Acknowledgements

The authors thank Kazuki Osawa for fruitful discussions and support with using and extending the ASDL library. The authors also thank Wu Lin for pointing out a mistake in the original equation of the K-FAC-expand approximation. Finally, the authors thank Felix Dangel for helpful comments on this manuscript. Runa Eschenhagen is supported by ARM and the Cambridge Trust. Alexander Immer is funded by the Max Planck ETH Center for Learning Systems (CLS). Richard E. Turner is supported by Google, Amazon, ARM, Improbable and EPSRC grant EP/T005386/1. Frank Schneider and Philipp Hennig are funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) in the frame of the priority programme SPP 2298 "Theoretical Foundations of Deep Learning" - HE 7114/5-1, as well as the DFG Cluster of Excellence "Machine Learning - New Perspectives for Science", EXC 2064/1, project number 390727645; the German Federal Ministry of Education and Research (BMBF) through the Tübingen AI Center (FKZ: 01IS18039A); and funds from the Ministry of Science, Research and Arts of the State of Baden-Württemberg.

References

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

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.

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.

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.

Bi, K., Xie, L., Zhang, H., Chen, X., Gu, X., and Tian, Q. Pangu-weather: A 3d high-resolution model for fast and accurate global weather forecast. arXiv 2211.02556, 2022.

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.

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.

Dahl, G. E., Schneider, F., Nado, Z., Agarwal, N., Sastry, C. S., Hennig, P., Medapati, S., Eschenhagen, R., Kasimbeg, P., Suo, D., Bae, J., Gilmer, J., Peirson, A. L., Khan, B., Anil, R., Rabbat, M., Krishnan, S., Snider, D., Amid, E., Chen, K., Maddison, C. J., Vasudev, R., Badura, M., Garg, A., and Mattson, P. Benchmarking neural network training algorithms. arXiv 2306.07179, 2023.

Dangel, F. Convolutions through the lens of tensor networks. arXiv 2307.02275, 2023.

11

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.

Goldfarb, D., Ren, Y., and Bahamou, A. Practical Quasi-Newton methods for training deep neural networks. In *NeurIPS*, 2020.

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

Grosse, R., Bae, J., Anil, C., Elhage, N., Tamkin, A., Tajdini, A., Steiner, B., Li, D., Durmus, E., Perez, E., Hubinger, E., Lukošiūtė, K., Nguyen, K., Joseph, N., McCandlish, S., Kaplan, J., and Bowman, S. R. Studying large language model generalization with influence functions. arXiv 2308.03296, 2023.

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

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*, 2021.

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.

Karpathy, A. nanoGPT, 2023. URL https://github.com/karpathy/nanoGPT.

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.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2016.

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

Lin, W., Duruisseaux, V., Leok, M., Nielsen, F., Khan, M. E., and Schmidt, M. Simplifying momentum-based positive-definite submanifold optimization with applications to deep learning. In *ICML*, 2023.

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.

12

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

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

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.

Nan, L., Radev, D. R., Zhang, R., Rau, A., Sivaprasad, A., Hsieh, C., Tang, X., Vyas, A., Verma, N., Krishna, P., Liu, Y., Irwanto, N., Pan, J., Rahman, F., Zaidi, A., Mutuma, M., Tarabar, Y., Gupta, A., Yu, T., Tan, Y. C., Lin, X. V., Xiong, C., Socher, R., and Rajani, N. F. DART: open-domain structured data record to text generation. In *NAACL-HLT*, 2021.

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

Osawa, K., Ishikawa, S., Yokota, R., Li, S., and Hoefler, T. ASDL: A unified interface for gradient preconditioning in PyTorch. arXiv 2305.04684, 2023.

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.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. 2019.

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.

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.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. The graph neural network model. *IEEE Transactions on Neural Networks*, 20, 2009.

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.

Tang, Z., Jiang, F., Gong, M., Li, H., Wu, Y., Yu, F., Wang, Z., and Wang, M. SKFAC: Training neural networks with faster Kronecker-factored approximate curvature. In *CVPR*, 2021.

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.

13

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

Yang, G. Tensor programs i: Wide feedforward or recurrent neural networks of any architecture are gaussian processes. In *NeurIPS*, 2019.

Yang, M., Xu, D., Wen, Z., Chen, M., and Xu, P. Sketch-based empirical natural gradient methods for deep learning. *Journal of Scientific Computing*, 92(3), 2022.

Yang, M., Xu, D., Cui, Q., Wen, Z., and Xu, P. An efficient Fisher matrix approximation method for large-scale neural network optimization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(5), 2023.

Yao, Z., Gholami, A., Shen, S., Mustafa, M., Keutzer, K., and Mahoney, M. W. ADAHESSIAN: an adaptive second order optimizer for machine learning. In *AAAI*, 2021.

Zagoruyko, S. and Komodakis, N. Wide residual networks. In *BMVC*, 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*, 2019a.

Zhang, H., Dauphin, Y. N., and Ma, T. Fixup initialization: Residual learning without normalization. In *ICLR*, 2019b.

14

A   Weight-sharing in graph neural networks

In this section, we expand on the third example in Section 2.2 on graph neural networks (GNNs) and show how they use linear layers with weight-sharing.

Graph convolutional network for node classification. A popular type of GNN is called graph convolutional network (Kipf & Welling, 2016, GCN). A GCN defines a convolution operation on graph structures, by repeatedly aggregating feature information over the neighbourhood 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 utilise a slightly different type of weight-sharing, which will become apparent in Equation (12). 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 Appendix B.3.2; 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 ℓ-th GCN layer is defined as

$$f_{\boldsymbol{\theta}_\ell}(\boldsymbol{X}) = \phi(\hat{\boldsymbol{C}} \boldsymbol{X} \boldsymbol{W}_\ell^\intercal) \tag{11}$$

which is identical to a regular dense linear layer from Section 2.2, 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 normalised 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

$$\tilde{\boldsymbol{x}}_n := \sum_{j=1}^{N} \hat{\boldsymbol{C}}_{nj} \boldsymbol{x}_j = \sum_{j \in \mathcal{N}(n)} \hat{\boldsymbol{C}}_{nj} \boldsymbol{x}_j, \tag{12}$$

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

$$f_{\boldsymbol{\theta}_\ell}(\tilde{\boldsymbol{x}}_n) = \phi(\boldsymbol{W}_\ell \tilde{\boldsymbol{x}}_n), \tag{13}$$

where $\mathcal{N}(n) := \{j \in \{1, \ldots, N\} | \hat{\boldsymbol{C}}_{nj} \neq 0\}$ is the neighbourhood of the node with index n. Notably, the forward pass for a single node $\boldsymbol{x}_n$ depends on its neighbourhood, i.e. we cannot express the forward pass for the node without access to the feature information of the nodes in its neighbourhood $\mathcal{N}(n)$. 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 neighbourhood, over which the weights are shared, *before* the matrix-vector product. Hence, in contrast to the GraphNetwork introduced in the next paragraph, this model does not require special consideration when applying K-FAC (c.f. setting (i) in Section 3.1).

GraphNetwork for graph classification. One more general formulation of a GNN is an instance of the GraphNetwork introduced in Battaglia et al. (2018). The GraphNetwork 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 i-th 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 $n = 1, \ldots, N$. 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 $n \in \{1, \ldots, N\}$.

15

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

$$
\begin{aligned}
\boldsymbol{X}_n^E &\leftarrow \phi^E(\boldsymbol{X}_n^E, \boldsymbol{X}_{n,\boldsymbol{r}_n}^V, \boldsymbol{X}_{n,\boldsymbol{s}_n}^V, \boldsymbol{x}_n^u) \\
\boldsymbol{X}_n^V &\leftarrow \phi^V(\boldsymbol{X}_n^V, \tilde{\boldsymbol{X}}_n^E, \boldsymbol{x}_n^u) \\
\boldsymbol{x}_n^u &\leftarrow \phi^u(\boldsymbol{x}_n^u, \bar{\boldsymbol{X}}_n^V, \bar{\boldsymbol{X}}_n^E),
\end{aligned}
\tag{14}
$$

and three permutation-invariant aggregation functions ρ

$$
\begin{aligned}
\tilde{\boldsymbol{X}}_n^E &\leftarrow \rho^{E \to V}(\boldsymbol{X}_n^E) \\
\bar{\boldsymbol{X}}_n^E &\leftarrow \rho^{E \to u}(\boldsymbol{X}_n^E) \\
\bar{\boldsymbol{X}}_n^V &\leftarrow \rho^{V \to u}(\boldsymbol{X}_n^V).
\end{aligned}
\tag{15}
$$

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

One forward pass through a GraphNetwork 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 (e.g. the ogbg-molpcba dataset used in Section 4.2). We specifically consider a GraphNetwork instance with simple MLPs for all update functions ϕ and an element-wise sum for the aggregation functions ρ. Moreover, multiple of these GraphNetwork 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 GraphNetwork block.

