that simply stacks multiple linear weight-sharing layers; however, it does not even hold for simplistic Transformer models since the dot-product attention mechanism (Section 2.1.1.1) in Transformers directly correlates elements across the weight-sharing dimension; we discuss this in more detail in Section 3.2. Besides the connection of the two expressions in this simple case, there seems to be no obvious motivation or justification for the approximation, besides that is easy to implement within current implementations of K-FAC (c.f. Section 3.4). However, this case applies to networks that simply stack linear weight-sharing layers, we can show that the approximation in Equation (3.7) is exact in the same simple cases as regular K-FAC.

For a typical neural network with nonlinear activation functions, K-FAC is only an approximation. However, for regular individual linear layers and deep linear networks, K-FAC is known to be exact assuming a Gaussian likelihood (Bernacchia et al., 2018). While this holds for the full GGN/Fisher, we only focus on the block-diagonal case here. To motivate K-FAC-expand, we want to show that similar statements hold for a single linear weight-sharing layer and deep linear networks with weight-sharing in the expand setting. First, we state a simple condition for which the approximation is indeed exact; this line of reasoning could also be applied to K-FAC for regular linear layers, since only the effective number of data points changes from $N \cdot R$ to N. Note, that we could also state more trivial sufficient conditions for the exactness of the approximation, i.e. $N = R = 1$ and when all inputs to a layer $\boldsymbol{a}_{\ell,n,r}$ are the same for all $n \in \{1, \dots, N\}$ and $r \in \{1, \dots, R\}$. We do not state these types of conditions explicitly from now on.

Lemma 3.1 (Sufficient condition for exactness of K-FAC-expand in the expand setting). *Let $\boldsymbol{C}_\ell \in \mathbb{R}^{P_{\ell,\text{out}} \times P_{\ell,\text{out}}}$ be a constant matrix for layer ℓ. If $\boldsymbol{b}_{\ell,n,r,m} = \mathbf{0}$ for all $r \neq m$ and $\hat{\boldsymbol{b}}_{\ell,n,r}\boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r)\hat{\boldsymbol{b}}_{\ell,n,r}^T = \boldsymbol{C}_\ell$ for all $n \in \{1, \dots, N\}$ and $r \in \{1, \dots, R\}$, then the K-FAC approximation in Equation (3.7) is equal to the exact GGN/Fisher of the ℓth layer in the expand setting.*

Proof. As mentioned before, when $\boldsymbol{b}_{\ell,n,r,m} = \mathbf{0}$ for all $r \neq m$ the last line of Equation (3.5) and the first line of Equation (3.6) both simplify to Equation (3.8), i.e. $\hat{\boldsymbol{b}}_{\ell,n,r} = \boldsymbol{b}_{\ell,n,r,r}$. Hence, we can directly show that the second and third approximation in Equation (3.6) equal the exact expression for the GGN of layer ℓ from there. We have

$$
\begin{aligned}
&\left(\frac{1}{NR}\sum_{n=1}^{N}\sum_{r=1}^{R}\boldsymbol{a}_{\ell,n,r}\boldsymbol{a}_{\ell,n,r}^T\right) \otimes \left(\sum_{n=1}^{N}\sum_{r=1}^{R}\boldsymbol{b}_{\ell,n,r,r}\boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r)\boldsymbol{b}_{\ell,n,r,r}^T\right) \\
=&\left(\frac{1}{NR}\sum_{n=1}^{N}\sum_{r=1}^{R}\boldsymbol{a}_{\ell,n,r}\boldsymbol{a}_{\ell,n,r}^T\right) \otimes (NR\boldsymbol{C}_\ell) \\
=&\left(\sum_{n=1}^{N}\sum_{r=1}^{R}\boldsymbol{a}_{\ell,n,r}\boldsymbol{a}_{\ell,n,r}^T\right) \otimes \boldsymbol{C}_\ell \\
=&\sum_{n=1}^{N}\sum_{r=1}^{R}\left(\boldsymbol{a}_{\ell,n,r}\boldsymbol{a}_{\ell,n,r}^T\right) \otimes \left(\boldsymbol{b}_{\ell,n,r,r}\boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r)\boldsymbol{b}_{\ell,n,r,r}^T\right),
\end{aligned}
\tag{3.9}
$$

where we have used the assumption that $\hat{\boldsymbol{b}}_{\ell,n,r}\boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r)\hat{\boldsymbol{b}}_{\ell,n,r}^T = \boldsymbol{C}_\ell$ is the same for all $n \in \{1,\dots,N\}$ and $r \in \{1,\dots,R\}$. □

Leveraging this simple insight, we can provide an example of a single layer where the assumptions of Lemma 3.1 are fulfilled.

Proposition 3.2 (Exactness of K-FAC-expand for single layer and Gaussian likelihood in the expand setting). *For a single linear weight-sharing layer and a Gaussian likelihood with p.d. covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{C\times C}$, K-FAC-expand is exact in the expand setting.*

Proof. We can write $f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r = \boldsymbol{W}_\ell \boldsymbol{x}_{n,r}$ and hence $\boldsymbol{b}_{\ell,n,r,m} = \mathbf{0}$ for $r \neq m$. Moreover, we have $\boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r) = \boldsymbol{\Sigma}^{-1}$ and $\hat{\boldsymbol{b}}_{\ell,n,r} = \mathbf{I}_C$ ($P_{\ell,\text{out}} = C$ for a single layer). Hence,

$$
\boldsymbol{b}_{\ell,n,r}\boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r)\boldsymbol{b}_{\ell,n,r}^T = \mathbf{I}_C\boldsymbol{\Sigma}^{-1}\mathbf{I}_C = \boldsymbol{\Sigma}^{-1}
$$

for all $n \in \{1,\dots,N\}$ and $r \in \{1,\dots,R\}$. Therefore, the desired result follows from Lemma 3.1. □

A natural question might be if the same result also holds for *deep* linear networks. A deep linear network is here defined as a model of the form

$$
f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{W}_L \dots \boldsymbol{W}_\ell \dots \boldsymbol{W}_1 \boldsymbol{x} = \boldsymbol{W}\boldsymbol{x}, \tag{3.10}
$$

where $\boldsymbol{x} \in \mathbb{R}^D$ and $\boldsymbol{W}_L \in \mathbb{R}^{C\times P_{L,\text{in}}}$, $\boldsymbol{W}_\ell \in \mathbb{R}^{P_{\ell,\text{out}}\times P_{\ell,\text{in}}}$ (with $P_{\ell,\text{in}} = P_{\ell-1,\text{out}}$), and $\boldsymbol{W}_1 \in \mathbb{R}^{P_{1,\text{out}}\times D}$. While it might seem nonsensical to decompose a single weight matrix $\boldsymbol{W}$ into L separate ones, it creates nonlinear training dynamics of gradient descent training algorithms, while still having analytical solutions (Saxe et al., 2014; Bernacchia et al., 2018). We adopt the notation of Bernacchia et al. (2018) and define

$$
\boldsymbol{W}_\ell^a := \boldsymbol{W}_L \dots \boldsymbol{W}_{\ell+1} \tag{3.11}
$$

as the product of the weight matrices *ahead* of $\boldsymbol{W}_\ell$ and

$$\boldsymbol{W}_\ell^b := \boldsymbol{W}_{\ell-1} \dots \boldsymbol{W}_1 \tag{3.12}$$

as the product of the weight matrices *behind* of $\boldsymbol{W}_\ell$. Hence, we can write $f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{W}_\ell^a \boldsymbol{W}_\ell \boldsymbol{W}_\ell^b x$. Note, that now

$$\boldsymbol{a}_{\ell,n,r} = \boldsymbol{W}_\ell^b \boldsymbol{x}_{n,r} \in \mathbb{R}^{P_{\ell,\text{in}}} \tag{3.13}$$

and

$$\hat{\boldsymbol{b}}_{\ell,n,r} = \boldsymbol{W}_\ell^{a^T} \in \mathbb{R}^{P_{\ell,\text{out}} \times C}. \tag{3.14}$$

Using these insights, we can now easily state the result for deep linear networks.

Proposition 3.3 (Exactness of K-FAC-expand for deep linear network and Gaussian likelihood in the expand setting). *For layer ℓ of a deep linear network defined in Equation (3.10) and a Gaussian likelihood with p.d. covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{C \times C}$, K-FAC-expand is exact in the expand setting.*

Proof. We can write $f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r = \boldsymbol{W}\boldsymbol{x}_{n,r}$ and hence $\boldsymbol{b}_{\ell,n,r,m} = \boldsymbol{0}$ for $r \neq m$. We have $\boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r) = \boldsymbol{\Sigma}^{-1}$ and $\hat{\boldsymbol{b}}_{\ell,n,r} = \boldsymbol{W}_\ell^{a^T}$. Hence, $\hat{\boldsymbol{b}}_{\ell,n,r} \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r) \hat{\boldsymbol{b}}_{\ell,n,r}^T = \boldsymbol{W}_\ell^{a^T} \boldsymbol{\Sigma}^{-1} \boldsymbol{W}_\ell^a$ for all $n \in \{1, \dots, N\}$ and $r \in \{1, \dots, R\}$. Therefore, the desired result follows from Lemma 3.1. □

3.1.2 The Reduce Setting and K-FAC-reduce

The second base setting is characterized by a loss with just N loss terms, i.e.

The Reduce Setting

$$\mathcal{L}_{\text{reduce}}(f_{\boldsymbol{\theta}}, \mathcal{D}) := -\sum_{n=1}^{N} \log p(y_n | f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)), \tag{3.15}$$

where the crucial observation is that the weight-sharing dimension must have been reduced somewhere in the forward pass of the neural network $f_{\boldsymbol{\theta}}$. A typical instance where this type of loss is used together with a model with linear weight-sharing layers is image classification with a Vision Transformer, introduced in Section 2.1.1.1. Note, that the inputs $\boldsymbol{x}_n$ and labels y_n do not have a weight-sharing dimension here; in general, it is also possible for the inputs to have this additional dimension of size R already.

Since $\boldsymbol{A}_{\ell,n} \in \mathbb{R}^{R \times P_{\ell,\text{in}}}$ is now a matrix, we have $\boldsymbol{S}_{\ell,n} = \boldsymbol{A}_{\ell,n} \boldsymbol{W}_\ell^T \in \mathbb{R}^{R \times P_{\ell,\text{out}}}$. Hence, $\boldsymbol{J}_{\boldsymbol{\theta}_\ell} \boldsymbol{S}_{\ell,n}$ and $\boldsymbol{J}_{\boldsymbol{S}_{\ell,n}} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)$ are now both tensors. Luckily, we can avoid dealing with tensors directly by writing

$$(\boldsymbol{J}_{\boldsymbol{\theta}_\ell} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))_{ij} = \sum_{r=1}^{R} \sum_{p=1}^{P_{\ell,\text{out}}} \frac{\partial f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_i}{\partial \boldsymbol{S}_{\ell,n,rp}} \frac{\partial \boldsymbol{S}_{\ell,n,rp}}{\partial \boldsymbol{\theta}_{\ell,j}}, \tag{3.16}$$

or in matrix form

$$\boldsymbol{J}_{\boldsymbol{\theta}_\ell} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n) = \sum_{r=1}^{R} \boldsymbol{J}_{\boldsymbol{s}_{\ell,n,r}} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n) \boldsymbol{J}_{\boldsymbol{\theta}_\ell} \boldsymbol{s}_{\ell,n,r}, \tag{3.17}$$

where $\boldsymbol{s}_{\ell,n,r} \in \mathbb{R}^{P_{\ell,\text{out}}}$ is the rth row of $\boldsymbol{S}_{\ell,n}$ and $\boldsymbol{s}_{\ell,n,r} = \boldsymbol{W}_\ell \boldsymbol{a}_{\ell,n,r}$.

Using this equivalence we can approximate the GGN for layer ℓ as

$$
\begin{aligned}
\mathbf{GGN}(\boldsymbol{\theta}_\ell) &= \sum_{n=1}^{N} \boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{x}_n)^T \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{x}_n) \\
&= \sum_{n=1}^{N} \left(\sum_{r=1}^{R} \boldsymbol{J}_{\boldsymbol{s}_{\ell,n,r}} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n) \boldsymbol{J}_{\boldsymbol{\theta}_\ell} \boldsymbol{s}_{\ell,n,r} \right)^T \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \left(\sum_{r=1}^{R} \boldsymbol{J}_{\boldsymbol{s}_{\ell,n,r}} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n) \boldsymbol{J}_{\boldsymbol{\theta}_\ell} \boldsymbol{s}_{\ell,n,r} \right) \\
&= \sum_{n=1}^{N} \left(\sum_{r=1}^{R} \boldsymbol{a}_{\ell,n,r} \otimes \boldsymbol{b}_{\ell,n,r} \right) \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \left(\sum_{r=1}^{R} \boldsymbol{a}_{\ell,n,r} \otimes \boldsymbol{b}_{\ell,n,r} \right)^T \\
&\approx \sum_{n=1}^{N} \underbrace{\left[\frac{1}{R} \sum_{r=1}^{R} \boldsymbol{a}_{\ell,n,r} \right]}_{=:\hat{\boldsymbol{a}}_{\ell,n}} \otimes \underbrace{\left[\sum_{r=1}^{R} \boldsymbol{b}_{\ell,n,r} \right]}_{=:\hat{\boldsymbol{b}}_{\ell,n}} \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \left[\frac{1}{R} \sum_{r=1}^{R} \boldsymbol{a}_{\ell,n,r}^T \right] \otimes \left[\sum_{r=1}^{R} \boldsymbol{b}_{\ell,n,r}^T \right] \\
&= \sum_{n=1}^{N} \left(\hat{\boldsymbol{a}}_{\ell,n} \hat{\boldsymbol{a}}_{\ell,n}^T \right) \otimes \left(\hat{\boldsymbol{b}}_{\ell,n} \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \hat{\boldsymbol{b}}_{\ell,n}^T \right) \\
&\approx \underbrace{\left[\frac{1}{N} \sum_{n=1}^{N} \hat{\boldsymbol{a}}_{\ell,n} \hat{\boldsymbol{a}}_{\ell,n}^T \right]}_{=:\hat{\boldsymbol{A}}_\ell} \otimes \underbrace{\left[\sum_{n=1}^{N} \hat{\boldsymbol{b}}_{\ell,n} \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \hat{\boldsymbol{b}}_{\ell,n}^T \right]}_{=:\hat{\boldsymbol{B}}_\ell},
\end{aligned}
\tag{3.18}
$$

where we have applied an approximation à la K-FAC a second time to the sum over the R terms of each of the N per-input Jacobians, before applying the same approximation as usual to the sum over the N data points. The idea to approximate the Jacobians within the GGN with a Kronecker-product has been proposed in the context of invariance learning with deep neural networks via differentiable Laplace approximations in Immer et al. (2022). We call the approximation in Equation (3.18) *K-FAC-reduce* and to highlight the difference to K-FAC-expand, we can rewrite it as

K-FAC-reduce

$$
\hat{\mathbf{GGN}}_{\boldsymbol{\theta}_\ell}^{\text{reduce}} := \underbrace{\left[\frac{1}{NR^2} \sum_{n=1}^{N} \left(\sum_{r=1}^{R} \boldsymbol{a}_{\ell,n,r} \right) \left(\sum_{r=1}^{R} \boldsymbol{a}_{\ell,n,r}^T \right) \right]}_{=\hat{\boldsymbol{A}}_\ell} \otimes \underbrace{\left[\sum_{n=1}^{N} \left(\sum_{r=1}^{R} \boldsymbol{b}_{\ell,n,r} \right) \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{X}_n)) \left(\sum_{r=1}^{R} \boldsymbol{b}_{\ell,n,r}^T \right) \right]}_{=\hat{\boldsymbol{B}}_\ell}. \tag{3.19}
$$

As for K-FAC-expand, we want to show that this approximation can be exact in the case of a single layer or a deep linear network and a Gaussian likelihood. First, we state an analogous condition to Lemma 3.1.

Lemma 3.4 (Sufficient condition for exactness of K-FAC-reduce in the reduce setting). *Let $\boldsymbol{D}_{\ell,n} \in \mathbb{R}^{P_{\ell,\text{out}} \times C}$ be a constant matrix for layer ℓ and data point $\boldsymbol{x}_n$. Further, let $\boldsymbol{C}_\ell \in \mathbb{R}^{P_{\ell,\text{out}} \times P_{\ell,\text{out}}}$ be a constant matrix for layer ℓ. If it holds for each n that $\boldsymbol{b}_{\ell,n,r} = \boldsymbol{D}_{\ell,n}$ for all $r \in \{1, \dots, R\}$ and $\hat{\boldsymbol{b}}_{\ell,n} \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \hat{\boldsymbol{b}}_{\ell,n}^T = \boldsymbol{C}_\ell$ for all $n \in \{1, \dots, N\}$, then the K-FAC-reduce approximation in Equation (3.19) is equal to the exact GGN of the ℓth layer in the reduce setting.*

Proof. We start with the first approximation and derive the exactness of this step under our assumptions. We have

$$
\begin{aligned}
&\sum_{n=1}^{N} \left(\frac{1}{R} \sum_{r=1}^{R} \boldsymbol{a}_{\ell,n,r} \otimes \sum_{r=1}^{R} \boldsymbol{b}_{\ell,n,r} \right) \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \left(\frac{1}{R} \sum_{r=1}^{R} \boldsymbol{a}_{\ell,n,r} \otimes \sum_{r=1}^{R} \boldsymbol{b}_{\ell,n,r} \right)^T \\
=&\sum_{n=1}^{N} \left(\frac{1}{R} \sum_{r=1}^{R} \boldsymbol{a}_{\ell,n,r} \otimes R\boldsymbol{D}_{\ell,n} \right) \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \left(\frac{1}{R} \sum_{r=1}^{R} \boldsymbol{a}_{\ell,n,r} \otimes R\boldsymbol{D}_{\ell,n} \right)^T \\
=&\sum_{n=1}^{N} \left(\sum_{r=1}^{R} \boldsymbol{a}_{\ell,n,r} \otimes \boldsymbol{b}_{\ell,n,r} \right) \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \left(\sum_{r=1}^{R} \boldsymbol{a}_{\ell,n,r} \otimes \boldsymbol{b}_{\ell,n,r} \right)^T,
\end{aligned}
\tag{3.20}
$$

where we have used the assumption that for each n, we have $\boldsymbol{b}_{\ell,n,r} = \boldsymbol{D}_{\ell,n}$ for all $r \in \{1, \dots, R\}$. Now we consider the second approximation in Equation (3.18). Analogously, we have

$$
\begin{aligned}
&\left(\frac{1}{N} \sum_{n=1}^{N} \hat{\boldsymbol{a}}_{\ell,n} \hat{\boldsymbol{a}}_{\ell,n}^T \right) \otimes \left(\sum_{n=1}^{N} \hat{\boldsymbol{b}}_{\ell,n} \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \hat{\boldsymbol{b}}_{\ell,n}^T \right) \\
=&\left(\frac{1}{N} \sum_{n=1}^{N} \hat{\boldsymbol{a}}_{\ell,n} \hat{\boldsymbol{a}}_{\ell,n}^T \right) \otimes N\boldsymbol{C}_\ell \\
=&\sum_{n=1}^{N} \left(\hat{\boldsymbol{a}}_{\ell,n} \hat{\boldsymbol{a}}_{\ell,n}^T \right) \otimes \left(\hat{\boldsymbol{b}}_{\ell,n} \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \hat{\boldsymbol{b}}_{\ell,n}^T \right),
\end{aligned}
\tag{3.21}
$$

where we have used that $\hat{\boldsymbol{b}}_{\ell,n} \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \hat{\boldsymbol{b}}_{\ell,n}^T = \boldsymbol{C}_\ell$ for all $n \in \{1, \dots, N\}$. □

Until now, we did not have to explicitly take the aggregation function $z : \mathbb{R}^{R \times P_{\ell,\text{out}}} \to \mathbb{R}^{P_{\ell,\text{out}}}$ into account, since its Jacobian is simply subsumed in $\boldsymbol{b}_{\ell,n,r}$. Since we want to verify that the approximation in the reduce case is also exact in the simple scenarios from Proposition 3.3 and Proposition 3.3, we now have to also check if the Jacobian $\boldsymbol{J}_{\boldsymbol{s}_{\ell,n,r}} \boldsymbol{z}_{\ell,n}$ with $\boldsymbol{z}_{\ell,n} := z(\boldsymbol{S}_{\ell,n}) \in \mathbb{R}^{P_{\ell,\text{out}}}$ is the same for all $r \in \{1, \dots, R\}$, to make sure the first condition in Lemma 3.4 is fulfilled. Maybe the simplest case where this holds is a scaled sum, i.e.