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

$$
\begin{aligned}
\phi^E(\boldsymbol{X}_n^E, \boldsymbol{X}_{n,\boldsymbol{r}_n}^V, \boldsymbol{X}_{n,\boldsymbol{s}_n}^V, \boldsymbol{x}_n^u) &:= \mathrm{concat}(\boldsymbol{X}_n^E, \boldsymbol{X}_{n,\boldsymbol{r}_n}^V, \boldsymbol{X}_{n,\boldsymbol{s}_n}^V, \mathrm{repeat}_{N_n^E}(\boldsymbol{x}_n^u))\boldsymbol{W}^{E^\mathsf{T}} \\
\phi^V(\boldsymbol{X}_n^V, \tilde{\boldsymbol{X}}_n^E, \boldsymbol{x}_n^u) &:= \mathrm{concat}(\boldsymbol{X}_n^V, \tilde{\boldsymbol{X}}_n^E, \mathrm{repeat}_{N_n^V}(\boldsymbol{x}_n^u))\boldsymbol{W}^{V^\mathsf{T}} \\
\phi^u(\boldsymbol{x}_n^u, \bar{\boldsymbol{X}}_n^V, \bar{\boldsymbol{X}}_n^E) &:= \boldsymbol{W}^u \mathrm{concat}(\boldsymbol{x}_n^u, \bar{\boldsymbol{X}}_n^V, \bar{\boldsymbol{X}}_n^E)
\end{aligned}
\tag{16}
$$

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 normalisation (Ba et al., 2016), and dropout (Hinton et al., 2012). 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.

Most importantly, we can now observe that this type of GNN shares its weights over each graph's edges and nodes: just as for the attention or convolution operations described in Section 2.2, 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 n-th input. Also, note that the update function ϕ^u used to update the global features $\boldsymbol{x}_n^u$ does not use any weight-sharing, as there is just one feature vector per data point. We have specifically introduced this notation of the inputs to show that the edge and node feature update functions are exactly examples of the concept of a linear weight-sharing layer, as introduced in Section 2.2. This might not be immediately obvious when only looking at the original notation used in Battaglia et al. (2018); therefore, this can be seen as an instructive example for expressing a neural network architecture in terms of linear weight-sharing layers. Consequently, our framework for K-FAC can directly be applied to this architecture, as we show in Appendix B.3.2.

16

B Extended derivation and discussion of K-FAC-expand and K-FAC-reduce

B.1 Background: K-FAC for regular linear layers

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. The Fisher information matrix is defined as[6]

$$\boldsymbol{F}(\boldsymbol{\theta}) = -\sum_{n=1}^{N} \mathbb{E}_{\boldsymbol{y} \sim p(\boldsymbol{y}|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))}[\nabla_{\boldsymbol{\theta}}^2 \log p(\boldsymbol{y}|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))]$$

$$= \sum_{n=1}^{N} \mathbb{E}_{\boldsymbol{y} \sim p(\boldsymbol{y}|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))}[\nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{y}|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))(\nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{y}|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)))^{\mathsf{T}}]. \tag{17}$$

Notably, the labels $\boldsymbol{y}$ are samples from the predictive distribution of the model and are not the empirical labels from the data. Replacing $\boldsymbol{y}$ with $\boldsymbol{y}_n$ leads to the *empirical* Fisher (EF), which is simply the uncentered covariance of the empirical gradient. While it is commonly used as a replacement for the Fisher (Chaudhari et al., 2017; Graves, 2011; Kingma & Ba, 2015), it can give rise to very different downstream behaviour when used for optimisation (Kunstner et al., 2019).

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

$$\boldsymbol{F}(\boldsymbol{\theta}) \approx \operatorname{diag}(\boldsymbol{F}(\boldsymbol{\theta}_1), \ldots, \boldsymbol{F}(\boldsymbol{\theta}_\ell), \ldots, \boldsymbol{F}(\boldsymbol{\theta}_L)) \in \mathbb{R}^{P \times P}, \tag{18}$$

where $\boldsymbol{F}(\boldsymbol{\theta}_\ell) \in \mathbb{R}^{P_\ell \times P_\ell}$ and $\operatorname{diag}(\cdot, \ldots, \cdot)$ 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 n-th 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 ℓ-th 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 ℓ-th layer can be written as $\nabla_{\boldsymbol{W}_\ell} \mathcal{L}(\boldsymbol{y}, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) = \nabla_{\boldsymbol{s}_{\ell,n}} \mathcal{L}(\boldsymbol{y}, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \boldsymbol{a}_{\ell,n}^{\mathsf{T}} =: \boldsymbol{g}_{\ell,n} \boldsymbol{a}_{\ell,n}^{\mathsf{T}} \in \mathbb{R}^{P_{\ell,\text{out}} \times P_{\ell,\text{in}}}$.

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.

$$\boldsymbol{F}(\boldsymbol{\theta}_\ell) = \sum_{n=1}^{N} \mathbb{E}_{\boldsymbol{y} \sim p(\boldsymbol{y}|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))}[\operatorname{vec}(\nabla_{\boldsymbol{W}_\ell} \mathcal{L}(\boldsymbol{y}, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)))\operatorname{vec}(\nabla_{\boldsymbol{W}_\ell} \mathcal{L}(\boldsymbol{y}, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)))^{\mathsf{T}}] \tag{19a}$$

$$= \sum_{n=1}^{N} \mathbb{E}_{\boldsymbol{y} \sim p(\boldsymbol{y}|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))}[\operatorname{vec}(\boldsymbol{g}_{\ell,n} \boldsymbol{a}_{\ell,n}^{\mathsf{T}})\operatorname{vec}(\boldsymbol{g}_{\ell,n} \boldsymbol{a}_{\ell,n}^{\mathsf{T}})^{\mathsf{T}}] \tag{19b}$$

$$= \sum_{n=1}^{N} \mathbb{E}_{\boldsymbol{y} \sim p(\boldsymbol{y}|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))}[(\boldsymbol{a}_{\ell,n} \otimes \boldsymbol{g}_{\ell,n})(\boldsymbol{a}_{\ell,n}^{\mathsf{T}} \otimes \boldsymbol{g}_{\ell,n}^{\mathsf{T}})] \tag{19c}$$

$$= \sum_{n=1}^{N} \mathbb{E}_{\boldsymbol{y} \sim p(\boldsymbol{y}|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))}[\boldsymbol{a}_{\ell,n} \boldsymbol{a}_{\ell,n}^{\mathsf{T}} \otimes \boldsymbol{g}_{\ell,n} \boldsymbol{g}_{\ell,n}^{\mathsf{T}}] \tag{19d}$$

$$\approx \underbrace{\left[\frac{1}{N}\sum_{n=1}^{N} \boldsymbol{a}_{\ell,n} \boldsymbol{a}_{\ell,n}^{\mathsf{T}}\right]}_{=:\boldsymbol{A}_\ell} \otimes \underbrace{\left[\sum_{n=1}^{N} \mathbb{E}_{\boldsymbol{y} \sim p(\boldsymbol{y}|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))}[\boldsymbol{g}_{\ell,n} \boldsymbol{g}_{\ell,n}^{\mathsf{T}}]\right]}_{=:\boldsymbol{G}_\ell}, \tag{19e}$$

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): $\operatorname{vec}(\boldsymbol{ABC}) = (\boldsymbol{C}^{\mathsf{T}} \otimes \boldsymbol{A})\operatorname{vec}(\boldsymbol{B})$ and $(\boldsymbol{A} \otimes \boldsymbol{B})^{\mathsf{T}} = \boldsymbol{A}^{\mathsf{T}} \otimes \boldsymbol{B}^{\mathsf{T}}$ for Equation (19c), and $(\boldsymbol{A} \otimes \boldsymbol{B})(\boldsymbol{C} \otimes \boldsymbol{D}) = \boldsymbol{AC} \otimes \boldsymbol{BD}$ for Equation (19d).

---

[6]More generally, it is defined with an expectation over $\boldsymbol{x} \sim p(\boldsymbol{x})$ as well.