$$
\begin{aligned}
z(\boldsymbol{S}_{\ell,n}) &= c \sum_{r=1}^{R} \boldsymbol{s}_{\ell,n,r} \\
&= c\boldsymbol{S}_{\ell,n}^T \mathbf{1}_R \\
&= \left(\mathbf{1}_R^T \otimes c\mathbf{I}_{P_{\ell,\text{out}}} \right) \text{vec}(\boldsymbol{S}_{\ell,n}^T) \\
&= \left(\mathbf{1}_R^T \otimes c\mathbf{I}_{P_{\ell,\text{out}}} \right) \boldsymbol{K}^{(R,P_{\ell,\text{out}})} \text{vec}(\boldsymbol{S}_{\ell,n})
\end{aligned}
\tag{3.22}
$$

with $c \in \mathbb{R}$ and the commutation matrix

$$\boldsymbol{K}^{(R,P_{\ell,\text{out}})} := \sum_{r=1}^{R} \sum_{p=1}^{P_{\ell,\text{out}}} (\boldsymbol{e}_{R,r}\boldsymbol{e}_{P_{\ell,\text{out}},p}^T) \otimes (\boldsymbol{e}_{P_{\ell,\text{out}},p}\boldsymbol{e}_{R,r}^T), \tag{3.23}$$

where $\boldsymbol{e}_{i,j}$ is the jth canonical vector of dimension i. This is a linear function in $\text{vec}(\boldsymbol{S}_{\ell,n})$ and we have $\boldsymbol{J}_{\boldsymbol{s}_{\ell,n,r}}\boldsymbol{z}_{\ell,n} = c\mathbf{I}_{P_{\ell,\text{out}}}$ for all $r \in \{1,\dots,R\}$. In particular, when $c = 1$ the aggregation function is a simple sum and when $c = 1/R$ it is the mean. Notably, it is *not* sufficient for z to be linear in $\text{vec}(\boldsymbol{S}_{\ell,n})$, because as soon as we have a weighted sum with weights $c_r \in$ and they are not the same for all $r \in \{1,\dots,R\}$, the Jacobians $\boldsymbol{J}_{\boldsymbol{s}_{\ell,n,r}}\boldsymbol{z}_{\ell,n}$ will also not be the same anymore. One example of an architecture that commonly uses a scaled sum as the aggregation function is the Vision Transformer (with $c = 1/R$).

After clarifying the role of the aggregation function in the exactness of K-FAC-reduce, we can now state a similar statement to Proposition 3.2.

Proposition 3.5 (Exactness of K-FAC-reduce for single layer and Gaussian likelihood in the reduce setting). *For a single linear layer, a Gaussian likelihood with p.d. covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{C\times C}$, and a scaled sum as defined in Equation (3.22) as the aggregation function applied to the output of the linear function, K-FAC-reduce is exact in the reduce setting.*

Proof. We have $\boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) = \boldsymbol{\Sigma}^{-1}$ and $\boldsymbol{b}_{\ell,n,r} = \left(\boldsymbol{J}_{\boldsymbol{z}_{\ell,n}} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)\boldsymbol{J}_{\boldsymbol{s}_{\ell,n,r}}\boldsymbol{z}_{\ell,n}\right)^T = c\mathbf{I}_C$ for all $r \in \{1,\dots,R\}$ and $n \in \{1,\dots,N\}$ ($P_{\ell,\text{out}} = C$ for a single layer). Hence, $\hat{\boldsymbol{b}}_{\ell,n}\boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))\hat{\boldsymbol{b}}_{\ell,n}^T = c^2R^2\mathbf{I}_C\boldsymbol{\Sigma}^{-1}\mathbf{I}_C = c^2R^2\boldsymbol{\Sigma}^{-1}$ for all $n \in \{1,\dots,N\}$. Therefore, the desired result follows from Lemma 3.4. □

Just as for K-FAC-expand, we can extend this result to deep linear networks.

Proposition 3.6 (Exactness of K-FAC-reduce for deep linear network and Gaussian likelihood in the reduce setting). *For layer ℓ of a deep linear network defined in Equation (3.10), a Gaussian likelihood with p.d. covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{C\times C}$, and a scaled sum as defined in Equation (3.22) as the aggregation function applied after all linear layers, K-FAC-reduce is exact in the reduce setting.*

Proof. We have $\boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r) = \boldsymbol{\Sigma}^{-1}$ and $\boldsymbol{b}_{\ell,n,r} = \left(\boldsymbol{J}_{\boldsymbol{z}_{\ell,n}} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)\boldsymbol{J}_{\boldsymbol{s}_{\ell,n,r}}\boldsymbol{z}_{\ell,n}\right)^T = c\boldsymbol{W}_\ell^{a^T}$ for all $r \in \{1,\dots,R\}$ and $n \in \{1,\dots,N\}$. Hence,

$$\hat{\boldsymbol{b}}_{\ell,n}\boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))\hat{\boldsymbol{b}}_{\ell,n}^T = c^2R^2\boldsymbol{W}_\ell^{a^T}\boldsymbol{\Sigma}^{-1}\boldsymbol{W}_\ell^a$$

for all $n \in \{1,\dots,N\}$. Therefore, the desired result follows from Lemma 3.4. □

To summarize, the difference between the expand and the reduce setting is at what point the aggregation over the additional weight-sharing dimension happens. If this dimension is not aggregated before the per-example loss, i.e. if the loss can be expanded to $N \cdot R$ instead of N terms, we call it the expand setting. If the aggregation happens inside the model, we call it the reduce setting. Both settings motivate an approximation each, K-FAC-expand and K-FAC-reduce. Moreover, we presented simple cases where the approximations are exact. In Section 4.1 we verify this numerically, and also show that using the inappropriate approximation results in

an inexact computation. In practice, however, both approximations can be applied in each of the two settings.

This point becomes even more relevant as the computational complexity of the calculation of the Kronecker factors also differs between the two approximations: while the calculation of $\boldsymbol{A}_\ell$ costs $\mathcal{O}(NRP_{\ell,\text{in}}^2)$ for K-FAC-expand, it is reduced to $\mathcal{O}(NP_{\ell,\text{in}}^2 + NRP_{\ell,\text{in}})$ for $\hat{\boldsymbol{A}}_\ell$ in the case of K-FAC-reduce. The same holds for $\boldsymbol{B}_\ell$ and $\hat{\boldsymbol{B}}_\ell$, since the complexity is reduced from $\mathcal{O}(NRP_{\ell,\text{out}}C^2 + NRP_{\ell,\text{out}}^2 C)$ to $\mathcal{O}(NP_{\ell,\text{out}}C^2 + NP_{\ell,\text{out}}^2 C + NRP_{\ell,\text{out}}C)$.

3.2 Example: K-FAC for Transformers

While we have mentioned (Vision) Transformers for translation and image classification as prototypical examples for the expand and the reduce setting, we have mostly ignored how linear weight-sharing layers are used within the architecture and how this affects the approximation quality of K-FAC-expand and K-FAC-reduce. Linear weight-sharing layers are crucial for the scaled dot-product attention mechanism in Equation (2.4). To get some intuition for models using this type of attention mechanism, we look at a network that only consists of one simplified variation of this scaled dot-product self-attention mechanism used in Transformers (we ignore the scaling and the softmax function), i.e.

$$f_{\boldsymbol{\theta}}(\boldsymbol{X}_n) = \underbrace{\boldsymbol{X}_n \boldsymbol{W}^{Q^T}}_{=:\boldsymbol{S}_{Q,n}} \underbrace{\boldsymbol{W}^K \boldsymbol{X}_n^T}_{=:\boldsymbol{S}_{K,n}^T} \underbrace{\boldsymbol{X}_n \boldsymbol{W}^{V^T}}_{=:\boldsymbol{S}_{V,n}}. \tag{3.24}$$

We can observe that it is no longer a linear function in the input $\boldsymbol{X}_n \in \mathbb{R}^{R \times D}$ and that we have three linear weight-sharing layers involved in this operation. First, we consider the expand setting, i.e. the output $f_{\boldsymbol{\theta}}(\boldsymbol{X}_n)$ is not reduced before the loss is applied.

Simplified dot-product attention in the expand setting. Since we want to understand if K-FAC-expand can be exact in this case, we first derive the Jacobians appearing in the derivation of K-FAC-expand in Equation (3.6) for all three involved layers, i.e. $\boldsymbol{J}_{\boldsymbol{s}_{Q,n,m}} f_{\boldsymbol{\theta}}(\boldsymbol{X}_n)_r$ for the layer with weights $\boldsymbol{W}^Q$, $\boldsymbol{J}_{\boldsymbol{s}_{K,n,m}} f_{\boldsymbol{\theta}}(\boldsymbol{X}_n)_r$ for the layer with weights $\boldsymbol{W}^K$, and $\boldsymbol{J}_{\boldsymbol{s}_{V,n,m}} f_{\boldsymbol{\theta}}(\boldsymbol{X}_n)_r$ for the layer with weights $\boldsymbol{W}^V$.

We can simply write the rth row of the output of the layer with the weight matrix $\boldsymbol{W}^Q$ as a function of $\boldsymbol{s}_{Q,n,r}$ as

$$f_{\boldsymbol{\theta}}(\boldsymbol{X}_n)_r = \boldsymbol{s}_{Q,n,r}^T \boldsymbol{S}_{K,n}^T \boldsymbol{S}_{V,n}. \tag{3.25}$$

Therefore, we have

$$\boldsymbol{J}_{\boldsymbol{s}_{Q,n,r}} f_{\boldsymbol{\theta}}(\boldsymbol{X}_n)_r = \boldsymbol{S}_{V,n}^T \boldsymbol{S}_{K,n} = \boldsymbol{b}_{Q,n,r}^T \in \mathbb{R}^{C \times P_{K,\text{out}}}, \tag{3.26}$$

with $C = P_{V,\text{out}}$ and $\boldsymbol{b}_{Q,n,r,m} = \boldsymbol{0}$ for all $m \neq r$, which is the first assumption necessary for Lemma 3.1 to hold. While $\boldsymbol{b}_{Q,n,r}$ is not the same for all $n \in \{1, \dots, N\}$, it is the same for all $r \in \{1, \dots, R\}$ and hence, under the same assumptions as in Proposition 3.2, K-FAC-expand is exact for the layer with weights $\boldsymbol{W}^Q$ in the special case of a single data point, $N = 1$.

For the other two involved linear layers, we cannot express the rth row of $f_{\boldsymbol{\theta}}(\boldsymbol{X}_n)$ as a function of the rth row of $\boldsymbol{S}_{K/V,n}$, i.e. elements from all rows of $\boldsymbol{S}_{K/V,n}$ contribute to the rth row of the output matrix. We can also see this by directly deriving $\boldsymbol{J}_{\boldsymbol{s}_{K/V,n,m}} f_{\boldsymbol{\theta}}(\boldsymbol{X}_n)_r$ which will be generally non-zero and dependent on r and m. We omit the explicit derivation by taking the

partial derivatives and directly state the results. For the second layer with weight matrix $\boldsymbol{W}^K$, we have

$$\boldsymbol{J}_{\boldsymbol{s}_{K,n,m}} f_{\boldsymbol{\theta}}(\boldsymbol{X}_n)_r = \boldsymbol{s}_{V,n,m} \boldsymbol{s}_{Q,n,r}^T \in \mathbb{R}^{C \times P_{Q,\text{out}}}. \tag{3.27}$$

Moreover, for the third layer with weight matrix $\boldsymbol{W}^V$, we have

$$\boldsymbol{J}_{\boldsymbol{s}_{V,n,m}} f_{\boldsymbol{\theta}}(\boldsymbol{X}_n)_r = \boldsymbol{s}_{Q,n,r}^T \boldsymbol{s}_{K,n,m} \mathbf{I}_C \in \mathbb{R}^{C \times C}. \tag{3.28}$$

This means that the assumption of Lemma 3.1 that the R elements along the weight-sharing dimension are independent does not hold, since the Jacobians depends on r and m. The approximation leads to an inexact computation, even though only linear layers are involved and a Gaussian likelihood is used. Similarly, we can inspect the corresponding reduce case.

Simplified dot-product attention in the reduce setting. Assuming we use a scaled sum z with factor c as the aggregation function, we can further rewrite the Jacobians occurring in Equation (3.18) as

$$\begin{aligned}
\boldsymbol{J}_{\boldsymbol{\theta}_\ell} z(f_{\boldsymbol{\theta}}(\boldsymbol{X}_n)) &= \sum_{r=1}^{R} \boldsymbol{J}_{\boldsymbol{s}_{\ell,n,r}} z(f_{\boldsymbol{\theta}}(\boldsymbol{X}_n)) \boldsymbol{J}_{\boldsymbol{\theta}_\ell} \boldsymbol{s}_{\ell,n,r} \\
&= \sum_{r=1}^{R} \boldsymbol{a}_{\ell,n,r} \otimes \boldsymbol{b}_{\ell,n,r} \\
&= \sum_{r=1}^{R} \boldsymbol{a}_{\ell,n,r} \otimes \left(c \sum_{m=1}^{R} \boldsymbol{J}_{\boldsymbol{s}_{\ell,n,r}} f_{\boldsymbol{\theta}}(\boldsymbol{X}_n)_m \right),
\end{aligned} \tag{3.29}$$

where $\boldsymbol{J}_{\boldsymbol{s}_{\ell,n,r}} f_{\boldsymbol{\theta}}(\boldsymbol{X}_n)_m$ are the same Jacobians we have derived for the expand case. Since according to Lemma 3.4 we need all $\boldsymbol{b}_{\ell,n,r}$ to be the same for all $r \in \{1, \dots, R\}$ for the first approximation to be exact under the assumptions of Proposition 3.5, K-FAC-reduce is only exact when $N = 1$ and only for the layer with weights $\boldsymbol{W}^Q$ – just as K-FAC-expand in the expand setting.

We can extend this scenario to a network consisting of L blocks as defined in Equation (3.24), the above statements regarding the special case where K-FAC-expand and K-FAC-reduce are exact for the layer with weights $\boldsymbol{W}^Q$ only hold for the *last* block. While we omit an explicit derivation, intuitively, this can be seen by the fact that we cannot rewrite the rth row of this model's output as a function of only the rth row of the layer's output $\boldsymbol{S}_{\ell,Q,n}$ of all layers with weights $\boldsymbol{W}_\ell^Q$, besides for the layer in the last block, i.e. the layer in the Lth block with weights $\boldsymbol{W}_L^Q$.

This shows that even without explicit nonlinear activation functions, the dot-product attention mechanism in Transformer models breaks the two approximations. Hence, it is not inherently clear how useful it is to consider the corresponding approximation in the expand and reduce setting. This becomes especially relevant given that we know that the computational complexity of K-FAC-reduce is smaller than of K-FAC-expand; we will continue this line of thought in Section 3.4 and Chapter 4.

3.3 Example: K-FAC for GNNs

Beyond Transformers, we have introduced GNNs as a class of models that also use linear weight-sharing layers. There are many types of GNNs and we will only explicitly cover two of them here.

Related work: node classification with GCN. We first consider a GCN layer, described in Section 2.1.1.2. This specification of K-FAC for GNNs has been previously derived for semi-supervised node classification in Izadi et al. (2020) and we include it for completeness, since it is, to the best of our knowledge, the only case where K-FAC has been applied to GNNs. The only difference to the normal derivation of K-FAC is that the inputs $\tilde{\boldsymbol{a}}_{\ell,n}$ to the ℓth layer for node with index n now depend on its neighborhood $\mathcal{N}(n)$, since

$$\tilde{\boldsymbol{a}}_{\ell,n} := \sum_{j \in \mathcal{N}(n)} \hat{\boldsymbol{C}}_{nj} \boldsymbol{a}_{\ell,j} \in \mathbb{R}^{P_{\ell,\text{in}}}. \tag{3.30}$$

Using this notation, the definition of the K-FAC GGN for node classification is simply

$$\begin{aligned} GGN(\boldsymbol{\theta}_\ell) &= \sum_{n=1}^{N} \boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{X})_n^T \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{X})_n) \boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{X})_n \\ &\approx \underbrace{\big[\frac{1}{N} \sum_{n=1}^{N} \tilde{\boldsymbol{a}}_{\ell,n} \tilde{\boldsymbol{a}}_{\ell,n}^T\big]}_{=:\tilde{\boldsymbol{A}}_\ell} \otimes \underbrace{\big[\sum_{n=1}^{N} \tilde{\boldsymbol{b}}_{\ell,n} \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{X})_n) \tilde{\boldsymbol{b}}_{\ell,n}^T\big]}_{=:\tilde{\boldsymbol{B}}_\ell}, \end{aligned} \tag{3.31}$$

where $\boldsymbol{X} \in \mathbb{R}^{N \times D}$, $\tilde{\boldsymbol{b}}_{\ell,n} := \boldsymbol{J}_{\tilde{\boldsymbol{s}}_{\ell,n}} f_{\boldsymbol{\theta}_\ell}(\boldsymbol{X})_n^T$, and $\tilde{\boldsymbol{s}}_{\ell,n} := \boldsymbol{W}_\ell \tilde{\boldsymbol{a}}_{\ell,n}$. Again, it is important to note that we need to have access to the whole neighborhood of $\boldsymbol{x}_n$ to be able to write the nth term of the GGN, which is why the input to the model is the matrix $\boldsymbol{X}$ containing all nodes for each loss term. Also, depending on the sparsity of $\hat{\boldsymbol{C}}$, i.e. the size of neighborhoods, we might have multiple identical terms. In the extreme case of all neighborhoods being the same, e.g. in the case of a fully connected graph, i.e. are values of $\hat{\boldsymbol{C}}$ are the same, all terms of the GGN will be the same.

According to our initial three cases, because we aggregate over each node's neighborhood *before* the forward pass through a linear layer, we do not need to think in terms of the expand and reduce settings here – as opposed to the case of the graph network we consider next.

Graph classification with graph network. Now, we want to look at a more general architecture, an instance of the graph network introduced in Battaglia et al. (2018) and described in Section 2.1.1.2. It is important to note that while the inputs and the graph network block structure look different from our standard input and linear layer, this case can be treated the same. This architecture is therefore a good didactic example of how to apply the here presented framework of thinking about K-FAC for linear weight-sharing layers to new model architectures. In contrast to the original description in Battaglia et al. (2018), we already defined the inputs to a graph network block according to our definition of an input that leads to weight-sharing, i.e. with an additional weight-sharing dimension of size R. This is in fact the crucial step to be able to apply our framework in this setting. As noted in Section 2.1.1.2, the inputs cannot be trivially batched in this formulation. This is not an issue for our derivation, but it requires special consideration in the implementation, which we will consider in Section 3.4.

First, we note that we consider the task of graph classification. Hence, our loss has the same form as Equation (3.15), which means that the weight-sharing dimensions have to be reduced at some point during the forward pass and we are in the reduce setting. Notably, this is the setting of the OGBG workload of the AlgoPerf benchmark introduced in Section 2.3. Following this line of reasoning, we would simply have to apply the corresponding K-FAC approximation. Since the inputs take a more complex form than in our description of the reduce case, we still have to adopt the notation from Section 2.1.1.2 to concretely write down the approximation.

To recap, a single graph input is defined as a 5-tuple $\mathbb{X}_n^G := (\boldsymbol{x}_n^u, \boldsymbol{X}_n^V, \boldsymbol{X}_n^E, \boldsymbol{r}_n, \boldsymbol{s}_n)$, where $\boldsymbol{x}_n^u \in \mathbb{R}^{D_u}$ are the global features, $\boldsymbol{X}_n^V \in \mathbb{R}^{N_n^V \times D_V}$ are the node features, and $\boldsymbol{X}_n^E \in \mathbb{R}^{N_n^E \times D_E}$ are the edge features. The vectors $\boldsymbol{r}_n \in \mathbb{R}^{N_n^E}$ and $\boldsymbol{s}_n \in \mathbb{R}^{N_n^E}$ store the indices of the receiving and sending nodes for each edge, respectively. The weight-sharing dimension of size R_n of graph $\mathbb{X}_n^G$ depends on the input graph itself (indicated by the index n) and which update function within a graph network block we want to derive K-FAC-reduce for. For ϕ^E this dimension is going to be $R_n = N_n^E$, whereas it will be $R_n = N_n^V$ for ϕ^V. In the case of ϕ^u we do not have a weight-sharing dimension, as it has been reduced before this layer is applied, and we can simply apply the regular K-FAC approximation. We can define the inputs to layer ℓ of type ϕ^E as

$$\boldsymbol{A}_{\ell,n} = \operatorname{concat}(\boldsymbol{X}_n^E, \boldsymbol{X}_{n,\boldsymbol{r}_n}^V, \boldsymbol{X}_{n,\boldsymbol{s}_n}^V, \operatorname{repeat}_{N_n^E}(\boldsymbol{x}_n^u)) \in \mathbb{R}^{N_n^E \times (D_E + 2D_V + D_u)} \tag{3.32}$$

and as

$$\boldsymbol{A}_{\ell,n} = \operatorname{concat}(\boldsymbol{X}_n^V, \tilde{\boldsymbol{X}}_n^E, \operatorname{repeat}_{N_n^V}(\boldsymbol{x}_n^u)) \in \mathbb{R}^{N_n^V \times (D_V + D_E + D_u)} \tag{3.33}$$

for ϕ^V. Correspondingly, we have $\boldsymbol{b}_{\ell,n} = \boldsymbol{J}_{\boldsymbol{S}_{\ell,n}} f_{\boldsymbol{\theta}}(\mathbb{X}_n^G)^T \in \mathbb{R}^{N_n^E \times D_E \times C}$ for ϕ^E and $\boldsymbol{b}_{\ell,n} = \boldsymbol{J}_{\boldsymbol{S}_{\ell,n}} f_{\boldsymbol{\theta}}(\mathbb{X}_n^G)^T \in \mathbb{R}^{N_n^V \times D_V \times C}$, with $\boldsymbol{S}_{\ell,n} = \boldsymbol{A}_{\ell,n} \boldsymbol{W}_\ell^{E^T} \in \mathbb{R}^{N_n^E \times D_E}$ and $\boldsymbol{S}_{\ell,n} = \boldsymbol{A}_{\ell,n} \boldsymbol{W}_\ell^{V^T} \in \mathbb{R}^{N_n^V \times D_V}$, respectively.