17

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 $\mathcal{O}(P_{\ell,\mathrm{in}}^2 P_{\ell,\mathrm{out}}^2)$ to $\mathcal{O}(P_{\ell,\mathrm{in}}^2 + P_{\ell,\mathrm{out}}^2)$ and the computational complexity of the preconditioning of the gradient with the approximate Fisher from $\mathcal{O}(P_{\ell,\mathrm{in}}^3 P_{\ell,\mathrm{out}}^3)$ to $\mathcal{O}(P_{\ell,\mathrm{in}}^3 + P_{\ell,\mathrm{out}}^3)$, since

$$
\begin{aligned}
\boldsymbol{F}(\boldsymbol{\theta}_\ell)^{-1}\boldsymbol{g}(\boldsymbol{\theta}_\ell) &\approx (\boldsymbol{A}_\ell \otimes \boldsymbol{G}_\ell)^{-1}\boldsymbol{g}(\boldsymbol{\theta}_\ell) \\
&= \mathrm{vec}\left(\boldsymbol{G}_\ell^{-1}\nabla_{\boldsymbol{W}_\ell}\mathcal{L}(\boldsymbol{y}, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))\boldsymbol{A}_\ell^{-1}\right)
\end{aligned}
\tag{20}
$$

with $\boldsymbol{g}(\boldsymbol{\theta}_\ell) = \mathrm{vec}\left(\nabla_{\boldsymbol{W}_\ell}\mathcal{L}(\boldsymbol{y}, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))\right)$ 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 (5) for many common loss functions, as we have learned Section 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 ℓ-th layer and $\boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) := \boldsymbol{H}_{f_{\boldsymbol{\theta}}}\mathcal{L}(\boldsymbol{y}_n, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \in \mathbb{R}^{C \times C}$ as the Hessian of the loss w.r.t. the model outputs. Now we can write $\boldsymbol{s}_{\ell,n} = \boldsymbol{W}_\ell\boldsymbol{a}_{\ell,n} = (\boldsymbol{a}_{\ell,n}^\mathsf{T} \otimes \mathbf{I}_{P_{\ell,\mathrm{out}}})\mathrm{vec}(\boldsymbol{W}_\ell)$ and with this we have $\boldsymbol{J}_{\boldsymbol{\theta}_\ell}\boldsymbol{s}_{\ell,n} = \boldsymbol{a}_{\ell,n}^\mathsf{T} \otimes \mathbf{I}_{P_{\ell,\mathrm{out}}}$. Additionally, by defining $\boldsymbol{b}_{\ell,n} := \boldsymbol{J}_{\boldsymbol{s}_{\ell,n}} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)^\mathsf{T} \in \mathbb{R}^{P_{\ell,\mathrm{out}} \times C}$ as the transposed Jacobian of the model outputs w.r.t. to the pre-activations of the ℓ-th layer, we note that $\boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{x}_n)^\mathsf{T} = (\boldsymbol{a}_{\ell,n}^\mathsf{T} \otimes \mathbf{I}_{P_{\ell,\mathrm{out}}})^\mathsf{T}\boldsymbol{b}_{\ell,n} = \boldsymbol{a}_{\ell,n} \otimes \boldsymbol{b}_{\ell,n}$.

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

$$
\begin{aligned}
\mathbf{GGN}(\boldsymbol{\theta}_\ell) &= \sum_{n=1}^{N} \boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{x}_n)^\mathsf{T}\boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))\boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{x}_n) \\
&= \sum_{n=1}^{N} (\boldsymbol{a}_{\ell,n} \otimes \boldsymbol{b}_{\ell,n})\boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))(\boldsymbol{a}_{\ell,n} \otimes \boldsymbol{b}_{\ell,n})^\mathsf{T} \\
&= \sum_{n=1}^{N} (\boldsymbol{a}_{\ell,n}\boldsymbol{a}_{\ell,n}^\mathsf{T}) \otimes (\boldsymbol{b}_{\ell,n}\boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))\boldsymbol{b}_{\ell,n}^\mathsf{T}) \\
&\approx \underbrace{\left[\frac{1}{N}\sum_{n=1}^{N} \boldsymbol{a}_{\ell,n}\boldsymbol{a}_{\ell,n}^\mathsf{T}\right]}_{=: \boldsymbol{A}_\ell} \otimes \underbrace{\left[\sum_{n=1}^{N} \boldsymbol{b}_{\ell,n}\boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))\boldsymbol{b}_{\ell,n}^\mathsf{T}\right]}_{=: \boldsymbol{B}_\ell}.
\end{aligned}
\tag{21}
$$

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 $\boldsymbol{y}$, while still being equivalent to the Fisher for common loss functions (c.f. Section 2.3). Moreover, in our context, it will be useful to have the Jacobians $\boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{x}_n)$ separate from the loss; therefore, we will only explicitly write our results for the GGN.

B.2 K-FAC for linear weight-sharing layers

B.2.1 The expand setting and K-FAC-expand

The expand 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

$$
\mathcal{L}_{\mathrm{expand}}(f_{\boldsymbol{\theta}}, \mathcal{D}) := -\sum_{n=1}^{N}\sum_{r=1}^{R} \log p(\boldsymbol{y}_{n,r}|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r),
\tag{22}
$$

where $f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r$ is the r-th row of the model output $f_{\boldsymbol{\theta}}(\boldsymbol{x}_n) \in \mathbb{R}^{R \times C}$ and $\boldsymbol{y}_{n,r}$ is the r-th row of the label $\boldsymbol{Y}_n \in \mathbb{R}^{R \times C}$. A typical example of this type of loss function is language translation, where N is the number of training examples and R is the sequence length.

18

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 ℓ-th layer has this additional dimension, i.e. $\boldsymbol{A}_{\ell,n} \in \mathbb{R}^{R \times D}$. This obviously does not exclude the case where $\boldsymbol{x}_n$ already has this weight-sharing dimension, e.g. a sequence of tokens in translation tasks.

We can express the Jacobian of the r-th 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

$$(\boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{x}_n)_r)_{ij} = \sum_{m=1}^{R} \sum_{p=1}^{P_{\ell,\text{out}}} \frac{\partial f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_{ri}}{\partial \boldsymbol{S}_{\ell,n,mp}} \frac{\partial \boldsymbol{S}_{\ell,n,mp}}{\partial \boldsymbol{\theta}_{\ell,j}} \tag{23}$$

or in matrix form

$$\boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{x}_n)_r = \sum_{m=1}^{R} \boldsymbol{J}_{\boldsymbol{s}_{\ell,n,m}} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r \boldsymbol{J}_{\boldsymbol{\theta}_\ell} \boldsymbol{s}_{\ell,n,m}. \tag{24}$$

Since the weights $\boldsymbol{\theta}_\ell$ are shared across the weight-sharing dimension of size R, we can write the r-th row of $\boldsymbol{S}_{\ell,n}$ as $\boldsymbol{s}_{\ell,n,r} = \boldsymbol{W}_\ell \boldsymbol{a}_{\ell,n,r}$ and we have $\boldsymbol{J}_{\boldsymbol{\theta}_\ell} \boldsymbol{s}_{\ell,n,r} = \boldsymbol{a}_{\ell,n,r}^\mathsf{T} \otimes \boldsymbol{I}_{P_{\ell,\text{out}}}$, as for regular K-FAC (c.f. Appendix B.1). We denote $\boldsymbol{b}_{\ell,n,r,k} := \boldsymbol{J}_{\boldsymbol{s}_{\ell,n,k}} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r^\mathsf{T}$. Hence, we have

$$\begin{aligned}
\boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{x}_n)_r^\mathsf{T} &= \left(\sum_{m=1}^{R} \boldsymbol{b}_{\ell,n,r,m}^\mathsf{T} (\boldsymbol{a}_{\ell,n,m}^\mathsf{T} \otimes \boldsymbol{I}_{P_{\ell,\text{out}}}) \right)^\mathsf{T} \\
&= \sum_{m=1}^{R} \boldsymbol{a}_{\ell,n,m} \otimes \boldsymbol{b}_{\ell,n,r,m}.
\end{aligned} \tag{25}$$

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 deriving the approximation in the same way as we would with N examples (c.f. Appendix B.1),

$$\begin{aligned}
\mathbf{GGN}(\boldsymbol{\theta}_\ell) &= \sum_{n=1}^{N} \sum_{r=1}^{R} \boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{x}_n)_r^\mathsf{T} \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r) \boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{x}_n)_r \\
&= \sum_{n=1}^{N} \sum_{r=1}^{R} \left(\sum_{m=1}^{R} \boldsymbol{a}_{\ell,n,m} \otimes \boldsymbol{b}_{\ell,n,r,m} \right) \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r) \left(\sum_{m'=1}^{R} \boldsymbol{a}_{\ell,n,m'}^\mathsf{T} \otimes \boldsymbol{b}_{\ell,n,r,m'}^\mathsf{T} \right).
\end{aligned} \tag{26}$$

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 approximate each $\boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{x}_n)_r$ with a Kronecker product of sums, as the K-FAC approximation does for the sum of the N data points, 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 contrast, what can be implemented in practice without additional backward passes (c.f. Appendix B.4) and what has been used for convolutional neural networks (Grosse & Martens, 2016) and language transformers (Grosse et al., 2023; Osawa et al., 2022; Pauloski et al., 2021; Zhang et al., 2019a) is

$$\begin{aligned}
\mathbf{GGN}(\boldsymbol{\theta}_\ell) &\approx \sum_{n=1}^{N} \sum_{r=1}^{R} \sum_{m=1}^{R} (\boldsymbol{a}_{\ell,n,m} \otimes \boldsymbol{b}_{\ell,n,r,m}) \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r) (\boldsymbol{a}_{\ell,n,m}^\mathsf{T} \otimes \boldsymbol{b}_{\ell,n,r,m}^\mathsf{T}) \\
&= \sum_{n=1}^{N} \sum_{m=1}^{R} (\boldsymbol{a}_{\ell,n,m} \boldsymbol{a}_{\ell,n,m}^\mathsf{T}) \otimes \left(\sum_{r=1}^{R} \boldsymbol{b}_{\ell,n,r,m} \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r) \boldsymbol{b}_{\ell,n,r,m}^\mathsf{T} \right),
\end{aligned} \tag{27}$$

where we ignore all the terms with $m \neq m'$, which allows us to express each of the $N \cdot R$ terms as a Kronecker product.[7] Consequently, we can apply the regular K-FAC approximation over $N \cdot R$ terms instead of just N terms as usual. We call the resulting approximation *K-FAC-expand*:

---

[7]However, the authors of the work on transformers do not discuss this extension of K-FAC at all; although Zhang et al. (2018) do explicitly discuss the tying of the embedding and linear output layer weights.

19

<div style="border:1px solid">

K-FAC-expand

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

$$(28)$$

</div>

There is one simple case where the exact expression in Equation (26) is identical to the approximation in Equation (27). When $\boldsymbol{b}_{\ell,n,r,m} = \mathbf{0}$ for all $r \neq m$, both expression are equivalent to

$$\sum_{n=1}^{N}\sum_{r=1}^{R}\left(\boldsymbol{a}_{\ell,n,r}\otimes\boldsymbol{b}_{\ell,n,r,r}\right)\boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r)\left(\boldsymbol{a}_{\ell,n,r}^{\mathsf{T}}\otimes\boldsymbol{b}_{\ell,n,r,r}^{\mathsf{T}}\right). \qquad (29)$$

With other words, when $f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r$ is independent of all pre-activations $\boldsymbol{s}_{\ell,n,m}$ with $m \neq r$ the two expressions coincide. This does not even hold for simplistic transformer models since the self-attention mechanism (Section 2.2) in transformers directly correlates elements across the weight-sharing dimension – we discuss this in more detail in Appendix B.3.1. Alternatively, we could also scale Equation (27) by R, which leads to a scaling of $1/N$ instead of $1/NR$ in Equation (28). This might be a better approximation when $\boldsymbol{b}_{\ell,n,r,m} \neq \mathbf{0}$ for $r \neq m$ and is also what has been used by previous work on transformers (Grosse et al., 2023; Osawa et al., 2022; Pauloski et al., 2021; Zhang et al., 2019a). However, we choose to use the scaling in Equation (28) since the condition above holds for networks that simply stack multiple linear weight-sharing layers; this allows us to show that the approximation in Equation (28) 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 $n \in \{1,\ldots,N\}$ and $r \in \{1,\ldots,R\}$. We do not state these types of conditions explicitly from now on.

Lemma 3 (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 $\boldsymbol{b}_{\ell,n,m,m}\boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_m)\boldsymbol{b}_{\ell,n,m,m}^{\mathsf{T}} = \boldsymbol{C}_\ell$ for all $n \in \{1,\ldots,N\}$ and $m \in \{1,\ldots,R\}$, then the K-FAC approximation in Equation (7) is equal to the exact GGN/Fisher of the ℓ-th 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 (26) and the first line of Equation (27) both simplify to Equation (29). Hence, we can directly show that the second and third approximation in Equation (27) equal the exact expression for the GGN of layer ℓ from there. We have

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

$$= \left(\frac{1}{NR}\sum_{n=1}^{N}\sum_{r=1}^{R}\boldsymbol{a}_{\ell,n,r}\boldsymbol{a}_{\ell,n,r}^{\mathsf{T}}\right) \otimes (NR\boldsymbol{C}_\ell)$$

$$= \left(\sum_{n=1}^{N}\sum_{r=1}^{R}\boldsymbol{a}_{\ell,n,r}\boldsymbol{a}_{\ell,n,r}^{\mathsf{T}}\right) \otimes \boldsymbol{C}_\ell$$

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

$$(30)$$

20

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

Leveraging this simple insight, we can provide an example of a single layer where the assumptions of Lemma 3 are fulfilled.

Proposition 4 (**Exactness of K-FAC-expand for single linear layer 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 $\boldsymbol{b}_{\ell,n,r,m} = \boldsymbol{0}$ for $r \neq m$. Moreover, we have $\boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_r) = \boldsymbol{\Sigma}^{-1}$ and $\boldsymbol{b}_{\ell,n,m,m} = \mathbf{I}_C$ ($P_{\ell,\text{out}} = C$ for a single layer). Hence,

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

for all $n \in \{1, \dots, N\}$ and $m \in \{1, \dots, R\}$. Therefore, the desired result follows from Lemma 3. $\qquad\square$

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

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{W}_L \dots \boldsymbol{W}_\ell \dots \boldsymbol{W}_1 \boldsymbol{x} = \boldsymbol{W}\boldsymbol{x}, \tag{31}$$

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 $\boldsymbol{W}_1 \in \mathbb{R}^{P_{1,\text{out}} \times D}$. Decomposing a single weight matrix $\boldsymbol{W}$ into L separate ones is a common framework for theoretical analysis since it creates nonlinear training dynamics for gradient-based training algorithms, while still having analytical solutions (Bernacchia et al., 2018; Saxe et al., 2014). We adopt the notation of Bernacchia et al. (2018) and define

$$\boldsymbol{W}_\ell^a := \boldsymbol{W}_L \dots \boldsymbol{W}_{\ell+1} \tag{32}$$

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

$$\boldsymbol{W}_\ell^b := \boldsymbol{W}_{\ell-1} \dots \boldsymbol{W}_1 \tag{33}$$

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

$$\boldsymbol{a}_{\ell,n,r} = \boldsymbol{W}_\ell^b \boldsymbol{x}_{n,r} \in \mathbb{R}^{P_{\ell,\text{in}}} \tag{34}$$

and

$$\boldsymbol{b}_{\ell,n,r,r} = \boldsymbol{W}_\ell^{a\mathsf{T}} \in \mathbb{R}^{P_{\ell,\text{out}} \times C}. \tag{35}$$

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

Proposition 1 (**Exactness of K-FAC-expand for deep linear network in the expand setting**). *For layer ℓ of a deep linear network defined as in Equation (8) 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 $\boldsymbol{b}_{\ell,n,r,r} = \boldsymbol{W}_\ell^{a\mathsf{T}}$. Hence, $\boldsymbol{b}_{\ell,n,m,m}\boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_m)\boldsymbol{b}_{\ell,n,m,m}^\mathsf{T} = \boldsymbol{W}_\ell^{a\mathsf{T}}\boldsymbol{\Sigma}^{-1}\boldsymbol{W}_\ell^a$ for all $n \in \{1, \dots, N\}$ and $m \in \{1, \dots, R\}$. Therefore, the desired result follows from Lemma 3. $\qquad\square$

B.2.2 The reduce setting and K-FAC-reduce

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

The Reduce Setting

$$\mathcal{L}_{\text{reduce}}(f_{\boldsymbol{\theta}}, \mathcal{D}) := -\sum_{n=1}^{N} \log p(\boldsymbol{y}_n | f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)), \tag{36}$$