Using this notation, we can approximate the GGN for layer ℓ, assuming its type is either ϕ^E or ϕ^V, as

$$\begin{aligned}
\mathbf{GGN}(\boldsymbol{\theta}_\ell) &= \sum_{n=1}^{N} \boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\mathbb{X}_n^G)^T \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\mathbb{X}_n^G)) \boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\mathbb{X}_n^G) \\
&\approx \sum_{n=1}^{N} \underbrace{\left[\frac{1}{R_n} \sum_{r=1}^{R_n} \boldsymbol{a}_{\ell,n,r}\right]}_{=:\hat{\boldsymbol{a}}_{\ell,n}} \otimes \underbrace{\left[\sum_{r=1}^{R_n} \boldsymbol{b}_{\ell,n,r}\right]}_{=:\hat{\boldsymbol{b}}_{\ell,n}} \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\mathbb{X}_n^G)) \left[\frac{1}{R_n} \sum_{r=1}^{R_n} \boldsymbol{a}_{\ell,n,r}\right] \otimes \left[\sum_{r=1}^{R_n} \boldsymbol{b}_{\ell,n,r}\right]^T \\
&\approx \underbrace{\left[\frac{1}{N} \sum_{n=1}^{N} \hat{\boldsymbol{a}}_{\ell,n} \hat{\boldsymbol{a}}_{\ell,n}^T\right]}_{=:\hat{\boldsymbol{A}}_\ell} \otimes \underbrace{\left[\sum_{n=1}^{N} \hat{\boldsymbol{b}}_{\ell,n} \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\mathbb{X}_n^G)) \hat{\boldsymbol{b}}_{\ell,n}^T\right]}_{=:\hat{\boldsymbol{B}}_\ell},
\end{aligned} \tag{3.34}$$

analogously to Equation (3.18).

```
# Check if there even is a weight-sharing dimension; if not, the Kronecker
# factors can directly be calculated.
if in_data.ndim == 3:
    # Mini-batch size M, weight-sharing dimension R, feature dimension P_{ℓ,in/out}.
    M, R, P_in = in_data.shape
    P_out = out_grads.shape[2]
    if approximation == 'expand':
        # Flatten the weight-sharing dimension into the mini-batch dimension.
        in_data = in_data.view(M*R, P_in) / math.sqrt(R)
        out_grads = out_grads.view(M*R, P_out)
    elif approximation == 'reduce':
        # Reduce the weight-sharing dimension with mean and sum.
        in_data = in_data.mean(dim=1)
        out_grads = out_grads.sum(dim=1)
# Calculate Kronecker factors A_ℓ/Â_ℓ and B_ℓ/B̂_ℓ.
A = torch.matmul(in_data.T, in_data) / M
B = torch.matmul(out_grads.T, out_grads)
```

Listing 3.1: Illustration of K-FAC-expand and K-FAC-reduce with code. This piece of code calculates the approximations on one mini-batch and for one layer. Here we assume that only one additional weight-sharing dimension exists and that the first dimension is always the mini-batch dimension. The actual implementation in `ASDL` is very similar, the logic is just separated into multiple functions and does not make these simplifying assumptions. We receive the inputs to the layer, `in_data`, from a forward hook and the gradients of the loss w.r.t. the outputs of the layer, `out_grads`, from a backward hook.

3.4 Practical Considerations

While we have discussed theoretically how to apply K-FAC to linear weight-sharing layers, we now turn to implementation details and computational considerations that are crucial for the practical application of the approximations.

Implementation details. There are multiple libraries that implement K-FAC for popular deep learning frameworks like `Jax` (Bradbury et al., 2018) and `PyTorch` (Paszke et al., 2019), e.g. `KFAC-JAX` (Botev & Martens, 2022) for `Jax` and `BackPACK` (Dangel et al., 2020), `ASDL` (Osawa, 2021), and `KFAC-PyTorch` (Pauloski et al., 2021) for `PyTorch`. We focus on the implementation of the expand and the reduce case within `ASDL`, which we also use for the experiments in Chapter 4. K-FAC is implemented using forward and backward hooks, which allow us to get the inputs to a specific layer and the gradients of the loss w.r.t. the layer outputs – which are the ingredients we need for all K-FAC approximations. Notably, this requires that linear layers are implemented with `torch.nn.Linear` instances, since otherwise, the implementation with hooks does not work. The default implementation of multi-head attention in `PyTorch` does indeed not use the required linear modules, so the implementation has to be adjusted to work with common K-FAC implementations like `ASDL`. In contrast, other methods like Shampoo (Gupta et al., 2018) and Tensor Normal Training (Ren & Goldfarb, 2021) are agnostic to the architecture. Assuming the implementation of the model is appropriate, K-FAC-expand and K-FAC-reduce in their simplest form only require a minor adjustment in the code base for regular K-FAC, which is presented in Listing 3.1.

However, if we wanted to use K-FAC-expand in the expand and K-FAC-reduce in the reduce setting, we would need to find a way of automatically determining the setting we are in. For all models considered here, i.e. the (Vision) Transformer and GNN, only one of the two settings applies to all linear layers with weight-sharing. Hence, using a single additional forward pass,

we could check if any linear weight-sharing layers are used and what the shape of the model output is. From this, we can deduce if the expand or the reduce case applies. As we mentioned before, this might not even be desirable, as it is unclear if we should always use the approximation theoretically motivated by the setting. Alternatively, a single flag set by the user can just determine if K-FAC-expand or K-FAC-reduce is applied to all linear weight-sharing layers.

This implementation obviously assumes that we even have an explicit weight-sharing dimension. In the case of K-FAC-reduce for the GNN in Section 3.3, we have to adopt our implementation due to the batching technique which is employed for graph inputs in practice. Since each graph in a mini-batch $\mathcal{M}$ of size M might have a different weight-sharing dimension R_m, i.e. the number of nodes and the number of edges of each graph, we cannot batch them trivially. As a solution, the inputs for each graph as stated in Equation (3.32) and Equation (3.33) are simply concatenated in the first dimension, which results in a dimension of size $R_{\mathcal{M}} := \sum_{m=1}^{M} R_m$. To apply K-FAC-expand here, we do not have to modify anything, besides scaling the approximation for each mini-batch by $1/R_{\mathcal{M}}$ instead of $1/M$. To apply K-FAC-reduce, we can use a *scatter mean and sum*, which aggregates tensor elements according to indices, to implement the mean and sum operation without having an explicit weight-sharing dimension. Unfortunately, this creates two issues. First, we have to know that this adjustment to K-FAC is even required for a specific layer, since we cannot deduce it from the shape of the layer inputs. Second, the scatter mean/sum requires additional information, since we need to know to which graphs the nodes/edges in the input belong. One approach to resolve these issues is to define a custom layer type for this type of linear layer, which has an attribute containing the indices of all nodes/edges for each graph in the batch. However, this requires changes to the model architecture, because regular linear layers would have to be replaced by this particular subclass of them.

Besides the changes necessary for K-FAC-expand and K-FAC-reduce, we can use the same additional algorithmic tools often used for optimization with K-FAC. Typically, *damping* is used (Martens & Grosse, 2015), i.e. a scalar is added to the diagonal of the two Kronecker factors $\boldsymbol{A}$ and $\boldsymbol{B}$ or the diagonal of their product – the latter corresponds to adding the Hessian of an isotropic Gaussian prior over the weights. Also, since we usually operate in the stochastic setting and only compute the K-FAC approximation on mini-batches, sometimes an exponential moving average over the Kronecker factors is used.

Computational considerations. Besides the implementation details, we also have to consider the computational cost when deploying K-FAC approximations in practice. Here, we have to respect the same constraints as with regular K-FAC. When we have a large output dimension C, it is expensive or even unfeasible to propagate the $C \times C$ loss Hessian $\boldsymbol{H}_{f_{\boldsymbol{\theta}}} \ell(y_n, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))$ for each of the N data points through the computation graph. Instead, we use the fact that we have

$$\mathbb{E}_{y \sim p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))}[\nabla_{f_{\boldsymbol{\theta}}} \log p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \nabla_{f_{\boldsymbol{\theta}}} \log p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))^T] = \boldsymbol{H}_{f_{\boldsymbol{\theta}}} \ell(y_n, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \tag{3.35}$$

for the losses considered here, c.f. Section 2.2.3, and take S Monte Carlo (MC) samples from the model's predictive distribution $y_s \sim p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)$. Taking a single sample results in a rank-1 MC approximation of the true loss Hessian and only requires the propagation of a single vector through the computation graph for each data point.

Also, to decrease the computational overhead of K-FAC the frequency of how often the Kronecker factors and the inverse for preconditioning are computed can be decreased, such that these operations are not executed at every iteration. As noted in Section 3.1.2, K-FAC-reduce has lower computational complexity than K-FAC-expand.

While we have shown that using the corresponding approximation for each setting leads to exact computations in the same simple cases as with regular K-FAC, one might ask if this also holds for realistic nonlinear deep networks. We have seen in Section 3.2 that even for simplified self-attention mechanisms, the approximations are generally inexact. Hence, there is no guarantee that choosing the, according to the setting, appropriate approximation leads to better empirical results, and it is at least imaginable that using the "wrong" approximation provides a comparable or even greater benefit in practice. If this were the case, it might be interesting to apply K-FAC-reduce in the expand scenario, since it is simply faster. We therefore use both approximation in Chapter 4, irrespective of the setting at hand.