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

21

with a model with linear weight-sharing layers is image classification with a vision transformer or a convolutional neural network. Note, that the inputs $\boldsymbol{x}_n$ and labels $\boldsymbol{y}_n$ 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 $\boldsymbol{S}_{\ell,n} = \boldsymbol{A}_{\ell,n}\boldsymbol{W}_\ell^\intercal \in \mathbb{R}^{R \times P_{\ell,\text{out}}}$. 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 multi-dimensional arrays. Luckily, we can simplify this by writing

$$(\boldsymbol{J}_{\boldsymbol{\theta}_\ell}f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))_{ij} = \sum_{r=1}^{R} \sum_{p=1}^{P_{\ell,\text{out}}} \frac{\partial f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)_i}{\partial \boldsymbol{S}_{\ell,n,rp}} \frac{\partial \boldsymbol{S}_{\ell,n,rp}}{\partial \boldsymbol{\theta}_{\ell,j}}, \tag{37}$$

or in matrix form

$$\boldsymbol{J}_{\boldsymbol{\theta}_\ell}f_{\boldsymbol{\theta}}(\boldsymbol{x}_n) = \sum_{r=1}^{R} \boldsymbol{J}_{\boldsymbol{s}_{\ell,n,r}}f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)\boldsymbol{J}_{\boldsymbol{\theta}_\ell}\boldsymbol{s}_{\ell,n,r}, \tag{38}$$

where $\boldsymbol{s}_{\ell,n,r} \in \mathbb{R}^{P_{\ell,\text{out}}}$ is the r-th 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)^\intercal \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)^\intercal \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)^\intercal \\
&\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}^\intercal \right] \otimes \left[\sum_{r=1}^{R} \boldsymbol{b}_{\ell,n,r}^\intercal \right] \\
&= \sum_{n=1}^{N} \left(\hat{\boldsymbol{a}}_{\ell,n}\hat{\boldsymbol{a}}_{\ell,n}^\intercal \right) \otimes \left(\hat{\boldsymbol{b}}_{\ell,n}\boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))\hat{\boldsymbol{b}}_{\ell,n}^\intercal \right) \\
&\approx \underbrace{\left[\frac{1}{N}\sum_{n=1}^{N} \hat{\boldsymbol{a}}_{\ell,n}\hat{\boldsymbol{a}}_{\ell,n}^\intercal \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}^\intercal \right]}_{=:\hat{\boldsymbol{B}}_\ell},
\end{aligned} \tag{39}$$

where we have approximated the sum over the R Kronecker products with a Kronecker product of sums for each of the N per-input Jacobians, before applying the same type of approximation as usual to the sum over the N data points. This approximation has been proposed in Tang et al. (2021) to improve the efficiency of their proposed K-FAC variation (SKFAC) for convolutions by reducing the spatial dimension, purely based on the empirical observation that it works well in practice. The idea to approximate the Jacobians within the GGN with a Kronecker-product has also 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 (39) *K-FAC-reduce* and to highlight the difference to K-FAC-expand, we can rewrite it as

22

K-FAC-reduce

$$\mathbf{G\hat{G}N}_{\boldsymbol{\theta}_\ell}^{\text{reduce}} :=$$

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

(40)

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.

Lemma 5 (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 $r \in \{1, \ldots, R\}$ and $\hat{\boldsymbol{b}}_{\ell,n} \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \hat{\boldsymbol{b}}_{\ell,n}^{\mathsf{T}} = \boldsymbol{C}_\ell$ for all $n \in \{1, \ldots, N\}$, then the K-FAC-reduce approximation in Equation (10) is equal to the exact GGN of the ℓ-th layer in the reduce setting.*

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

$$\sum_{n=1}^{N} \left(\frac{1}{R} \sum_{r=1}^{R} \boldsymbol{a}_{\ell,n,r} \otimes \sum_{r=1}^{R} \boldsymbol{b}_{\ell,n,r} \right) \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \left(\frac{1}{R} \sum_{r=1}^{R} \boldsymbol{a}_{\ell,n,r} \otimes \sum_{r=1}^{R} \boldsymbol{b}_{\ell,n,r} \right)^{\mathsf{T}}$$

$$= \sum_{n=1}^{N} \left(\frac{1}{R} \sum_{r=1}^{R} \boldsymbol{a}_{\ell,n,r} \otimes R\boldsymbol{D}_{\ell,n} \right) \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \left(\frac{1}{R} \sum_{r=1}^{R} \boldsymbol{a}_{\ell,n,r} \otimes R\boldsymbol{D}_{\ell,n} \right)^{\mathsf{T}} \qquad (41)$$

$$= \sum_{n=1}^{N} \left(\sum_{r=1}^{R} \boldsymbol{a}_{\ell,n,r} \otimes \boldsymbol{b}_{\ell,n,r} \right) \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \left(\sum_{r=1}^{R} \boldsymbol{a}_{\ell,n,r} \otimes \boldsymbol{b}_{\ell,n,r} \right)^{\mathsf{T}},$$

where we have used the assumption that for each n, we have $\boldsymbol{b}_{\ell,n,r} = \boldsymbol{D}_{\ell,n}$ for all $r \in \{1, \ldots, R\}$. Now we consider the second approximation in Equation (39). Analogously, we have

$$\left(\frac{1}{N} \sum_{n=1}^{N} \hat{\boldsymbol{a}}_{\ell,n} \hat{\boldsymbol{a}}_{\ell,n}^{\mathsf{T}} \right) \otimes \left(\sum_{n=1}^{N} \hat{\boldsymbol{b}}_{\ell,n} \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \hat{\boldsymbol{b}}_{\ell,n}^{\mathsf{T}} \right)$$

$$= \left(\frac{1}{N} \sum_{n=1}^{N} \hat{\boldsymbol{a}}_{\ell,n} \hat{\boldsymbol{a}}_{\ell,n}^{\mathsf{T}} \right) \otimes N\boldsymbol{C}_\ell \qquad (42)$$

$$= \sum_{n=1}^{N} \left(\hat{\boldsymbol{a}}_{\ell,n} \hat{\boldsymbol{a}}_{\ell,n}^{\mathsf{T}} \right) \otimes \left(\hat{\boldsymbol{b}}_{\ell,n} \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \hat{\boldsymbol{b}}_{\ell,n}^{\mathsf{T}} \right),$$

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

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 1 and Proposition 2, we now have to also check if the Jacobian $\boldsymbol{J}_{\boldsymbol{s}_{\ell,n,r}} z_{\ell,n}$ with $z_{\ell,n} := z(\boldsymbol{S}_{\ell,n}) \in \mathbb{R}^{P_{\ell,\text{out}}}$ is the same for all $r \in \{1, \ldots, R\}$, to make sure the first condition in Lemma 5 is fulfilled. Maybe

23

the simplest case where this holds is a scaled sum, i.e.

$$
\begin{aligned}
z(\boldsymbol{S}_{\ell,n}) &= c \sum_{r=1}^{R} \boldsymbol{s}_{\ell,n,r} \\
&= c \boldsymbol{S}_{\ell,n}^{\mathsf{T}} \mathbf{1}_R \\
&= \left(\mathbf{1}_R^{\mathsf{T}} \otimes c\mathbf{I}_{P_{\ell,\text{out}}}\right) \operatorname{vec}(\boldsymbol{S}_{\ell,n}^{\mathsf{T}}) \\
&= \left(\mathbf{1}_R^{\mathsf{T}} \otimes c\mathbf{I}_{P_{\ell,\text{out}}}\right) \boldsymbol{K}^{(R,P_{\ell,\text{out}})} \operatorname{vec}(\boldsymbol{S}_{\ell,n})
\end{aligned}
\tag{43}
$$

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

$$
\boldsymbol{K}^{(R,P_{\ell,\text{out}})} := \sum_{r=1}^{R} \sum_{p=1}^{P_{\ell,\text{out}}} (\boldsymbol{e}_{R,r} \boldsymbol{e}_{P_{\ell,\text{out}},p}^{\mathsf{T}}) \otimes (\boldsymbol{e}_{P_{\ell,\text{out}},p} \boldsymbol{e}_{R,r}^{\mathsf{T}}),
\tag{44}
$$

where $\boldsymbol{e}_{i,j}$ is the j-th canonical vector of dimension i. This is a linear function in $\operatorname{vec}(\boldsymbol{S}_{\ell,n})$ and we have $\boldsymbol{J}_{\boldsymbol{s}_{\ell,n,r}} \boldsymbol{z}_{\ell,n} = c\mathbf{I}_{P_{\ell,\text{out}}}$ for all $r \in \{1, \ldots, R\}$. 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 $\operatorname{vec}(\boldsymbol{S}_{\ell,n})$, because as soon as we have a weighted sum with weights $c_r \in \mathbb{R}$ and they are not the same for all $r \in \{1, \ldots, R\}$, the Jacobians $\boldsymbol{J}_{\boldsymbol{s}_{\ell,n,r}} \boldsymbol{z}_{\ell,n}$ will also not be the same anymore. Both vision transformers and convolutional neural networks with average pooling use scaled sums as the aggregation function (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 4.