Chapter 4
Experiments

The goal of our experiments is to (i) visualize our findings from Chapter 3, (ii) provide a proof of concept that K-FAC can potentially be useful for optimizing models with linear weight-sharing layers, in particular a Vision Transformer on ImageNet, and (iii) test if it provides any benefit to use the theoretically motivated "correct" K-FAC approximation corresponding to the setting at hand. In Section 4.2, to address points (ii) and (iii), we choose to focus on the Vision Transformer ImageNet workload among the three workloads introduced in Section 2.3 because it represents the reduce case, in which K-FAC, to the best of our knowledge, has never been applied before.

4.1 Visualizations

As we describe in Chapter 3, there are two settings that motivate two different approximations. In each setting, we expect the corresponding approximation to be exact in simple cases, involving only (deep) linear networks and a Gaussian likelihood. However, both approximations, K-FAC-expand and K-FAC-reduce, can be applied in each setting. To visualize the difference between the two approximations and verify the claims in Section 3.1 numerically, we look at the quantities involved in computing the K-FAC approximations for the third layer of a small six-layer deep linear network. We show `in_data` and `out_grads` after the preprocessing step for the two approximations, illustrated in Listing 3.1, has been applied.

In Figure 4.1, we consider the expand setting and show that while K-FAC-expand is exact for this layer, K-FAC-reduce is not. We can observe that, as expected, the `out_grads` are the same across the data and weight-sharing dimension of size $N \cdot R$. Also, the only difference between $\boldsymbol{B}$ in both approximations is a scalar factor of $1/R$. Similarly, we can observe in Figure 4.2 that K-FAC-reduce is exact in the reduce case, whereas K-FAC-expand is not. The observation about `out_grads` and $\boldsymbol{B}$ from the expand case also holds here.

While choosing the appropriate approximation for the setting leads to exact computations in these very simple cases, as soon as we consider even simplified Transformer models described in Section 3.2, both approximations are usually inexact and there is no obvious "correct" choice anymore. When we introduce nonlinearities and more complex operations, the choice becomes even less clear. Hence, these simple toy cases do not provide us with much, if any, information for the usage of the two approximations in practice – we need to consider proper benchmarks.

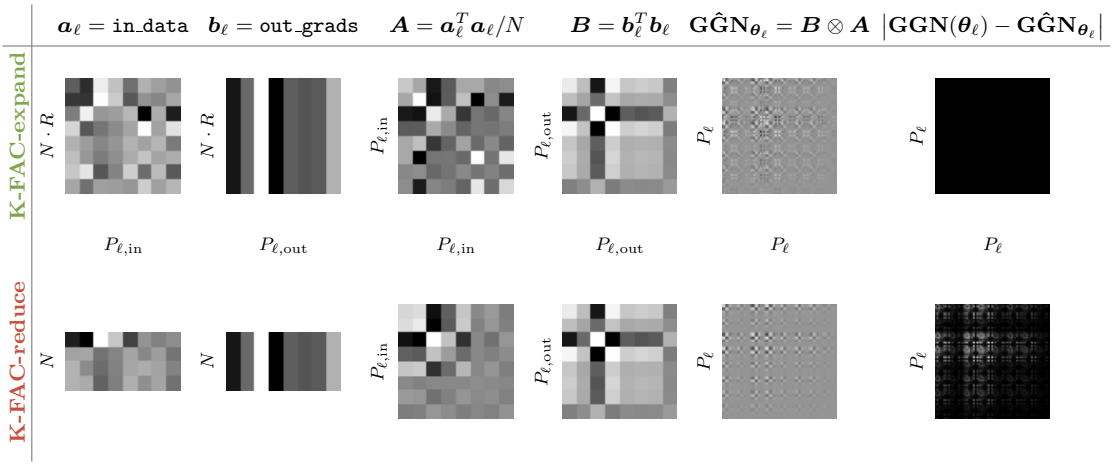

Figure 4.1: Visualization of K-FAC-expand and K-FAC-reduce in the expand setting. The shown quantities are for one of six layers of a deep linear network. We have $N = 4, R = 2, P_{\ell,\text{in}} = 8, P_{\ell,\text{out}} = 8$, and $P_\ell = P_{\ell,\text{in}} \cdot P_{\ell,\text{out}} = 64$. As we have seen in Section 3.1.1, K-FAC-expand is exact for the expand case in this setting and K-FAC-reduce is not. Note that the scale of gray tones is not the same for all quantities for better visibility; however, it is the same for the two plots of the approximation error (black is equivalent to zero).

4.2 Vision Transformer on ImageNet

To test the approximations in the reduce setting, where we have a loss of the type of Equation (3.15), we use both variations to train a Vision Transformer on the ImageNet dataset, a workload of the AlgoPerf benchmark introduced in Section 2.3[1]. We also stick to the setup of this benchmark, i.e. we train until we reach a target validation accuracy of 0.77171. Evaluation happens on the full validation dataset and a subset of the training data. The validation target was set by running a baseline algorithm, NAdamW (Loshchilov & Hutter, 2019) (see Table 1 in Choi et al. (2019) for the specific implementation), with multiple different hyperparameter settings, finding the hyperparameters leading to the best validation performance and taking the average validation accuracy over multiple runs with different random seeds for this best setting. We also use NAdamW with this hyperparameter setting as the baseline to compare the K-FAC approximations against, as it represents a well-tuned first-order method. It uses a learning rate of about $2e-3$, $\beta_1 = 0.7132$, $\beta_2 = 0.9982$, weight decay of 0.026595, $\epsilon = 1e-8$, and a batch size of 1024. Moreover, it clips the gradients to keep their norm below 1.

For K-FAC-expand and K-FAC-reduce, we only tune the learning rate schedule, i.e. the warm-up steps and the expected number of steps. Then we simply apply the baseline algorithm to the preconditioned gradients, using the same hyperparameters. K-FAC-expand and K-FAC-reduce use 10.5k warm-up steps and a step hint of 105k, whereas the baseline uses about 14k and 140k, respectively, which simply corresponds to applying a factor of 0.75 to the baseline values. We update the curvature estimate every step and the preconditioner every 10 steps. For the damping value, we choose the `ASDL` default of $1e-5$ and an exponential moving average over the Kronecker factors, with a factor of β_2.

As presented in the first column of Figure 4.3, K-FAC-reduce and K-FAC-expand both converge almost identically in terms of training steps and both outperform the baseline, as they only

[1]The code is available at `https://bit.ly/algoperf`.

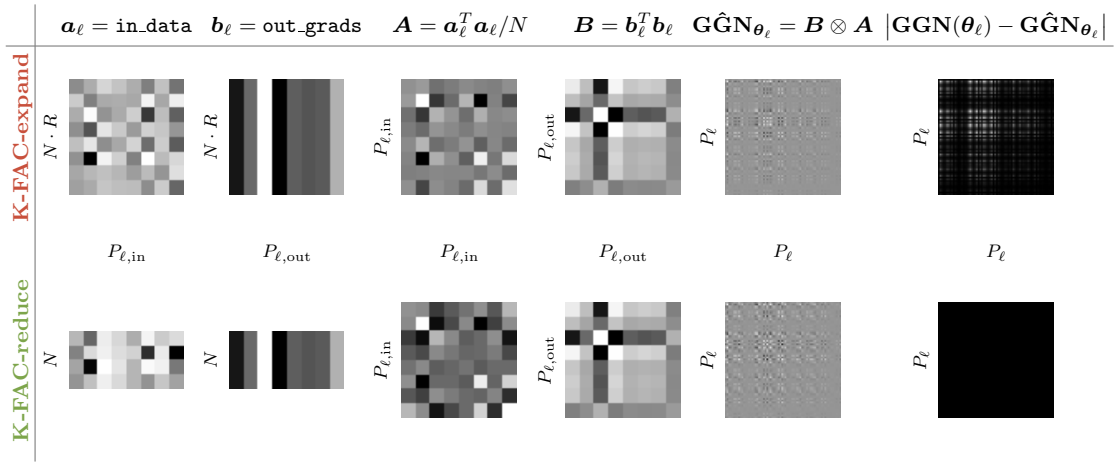

Figure 4.2: Visualization of K-FAC-expand and K-FAC-reduce in the reduce setting. Otherwise, the setting is the same as in Figure 4.1. As we have seen in Section 3.1.2, K-FAC-reduce is exact for the reduce case in this setting and K-FAC-expand is not.

take about 80% of the steps to reach the validation target: NAdamW takes about 117.4k steps, whereas K-FAC-expand takes about 92.6k and K-FAC-reduce takes about 93.7k steps.

However, when we consider the convergence to the target in terms of the wall-clock time, shown in the second column of Figure 4.3, we can see that the baseline is still faster than the K-FAC variants, despite taking more steps; NAdamW runs for about 25 hours, K-FAC-expand for about 50 hours, and K-FAC-reduce for 37 hours. The huge increase in the runtime of the two K-FAC approximations compared to the baseline can be easily explained by the overhead of K-FAC and the gap could potentially be closed by decreasing the update frequency of the curvature estimate and the preconditioner.

Moreover, we can see that while K-FAC-expand and K-FAC-reduce perform similarly in terms of steps, the smaller computational complexity of K-FAC-reduce leads to faster convergence than K-FAC-expand in terms of wall-clock time. The average wall-clock time of a single training step is 0.76 seconds for the baseline NAdamW, 1.94 seconds for K-FAC-expand, and 1.43 seconds for K-FAC-reduce. K-FAC-reduce costs about two times as much as the baseline, which can be explained by the need for a second backward pass for the MC approximation of the Fisher, as described below Equation (3.35). K-FAC-expand costs even more due to the multiplication of larger matrices, which can be seen in Listing 3.1.

When we choose a more aggressive learning rate schedule for the two K-FAC approximations by reducing the factor we multiply the baseline warm-up and expected steps with from 0.75 to 0.6875, K-FAC-expand reaches the target at only 88.2k steps, whereas K-FAC-reduce does not reach the target anymore (it converges at about 76.5% and 77% in two trials with different random seeds).

Since all trials are only run for one random seed (besides the just mentioned K-FAC-reduce run), we cannot conclusively make statements about the difference in performance of the two approximations; it might just be an artifact of the stochasticity in the training process. However, even though these results are highly preliminary, it is still worth pointing out that K-FAC-expand seems to have a slight edge over K-FAC-reduce here, *despite* the fact that the workload is an example of the reduce setting.