Proposition 6 (Exactness of K-FAC-reduce for single linear layer 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 (43) 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 $\boldsymbol{b}_{\ell,n,r} = \left(\boldsymbol{J}_{\boldsymbol{z}_{\ell,n}} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n) \boldsymbol{J}_{\boldsymbol{s}_{\ell,n,r}} \boldsymbol{z}_{\ell,n}\right)^{\mathsf{T}} = c\mathbf{I}_C$ for all $r \in \{1, \ldots, R\}$ and $n \in \{1, \ldots, N\}$ ($P_{\ell,\text{out}} = C$ for a single layer). Hence, $\hat{\boldsymbol{b}}_{\ell,n} \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \hat{\boldsymbol{b}}_{\ell,n}^{\mathsf{T}} = c^2 R^2 \mathbf{I}_C \boldsymbol{\Sigma}^{-1} \mathbf{I}_C = c^2 R^2 \boldsymbol{\Sigma}^{-1}$ for all $n \in \{1, \ldots, N\}$. Therefore, the desired result follows from Lemma 5. $\square$

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

Proposition 2 (Exactness of K-FAC-reduce for deep linear network in the reduce setting). *For layer ℓ of a deep linear network (Equation (8)), a Gaussian likelihood with p.d. covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{C \times C}$, and a scaled sum aggregation function, 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 $\boldsymbol{b}_{\ell,n,r} = \left(\boldsymbol{J}_{\boldsymbol{z}_{\ell,n}} f_{\boldsymbol{\theta}}(\boldsymbol{x}_n) \boldsymbol{J}_{\boldsymbol{s}_{\ell,n,r}} \boldsymbol{z}_{\ell,n}\right)^{\mathsf{T}} = c\boldsymbol{W}_\ell^{a\mathsf{T}}$ for all $r \in \{1, \ldots, R\}$ and $n \in \{1, \ldots, N\}$. Hence,

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

for all $n \in \{1, \ldots, N\}$. Therefore, the desired result follows from Lemma 5. $\square$

To summarise, 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 Figure 1 and Figure 2 we verify this numerically, and also show that using the inappropriate approximation results in an inexact computation.

Remark 1. *In practice, however, both approximations can be applied in each of the two settings.*

24

B.3 Examples of K-FAC for linear weight-sharing layers in the wild

B.3.1 K-FAC for self-attention

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 self-attention mechanism in Section 2.2. To gain some intuition for models using this type of attention mechanism, we look at a network that only consists of one simplified variation of the self-attention mechanism used in transformers (we ignore the softmax function, but also consider a linear projection of the output with weight matrix $\boldsymbol{W}^V$), i.e.

$$f_{\boldsymbol{\theta}}(\boldsymbol{X}_n) = \underbrace{\boldsymbol{X}_n \boldsymbol{W}^{Q^\mathsf{T}}}_{=:\boldsymbol{S}_{Q,n}} \underbrace{\boldsymbol{W}^K \boldsymbol{X}_n^\mathsf{T}}_{=:\boldsymbol{S}_{K,n}^\mathsf{T}} \underbrace{\boldsymbol{X}_n \boldsymbol{W}^{V^\mathsf{T}}}_{=:\boldsymbol{S}_{V,n}}. \tag{45}$$

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 self-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 (27) 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 r-th row of the output of the layer with the weight matrix $\boldsymbol{W}^Q$ as a function of $\boldsymbol{s}_{Q,n,r}$ as

$$f_{\boldsymbol{\theta}}(\boldsymbol{X}_n)_r = \boldsymbol{s}_{Q,n,r}^\mathsf{T} \boldsymbol{S}_{K,n}^\mathsf{T} \boldsymbol{S}_{V,n}. \tag{46}$$

Therefore, we have

$$\boldsymbol{J}_{\boldsymbol{s}_{Q,n,r}} f_{\boldsymbol{\theta}}(\boldsymbol{X}_n)_r = \boldsymbol{S}_{V,n}^\mathsf{T} \boldsymbol{S}_{K,n} = \boldsymbol{b}_{Q,n,r}^\mathsf{T} \in \mathbb{R}^{C \times P_{K,\text{out}}}, \tag{47}$$

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 to hold. While $\boldsymbol{b}_{Q,n,r}$ is not the same for all $n \in \{1, \ldots, N\}$, it is the same for all $r \in \{1, \ldots, R\}$ and hence, under the same assumptions as in Proposition 4, 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 r-th row of $f_{\boldsymbol{\theta}}(\boldsymbol{X}_n)$ as a function of the r-th row of $\boldsymbol{S}_{K/V,n}$, i.e. elements from all rows of $\boldsymbol{S}_{K/V,n}$ contribute to the r-th 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

$$\boldsymbol{J}_{\boldsymbol{s}_{K,n,m}} f_{\boldsymbol{\theta}}(\boldsymbol{X}_n)_r = \boldsymbol{s}_{V,n,m} \boldsymbol{s}_{Q,n,r}^\mathsf{T} \in \mathbb{R}^{C \times P_{Q,\text{out}}}. \tag{48}$$

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

$$\boldsymbol{J}_{\boldsymbol{s}_{V,n,m}} f_{\boldsymbol{\theta}}(\boldsymbol{X}_n)_r = \boldsymbol{s}_{Q,n,r}^\mathsf{T} \boldsymbol{s}_{K,n,m} \boldsymbol{I}_C \in \mathbb{R}^{C \times C}. \tag{49}$$

This means that the assumption of Lemma 3 that the R elements along the weight-sharing dimension are independent does not hold, since the Jacobians depend 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 self-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 (39) as

$$\begin{aligned}
\boldsymbol{J}_{\boldsymbol{\theta}_\ell} z(f_{\boldsymbol{\theta}}(\boldsymbol{X}_n)) &= \sum_{r=1}^{R} \boldsymbol{J}_{\boldsymbol{s}_{\ell,n,r}} z(f_{\boldsymbol{\theta}}(\boldsymbol{X}_n)) \boldsymbol{J}_{\boldsymbol{\theta}_\ell} \boldsymbol{s}_{\ell,n,r} \\
&= \sum_{r=1}^{R} \boldsymbol{a}_{\ell,n,r} \otimes \boldsymbol{b}_{\ell,n,r} \\
&= \sum_{r=1}^{R} \boldsymbol{a}_{\ell,n,r} \otimes \left(c \sum_{m=1}^{R} \boldsymbol{J}_{\boldsymbol{s}_{\ell,n,r}} f_{\boldsymbol{\theta}}(\boldsymbol{X}_n)_m \right),
\end{aligned} \tag{50}$$

25

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 5 we need all $\boldsymbol{b}_{\ell,n,r}$ to be the same for all $r \in \{1, \ldots, R\}$ for the first approximation to be exact under the assumptions of Proposition 6, 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.

When we extend this scenario to a network consisting of L blocks as defined in Equation (45), 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 r-th row of this model's output as a function of only the r-th 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 L-th block with weights $\boldsymbol{W}_L^Q$.

This shows that even without explicit nonlinear activation functions, the self-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 is especially relevant given that we know that the computational complexity of K-FAC-reduce is generally smaller than of K-FAC-expand and given the similar downstream optimisation performance we report in Section 4.

B.3.2 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 Appendix A. 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 a GNN. The only difference to the normal derivation of K-FAC is that the inputs $\tilde{\boldsymbol{a}}_{\ell,n}$ to the ℓ-th layer for the node with index n now depend on its neighbourhood $\mathcal{N}(n)$, since

$$\tilde{\boldsymbol{a}}_{\ell,n} := \sum_{j \in \mathcal{N}(n)} \hat{\boldsymbol{C}}_{nj} \boldsymbol{a}_{\ell,j} \in \mathbb{R}^{P_{\ell,\mathrm{in}}}. \tag{51}$$

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

$$\begin{aligned} GGN(\boldsymbol{\theta}_\ell) &= \sum_{n=1}^N \boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{X})_n^\mathsf{T} \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{X})_n) \boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\boldsymbol{X})_n \\ &\approx \underbrace{\Big[\frac{1}{N} \sum_{n=1}^N \tilde{\boldsymbol{a}}_{\ell,n} \tilde{\boldsymbol{a}}_{\ell,n}^\mathsf{T}\Big]}_{=:\tilde{\boldsymbol{A}}_\ell} \otimes \underbrace{\Big[\sum_{n=1}^N \tilde{\boldsymbol{b}}_{\ell,n} \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\boldsymbol{X})_n) \tilde{\boldsymbol{b}}_{\ell,n}^\mathsf{T}\Big]}_{=:\tilde{\boldsymbol{B}}_\ell}, \end{aligned} \tag{52}$$

where $\boldsymbol{X} \in \mathbb{R}^{N \times D}$, $\tilde{\boldsymbol{b}}_{\ell,n} := \boldsymbol{J}_{\tilde{\boldsymbol{s}}_{\ell,n}} f_{\boldsymbol{\theta}}(\boldsymbol{X})_n^\mathsf{T}$, 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 neighbourhood of $\boldsymbol{x}_n$ to be able to write the n-th 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 neighbourhoods, we might have multiple identical terms. In the extreme case of all neighbourhoods 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 the first of the three cases in Section 3.1, we do not need to think in terms of the expand and reduce settings here – as opposed to the case of the GraphNetwork we consider next, because we aggregate over each node's neighbourhood *before* the forward pass through a linear layer.

Graph classification with GraphNetwork. Now, we want to look at a more general architecture, an instance of the GraphNetwork introduced in Battaglia et al. (2018) and described in Appendix A. It is important to note that while the inputs and the GraphNetwork 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 presented framework of thinking about K-FAC for linear weight-sharing layers to new model architectures. In contrast to the original description in Battaglia

26

et al. (2018), we already defined the inputs to a GraphNetwork 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 Appendix A, 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 Appendix B.4.

First, we note that we consider the task of graph classification. Hence, our loss has the same form as Equation (36), 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-molpcba workload of the `AlgoPerf` benchmark from the experiment in Section 4.2. 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 Appendix A to concretely write down the approximation.

To recap, 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 GraphNetwork 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

$$\boldsymbol{A}_{\ell,n} = \text{concat}(\boldsymbol{X}_n^E, \boldsymbol{X}_{n,\boldsymbol{r}_n}^V, \boldsymbol{X}_{n,\boldsymbol{s}_n}^V, \text{repeat}_{N_n^E}(\boldsymbol{x}_n^u)) \in \mathbb{R}^{N_n^E \times (D_E + 2D_V + D_u)} \tag{53}$$

and as

$$\boldsymbol{A}_{\ell,n} = \text{concat}(\boldsymbol{X}_n^V, \tilde{\boldsymbol{X}}_n^E, \text{repeat}_{N_n^V}(\boldsymbol{x}_n^u)) \in \mathbb{R}^{N_n^V \times (D_V + D_E + D_u)} \tag{54}$$

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

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

$$\mathbf{GGN}(\boldsymbol{\theta}_\ell) = \sum_{n=1}^N \boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\mathbb{X}_n^G)^\intercal \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\mathbb{X}_n^G)) \boldsymbol{J}_{\boldsymbol{\theta}_\ell}(\mathbb{X}_n^G)$$

$$\approx \sum_{n=1}^N \underbrace{\left[\sum_{r=1}^{R_n} \frac{1}{\sqrt{R_n}} \boldsymbol{a}_{\ell,n,r}\right]}_{=:\hat{\boldsymbol{a}}_{\ell,n}} \otimes \underbrace{\left[\sum_{r=1}^{R_n} \frac{1}{\sqrt{R_n}} \boldsymbol{b}_{\ell,n,r}\right]}_{=:\hat{\boldsymbol{b}}_{\ell,n}} \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\mathbb{X}_n^G)) \left[\sum_{r=1}^{R_n} \frac{1}{\sqrt{R_n}} \boldsymbol{a}_{\ell,n,r}\right] \otimes \left[\sum_{r=1}^{R_n} \frac{1}{\sqrt{R_n}} \boldsymbol{b}_{\ell,n,r}\right]^\intercal$$

$$\approx \underbrace{\left[\frac{1}{N} \sum_{n=1}^N \hat{\boldsymbol{a}}_{\ell,n} \hat{\boldsymbol{a}}_{\ell,n}^\intercal\right]}_{=:\hat{\boldsymbol{A}}_\ell} \otimes \underbrace{\left[\sum_{n=1}^N \hat{\boldsymbol{b}}_{\ell,n} \boldsymbol{\Lambda}(f_{\boldsymbol{\theta}}(\mathbb{X}_n^G)) \hat{\boldsymbol{b}}_{\ell,n}^\intercal\right]}_{=:\hat{\boldsymbol{B}}_\ell},$$

$$\tag{55}$$

analogously to Equation (39). However, there is one important difference: since R_n now depends on the n-th data point, it makes a difference where we insert the scaling by $1/R_n$, since $\hat{\boldsymbol{A}}_\ell$ and/or $\hat{\boldsymbol{B}}_\ell$ are now weighted sums. To avoid having one potentially non-uniformly weighted and one unweighted sum, we choose to not just include the scaling by $1/R_n$ in $\hat{\boldsymbol{a}}_{\ell,n}$ as in Equation (39), or in $\hat{\boldsymbol{b}}_{\ell,n}$. Instead, to try to keep the overall scale and the weighting of the N terms balanced, we simply include a scaling by $1/\sqrt{R_n}$ in *both*, $\hat{\boldsymbol{a}}_{\ell,n}$ and $\hat{\boldsymbol{b}}_{\ell,n}$.

B.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`

27

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

Listing 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. 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. Here we assume that only one additional weight-sharing dimension exists and that the first dimension is always the mini-batch dimension. Note that the dimensions of `in_data` and `out_grads` are the same, *independently of the setting we are in* (c.f. Section 3.1). This illustrates why we can choose to use either approximation, K-FAC-expand or K-FAC-reduce, in each setting. The actual implementation in ASDL is very similar for `torch.nn.Linear` modules, the logic is just separated into multiple functions and allows for multiple weight-sharing dimensions. However, the adjustments for the GraphNetwork are more involved.

(Botev & Martens, 2022) for `Jax` and `BackPACK` (Dangel et al., 2020), `ASDL` (Osawa et al., 2023), and `KFAC-PyTorch` (Pauloski et al., 2021) for `PyTorch`. We focus on the implementation of K-FAC-expand and K-FAC-reduce within `ASDL`, which we also use for the experiments in Section 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 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 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 GraphNetwork introduced in Appendix A, we have to adopt our implementation due to the batching technique that 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 (53) and Equation (54) 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

28

each mini-batch by $1/R_{\mathcal{M}}$ instead of $1/M$.[8] To apply K-FAC-reduce, we can use a *scatter mean*, which aggregates tensor elements according to indices, to implement the mean (with the square root scaling from Equation (55)) 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 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 implementation, because additional attributes have to be set for the regular linear modules.

Besides the changes necessary for K-FAC-expand and K-FAC-reduce, we can use the same additional algorithmic tools often used for optimisation 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. However, in our experiments, we do not use such a moving average and only compute K-FAC for a single mini-batch (besides for Figure 5, as mentioned in Appendix C.4) and still reduce the update frequency of the K-FAC statistics and the preconditioner.

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 $\boldsymbol{H}_{f_{\boldsymbol{\theta}}} \ell(\boldsymbol{y}_n, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))$ for each of the N data points through the computation graph. Instead, we use the fact that we have

$$\mathbb{E}_{\boldsymbol{y} \sim p(\boldsymbol{y}|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))}[\nabla_{f_{\boldsymbol{\theta}}} \log p(\boldsymbol{y}|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \nabla_{f_{\boldsymbol{\theta}}} \log p(\boldsymbol{y}|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))^{\mathsf{T}}] = \boldsymbol{H}_{f_{\boldsymbol{\theta}}} \ell(\boldsymbol{y}_n, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \qquad (56)$$

and take S Monte Carlo (MC) samples from the model's predictive distribution $\boldsymbol{y}_s \sim p(\boldsymbol{y}|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))$. 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.