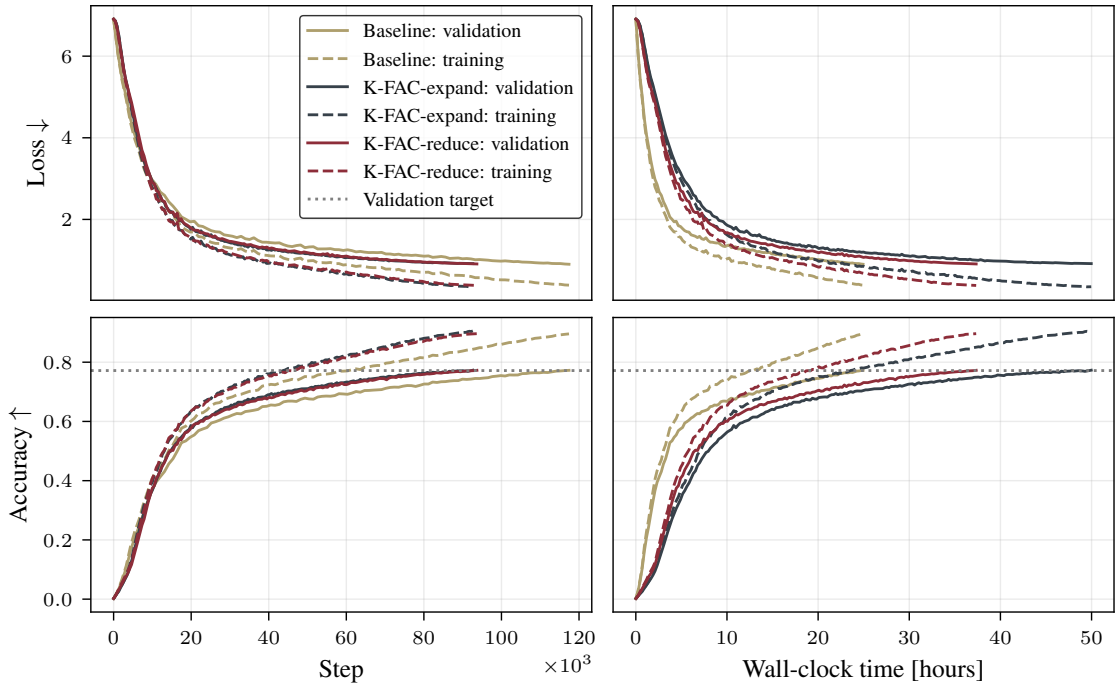

Figure 4.3: Vision Transformer on ImageNet. K-FAC-expand and K-FAC-reduce behave almost identically and both outperform the baseline (NAdamW) in terms of steps to the target – they only need about 80% of the steps. We can see that the baseline is still the fastest algorithm in terms of wall-clock time, due to the overhead of K-FAC. Moreover, K-FAC-reduce's smaller computational complexity compared to K-FAC-expand is also apparent.

Finally, since we only tune the learning rate schedule, as described here, it is possible that tuning more hyperparameters for each approximation individually leads to improved performance in steps and could potentially highlight an even greater discrepancy between the performance of the two approximations – in either approximation's favor.

Chapter 5
Discussion and Conclusion

While the Vision Transformer on ImageNet experiment provides the first proof of concept that both K-FAC approximations could be useful for optimizing Transformer models, we need more experimental data to gain insight into the different behavior of K-FAC-expand and K-FAC-reduce. This includes using both methods on different workloads, e.g. the WMT Transformer and the OGBG GNN workload of the AlgoPerf benchmark. Moreover, more comprehensive tuning of each method has to be performed.

Extrapolating from the ImageNet experiment at hand, one potential outcome could be that both approximations converge comparably in terms of steps. In this case, it could make sense to apply K-FAC-reduce in practice, due to its favorable computational complexity. Moreover, the wall-clock time of K-FAC needs to be reduced to allow it to benefit from the faster convergence in steps in practice. This could be achieved by tuning the update and inversion frequency of the Kronecker factors. Additionally, Osawa et al. (2022) proposes to use K-FAC to improve the utilization of accelerators during pipeline parallelism for training large language models, which are based on Transformers. While they use the slower K-FAC-expand approximation in their work, they can still significantly reduce the (simulated) training time by 50-75%. Therefore, an interesting direction would be to see if we can get the same convergence per step with K-FAC-reduce in this setting, which might be able to further reduce the wall-clock time.

While we have evidence for the potential of K-FAC-expand and K-FAC-reduce to improve training efficiency, there still seem to exist gaps in the understanding of the underlying mechanism. We have derived K-FAC as an approximation to the GGN/Fisher and motivated its potential benefit in optimization by its connection to natural gradient descent and Newton's method. However, it is unclear if the improved convergence in steps can really be explained by the properties of the true GGN or Fisher. Other methods using a structured preconditioner that is not directly connected to the GGN/Fisher, like Shampoo (Gupta et al., 2018), have also shown promise for improving the efficiency of DNN training (Anil et al., 2020). Moreover, it has recently been proposed that K-FAC's potentially improved performance is due to its similarity to a first-order method – at least when the damping value is added to each Kronecker factor individually, which is the default in practice (Benzing, 2022). If this is indeed the case, much of the computational and implementation overhead of K-FAC might not be necessary to get the practical benefits.

Besides the empirical investigation of K-FAC-expand and K-FAC-reduce, it seems useful to try to improve our theoretical understanding of the approximations and their assumptions, e.g. by relating them to the assumptions made for deriving K-FAC for convolutional layers in Grosse & Martens (2016) and for recurrent neural networks in Martens et al. (2018), to create a unifying framework for all types of K-FAC approximations. Additionally, it could be a promising direction to investigate how to efficiently get access to other quantities from the backward pass, using hooks or other tools, besides the one which is currently used in K-FAC. This could enable

more accurate approximations, using weaker assumptions, which are also motivated by the same settings we consider here.

Finally, as mentioned in Chapter 1, there are many more use cases of practical GGN/Fisher approximations than just optimization, e.g. in Bayesian deep learning. The application of K-FAC-expand and K-FAC-reduce to model selection and invariance learning via Laplace approximations of the marginal likelihood (Immer et al., 2021a; 2022) for Transformer and GNN architectures is a promising direction.

Conclusion. We have classified the setting in which linear weight-sharing layers are used within a network based on the point of aggregation over the weight-sharing dimension of size R. This lead to two simple base cases, the expand and the reduce setting. The expand setting is characterized by $N \cdot R$ per-example losses and motivates the K-FAC-expand approximation. This approximation has previously been used to train Transformer models in natural language processing, but was not derived or justified. It is exact for single linear weight-sharing layers or for deep linear networks with weight-sharing with a Gaussian likelihood, just as regular K-FAC is for the corresponding simple models without weight-sharing. The reduce setting is characterized by N per-example losses, which implies that the weight-sharing dimension is reduced during the forward pass, inside of the model. The setting also motivates an approximation, K-FAC-reduce, which is exact in the same settings as K-FAC-expand, assuming that a scaled sum is used as the aggregation function (or another function with also fulfills the condition in Lemma 3.4).

While each approximation is theoretically motivated by a specific setting and is only exact in the simple settings mentioned above, in practice, both approximations can be applied in both cases. If we consider slightly more involved settings than the (deep) linear networks, i.e. a simplified dot-product attention mechanism, both approximations are generally inexact. For architectures used in practice, with additional nonlinearities, it is even less clear how accurate the approximations are. Hence, it does not directly follow which approximation we should use in each setting; it could be that other factors are more important to determine the best choice for a given purpose. Besides the question of the approximation quality, K-FAC-reduce has a lower computational complexity than K-FAC-expand, adding an additional trade-off to consider when deciding which approximation to use. Beyond Transformer models, we also showed how to determine the setting of a GNN for graph classification; this is an example of how to approach K-FAC for architectures with linear weight-sharing layers beyond the more straightforward Transformer cases.

The question of how K-FAC-expand and K-FAC-reduce behave in practice has to be answered empirically. We have provided preliminary evidence for the claim that both approximations can reduce the steps needed to reach a validation target compared to a well-tuned first-order method. The baseline, NAdamW, is still faster in terms of wall-clock time, due to the overhead of K-FAC. Interestingly, both approximations with the same hyperparameters converge practically identically for a Vision Transformer on ImageNet, an instance of the reduce setting, in terms of steps. In terms of wall-clock time, however, the lower computational complexity of K-FAC-reduce becomes apparent. More experiments are necessary to (i) test if both approximations still perform as similarly when tuned individually, (ii) see if the wall-clock time can be reduced enough to be competitive with the baseline, and (iii) determine if the same conclusions also hold for both methods in the expand case, e.g. for a Transformer on a language translation dataset like WMT, and for other model classes like graph neural networks.

While there is plenty of, especially empirical, work to do before we can make more general claims, we hope that this work can contribute to a foundation for K-FAC in the context of linear weight-sharing layers, and open up new approximations and applications for optimization and approximate inference in deep learning.

References

Amari, S. Natural gradient works efficiently in learning. *Neural computation*, 10(2), 1998.

Anil, R., Gupta, V., Koren, T., Regan, K., and Singer, Y. Scalable second order optimization for deep learning. arXiv 2002.09018, 2020.

Ba, L. J., Kiros, J. R., and Hinton, G. E. Layer normalization, 2016.

Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate. In *ICLR*, 2015.

Baird, H. Document image defect models and their uses. In *ICDAR*, 1993.

Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V. F., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., Gülçehre, Ç., Song, H. F., Ballard, A. J., Gilmer, J., Dahl, G. E., Vaswani, A., Allen, K. R., Nash, C., Langston, V., Dyer, C., Heess, N., Wierstra, D., Kohli, P., Botvinick, M. M., Vinyals, O., Li, Y., and Pascanu, R. Relational inductive biases, deep learning, and graph networks. arXiv 1806.01261, 2018.

Bengio, Y., Simard, P., and Frasconi, P. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 1994.

Benzing, F. Gradient descent on neurons and its link to approximate second-order optimization. In *ICML*, 2022.

Bernacchia, A., Lengyel, M., and Hennequin, G. Exact natural gradient in deep linear networks and its application to the nonlinear case. In *NeurIPS*, 2018.

Bojar, O., Buck, C., Federmann, C., Haddow, B., Koehn, P., Leveling, J., Monz, C., Pecina, P., Post, M., Saint-Amand, H., Soricut, R., Specia, L., and Tamchyna, A. s. Findings of the 2014 workshop on statistical machine translation (WMT14). In *Proceedings of the Ninth Workshop on Statistical Machine Translation*, 2014.

Bojar, O. r., Chatterjee, R., Federmann, C., Graham, Y., Haddow, B., Huang, S., Huck, M., Koehn, P., Liu, Q., Logacheva, V., Monz, C., Negri, M., Post, M., Rubino, R., Specia, L., and Turchi, M. Findings of the 2017 conference on machine translation (WMT17). In *Proceedings of the Second Conference on Machine Translation, Volume 2: Shared Task Papers*, 2017.

Botev, A. and Martens, J. KFAC-JAX, 2022. URL http://github.com/deepmind/kfac-jax.

Botev, A., Ritter, H., and Barber, D. Practical Gauss-Newton optimisation for deep learning. In *ICML*, 2017.

References

Boyd, S. and Vandenberghe, L. *Convex optimization*. Cambridge University Press, 2004.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

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

Chaudhari, P., Choromanska, A., Soatto, S., LeCun, Y., Baldassi, C., Borgs, C., Chayes, J. T., Sagun, L., and Zecchina, R. Entropy-SGD: Biasing gradient descent into wide valleys. In *ICLR*, 2017.

Choi, D., Shallue, C. J., Nado, Z., Lee, J., Maddison, C. J., and Dahl, G. E. On empirical comparisons of optimizers for deep learning. arXiv 1910.05446, 2019.

Cordonnier, J., Loukas, A., and Jaggi, M. On the relationship between self-attention and convolutional layers. In *ICLR*, 2020.

Dangel, F., Kunstner, F., and Hennig, P. BackPACK: Packing more into backprop. In *ICLR*, 2020.

Daxberger, E., Kristiadi, A., Immer, A., Eschenhagen, R., Bauer, M., and Hennig, P. Laplace redux–effortless Bayesian deep learning. In *NeurIPS*, 2021.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., and Houlsby, N. An image is worth 16x16 words: Transformers for image recognition at scale. In *ICLR*, 2021.

Duchi, J., Hazan, E., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *JMLR*, 12(61), 2011.

Graves, A. Practical variational inference for neural networks. In *NIPS*, 2011.

Grosse, R. B. and Martens, J. A Kronecker-factored approximate Fisher matrix for convolution layers. In *ICML*, 2016.

Gupta, V., Koren, T., and Singer, Y. Shampoo: Preconditioned stochastic tensor optimization. In *ICML*, 2018.

Heskes, T. On "natural" learning and pruning in multilayered perceptrons. *Neural Computation*, 12(4), 2000.

Hinton, G., Srivastava, N., and Swersky, K. RMSProp: Divide the gradient by a running average of its recent magnitude. *Neural networks for machine learning, Coursera lecture 6e*, 2012a.

Hinton, G. E. Learning translation invariant recognition in a massively parallel networks. In *PARLE Parallel Architectures and Languages Europe*, 1987.

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Improving neural networks by preventing co-adaptation of feature detectors. arXiv 1207.0580, 2012b.

Hochreiter, S. Untersuchungen zu dynamischen neuronalen netzen. *Diplomarbeit*, 1991.

Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., and Leskovec, J. Open graph benchmark: Datasets for machine learning on graphs. In *NeurIPS*, 2020.

Immer, A., Bauer, M., Fortuin, V., Rätsch, G., and Khan, M. E. Scalable marginal likelihood estimation for model selection in deep learning. In *ICML*, 2021a.

Immer, A., Korzepa, M., and Bauer, M. Improving predictions of Bayesian neural nets via local linearization. In *AISTATS*, 2021b.

Immer, A., van der Ouderaa, T. F., Rätsch, G., Fortuin, V., and van der Wilk, M. Invariance learning in deep neural networks with differentiable Laplace approximations. In *NeurIPS*, 2022.

Izadi, M. R., Fang, Y., Stevenson, R., and Lin, L. Optimization of graph neural networks with natural gradient descent. In *IEEE BigData*, 2020.

Jacot, A., Hongler, C., and Gabriel, F. Neural tangent kernel: Convergence and generalization in neural networks. In *NeurIPS*, 2018.

Joshi, C. Transformers are graph neural networks. *The Gradient*, 2020.

Jumper, J., Evans, R., Pritzel, A., Green, T., Figurnov, M., Ronneberger, O., Tunyasuvunakool, K., Bates, R., Žídek, A., Potapenko, A., Bridgland, A., Meyer, C., Kohl, S. A. A., Ballard, A. J., Cowie, A., Romera-Paredes, B., Nikolov, S., Jain, R., Adler, J., Back, T., Petersen, S., Reiman, D., Clancy, E., Zielinski, M., Steinegger, M., Pacholska, M., Berghammer, T., Bodenstein, S., Silver, D., Vinyals, O., Senior, A. W., Kavukcuoglu, K., Kohli, P., and Hassabis, D. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873), 2021.

Khan, M., Nielsen, D., Tangkaratt, V., Lin, W., Gal, Y., and Srivastava, A. Fast and scalable Bayesian deep learning by weight-perturbation in Adam. In *ICML*, 2018.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *ICLR*, 2015.

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13), 2017.

Kristiadi, A., Hein, M., and Hennig, P. Being Bayesian, even just a bit, fixes overconfidence in ReLU networks. In *ICML*, 2020.

Krogh, A. and Hertz, J. A simple weight decay can improve generalization. In *NIPS*, 1991.

Kunstner, F., Balles, L., and Hennig, P. Limitations of the empirical Fisher approximation for natural gradient descent. In *NeurIPS*, 2019.

LeCun, Y., Denker, J. S., and Solla, S. A. Optimal brain damage. In *NIPS*, 1990.

Liu, K., Ding, R., Zou, Z., Wang, L., and Tang, W. A comprehensive study of weight sharing in graph networks for 3d human pose estimation. In *Computer Vision ECCV*, 2020.

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization. In *ICLR*, 2019.

MacKay, D. J. Bayesian interpolation. *Neural computation*, 4(3), 1992a.

MacKay, D. J. The evidence framework applied to classification networks. *Neural Computation*, 4(5), 1992b.

MacKay, D. J. A practical Bayesian framework for backpropagation networks. *Neural Computation*, 4(3), 1992c.

Martens, J. Deep learning via Hessian-free optimization. In *ICML*, 2010.

Martens, J. New insights and perspectives on the natural gradient method. *JMLR*, 21(146), 2014.

Martens, J. and Grosse, R. Optimizing neural networks with Kronecker-factored approximate curvature. In *ICML*, 2015.

Martens, J., Ba, J., and Johnson, M. Kronecker-factored curvature approximations for recurrent neural networks. In *ICLR*, 2018.

MLCommons. Algorithms Working Group, 2022. https://mlcommons.org/en/groups/research-algorithms/, Last accessed: 20.12.2022.

Nesterov, Y. A method for solving the convex programming problem with convergence rate $\mathcal{O}(1/k^2)$. *Proceedings of the USSR Academy of Sciences*, 269, 1983.

Ollivier, Y., Arnold, L., Auger, A., and Hansen, N. Information-geometric optimization algorithms: A unifying picture via invariance principles. *JMLR*, 18(18), 2017.

Osawa, K. ASDL: Automatic second-order differentiation (for Fisher, gradient covariance, Hessian, Jacobian, and kernel) library. https://github.com/kazukiosawa/asdl, 2021.

Osawa, K., Swaroop, S., Khan, M. E. E., Jain, A., Eschenhagen, R., Turner, R. E., and Yokota, R. Practical deep learning with Bayesian principles. In *NeurIPS*, 2019.

Osawa, K., Li, S., and Hoefler, T. PipeFisher: Efficient training of large language models using pipelining and Fisher information matrices. arXiv 2211.14133, 2022.

Pan, P., Swaroop, S., Immer, A., Eschenhagen, R., Turner, R. E., and Khan, M. E. Continual deep learning by functional regularisation of memorable past. In *NeurIPS*, 2020.

Parmar, N., Vaswani, A., Uszkoreit, J., Kaiser, L., Shazeer, N., and Ku, A. Image Transformer. In *ICML*, 2018.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. PyTorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

Pauloski, J. G., Huang, Q., Huang, L., Venkataraman, S., Chard, K., Foster, I. T., and Zhang, Z. KAISA: an adaptive second-order optimizer framework for deep neural networks. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC21)*, 2021.

Polyak, B. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5), 1964.

Ramesh, A., Dhariwal, P., Nichol, A., Chu, C., and Chen, M. Hierarchical text-conditional image generation with CLIP latents. arXiv 2204.06125, 2022.

Ren, Y. and Goldfarb, D. Tensor normal training for deep learning models. In *NeurIPS*, 2021.

Ritter, H., Botev, A., and Barber, D. A scalable Laplace approximation for neural networks. In *ICLR*, 2018.

Robbins, H. and Monro, S. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951.

Rombach, R., Blattmann, A., Lorenz, D., Esser, P., and Ommer, B. High-resolution image synthesis with latent diffusion models. arXiv 2112.10752, 2021.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. ImageNet large scale visual recognition challenge. *IJCV*, 115, 2015.

Saxe, A. M., McClelland, J. L., and Ganguli, S. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. In *ICLR*, 2014.

Schmidt, R. M., Schneider, F., and Hennig, P. Descending through a crowded valley - benchmarking deep learning optimizers. In *ICML*, 2021.

Schraudolph, N. N. Fast curvature matrix-vector products for second-order gradient descent. *Neural computation*, 14(7), 2002.

Sen, P., Namata, G., Bilgic, M., Getoor, L., Galligher, B., and Eliassi-Rad, T. Collective classification in network data. *AI Magazine*, 29(3), 2008.

Singh, S. P. and Alistarh, D. Woodfisher: Efficient second-order approximation for neural network compression. In *NeurIPS*, 2020.

Strubell, E., Ganesh, A., and McCallum, A. Energy and policy considerations for modern deep learning research. In *AAAI*, 2020.

Tsai, Y. H., Bai, S., Yamada, M., Morency, L., and Salakhutdinov, R. Transformer dissection: An unified understanding for Transformer's attention via the lens of kernel. In *EMNLP*, 2019.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *NIPS*, 2017.

Wang, Y. Fisher scoring: An interpolation family and its Monte Carlo implementations. *Comput. Stat. Data Anal.*, 54(7), 2010.

Wong, S. C., Gatt, A., Stamatescu, V., and McDonnell, M. D. Understanding data augmentation for classification: When to warp? In *DICTA*, 2016.

Zhang, G., Sun, S., Duvenaud, D., and Grosse, R. Noisy natural gradient as variational inference. In *ICML*, 2018.

Zhang, G., Li, L., Nado, Z., Martens, J., Sachdeva, S., Dahl, G. E., Shallue, C. J., and Grosse, R. B. Which algorithmic choices matter at which batch sizes? Insights from a noisy quadratic model. In *NeurIPS*, 2019.