C   Additional experimental details and results

C.1   MLCommons' `AlgoPerf` benchmark for training algorithms

The goal of the `AlgoPerf` benchmark for training algorithms (Dahl et al., 2023) is to measure "training speedups due to algorithmic improvements" (MLCommons, 2022). Specifically, the benchmark defines multiple workloads, where one workload is defined by the combination of a dataset, a neural network model, a loss function, and a target performance defined by some evaluation metric. For example, training a ViT on ImageNet using the cross-entropy loss until a target validation accuracy of 77.309% has been reached constitutes a workload. In the `AlgoPerf` benchmark, training algorithms are compared on fixed hardware in terms of the wall-clock time they require to reach the fixed validation target performance.

For our experiments, we use two of the workloads of the `AlgoPerf` benchmark as realistic deep learning problems, to showcase the optimisation behaviour of our two K-FAC variations (Sections 4.2 and 4.3). Similar to the benchmark, we measure both the number of steps and the wall-clock time necessary to reach the target validation metric. To put the training behaviour of K-FAC-expand and K-FAC-reduce into perspective, we compare them to the *target-setting* run of the `AlgoPerf` benchmark, denoted *reference run* in the experiments (Sections 4.2 and 4.3). To determine the fixed validation target for each workload, four standard algorithms (AdamW, NAdamW, SGD with Nesterov momentum (Nesterov), and SGD with heavy ball momentum) were each tuned with a budget of 200 trials for every workload. The combination of algorithm and hyperparameters that reached the highest validation performance within a fixed time was then run 20 times and the median of the best achieved validation metric was set as the validation target. As our reference run, we use the workload-specific, best-performing algorithm and hyperparameter combination. Note that this should

---

[8]As explained below Equation (55), we could also choose a different way of scaling here. We choose this one as it enables a simple implementation in contrast to K-FAC-reduce.

29

Table 3: Results from Table 1 with standard errors over 10 epochs.

K-FAC	Batch size				
	128	256	512	1024	2048
expand	0.24 ± 0.01	0.38 ± 0.02	0.75 ± 0.03	1.36 ± 0.05	OOM
reduce	0.17 ± 0.01	0.24 ± 0.02	0.43 ± 0.03	0.63 ± 0.04	1.17 ± 0.08

not be confused with the baseline algorithms in the `AlgoPerf` benchmark, since the target-setting runs are tuned for the best possible validation performance on each workload.

K-FAC is run with most of the hyperparameters of the reference run and a tuning budget of 15 runs for tuning the learning rate and damping. This means that most hyperparameters are not directly tuned for K-FAC. To enable a fairer comparison, we also use the same additional budget for tuning the learning rate of the reference run, the same learning rate schedule, and tuning goal as K-FAC. However, none of these runs reaches the validation metric target, which is why the reference runs correspond to the original target-setting runs. Despite this, the experiments presented in Sections 4.2 and 4.3 are *not* meant to demonstrate that K-FAC is a superior training algorithm. To provide actual evidence for this claim we would have to run a *valid* and optimised submission on the entire `AlgoPerf` benchmark or a benchmark of comparable rigour. Nevertheless, using the well-tuned reference run allows us to put K-FAC's training performance into perspective.

The two `AlgoPerf` workloads used in this paper are:

Graph neural network on ogbg-molpcba. The workload consists of a (binary) cross-entropy loss, the GraphNetwork instance described in Appendix A, 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. The validation target mean average precision (mAP) determined by the target-setting runs is 0.28098 and the best performing reference run algorithm is Nesterov.

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. The validation target accuracy determined by the target-setting runs is 0.77309 and the best performing reference run algorithm is NAdamW.

C.2 Update step speed with K-FAC-expand and K-FAC-reduce

We provide the full results of the update step timing experiment with standard errors over 10 epochs in Table 3. The model architecture, optimiser, and data setup are exactly the same as described in Appendix C.5. The full results of the GPT-2 (`nanoGPT`) timing experiment are presented in Table 4; the mean and standard error are computed based on three runs.

Table 4: Timing of K-FAC GGN approximation for GPT-2 on full DART dataset.

K-FAC	Absolute time [min] $\downarrow$	Relative time [%] $\downarrow$
expand	9.55 ± 0.13	100
reduce	6.68 ± 0.04	70

C.3 Graph neural network on ogbg-molpcba

For this workload, we use a training batch size of 512 and a single NVIDIA V100 32GB GPU for each run. The reference run algorithm (Nesterov) uses a learning rate of 2.4917728606918423, β_1 equal to 0.9449369031171744, weight decay set to $1.285\,964\,054\,102\,592\,8e{-}7$, and linear warmup for 3,000 steps and a polynomial schedule with a decay steps factor of 0.861509027839639 and an end factor of $1e{-}3$. The two K-FAC variations use the exact same hyperparameters, but the warmup and expected number of steps (60,000) are multiplied by 0.75 and the learning rate and the damping are tuned via random search. The search space for the learning rate is log uniform values in $[0.1, 10]$ and for the damping in $[1e{-}3, 1]$. We choose the hyperparameters of the run that first reaches the validation target. This setting is a learning rate of 9.96871902194967 and damping of 0.7881965339190345 for K-FAC-expand and a learning rate of 0.5885756514016359 and damping

30

of 0.0579230193904011 for K-FAC-reduce. We then repeated each run five times. The Kronecker factors and the preconditioner are computed every 10 iterations and we use a single sample MC approximation of the Fisher, as explained in Appendix B.4.

C.4 Vision transformer on ImageNet

We use a training batch size of 1,024, $\epsilon = 1e-8$ for NAdamW, and $4\times$ NVIDIA V100 32 GPUs for all runs on this workload. The reference run algorithm (NAdamW) is using a learning rate of 0.0008445074561975979, β_1 equal to 0.8895758153482813, β_2 to 0.9978504782314613, weight decay set to 0.08135402759553023, linear warmup for 6999 steps, and a cosine decay schedule. The two K-FAC variations use the exact same hyperparameters, but the warmup and expected number of steps (140,000) is multiplied by 0.75 and the learning rate and the damping are tuned via random search. The search space for the learning rate and the damping is log uniform values in [1e−4, 1e−2]. We choose the hyperparameters of the run that first reaches the validation target. For both K-FAC variations, the best setting is a learning rate of 0.0012662938340704357 and damping of 0.00016524019235426572. The Kronecker factors and the preconditioner are computed every 50 iterations and we use a single sample MC approximation of the Fisher, as explained in Appendix B.4.

We also conduct a run where we update the Kronecker factors every step, use an exponential moving average with a factor (`ema_decay` in ASDL) equal to β_2, update the preconditioner every 10 steps, set the damping to 1e−5, and use all the other hyperparameters of a reference run setting, including the learning rate. The reference run setting corresponds to an earlier target-setting run result from the `AlgoPerf` repository and uses a learning rate of about 2e−3, $\beta_1 = 0.7132$, $\beta_2 = 0.9982$, and the weight decay is set to 0.026595. Moreover, it clips the gradients to keep their norm below 1. As before, we also multiply the number of warmup and expected number of steps by 0.75. Due to the high update frequency of the Kronecker factors, we can see the significant wall-clock time difference between K-FAC-expand and K-FAC-reduce in Figure 5. Both variations are similar in terms of steps to the target, K-FAC-expand takes about 92.6k and K-FAC-reduce takes about 93.7k steps, but whereas K-FAC-expand takes about 50 hours, K-FAC-reduce reaches the target after about 37 hours. Note, that both variations are still significantly slower than the NAdamW reference run which only takes about 25 hours, but 117.4k steps.

C.5 K-FAC for automatic hyperparameter selection via marginal likelihood optimisation

K-FAC is used for the Hessian approximation within the Laplace approximation by first accumulating it over the entire data set and using the eigendecomposition of Kronecker factors as in Immer et al. (2021). This is done every five epochs during training to update the weight decay parameters per layer of the neural network. Therefore, the computation time of K-FAC makes a significant contribution to the overall training time: reduce takes on average 75% of the overall time of expand, a result of reducing the wall-clock time of a single marginal likelihood update step by about 50%, as shown in Table 2. Here, we provide additional experimental details and the full table with standard errors.

For the marginal likelihood optimisation experiment, we use the same settings as Daxberger et al. (2021), i.e., a Wide ResNet 16-4 with Fixup initialisation and parameters instead of batch normalisation (Zagoruyko & Komodakis, 2016; Zhang et al., 2019b). We use a batch size of 128, an initial learning rate of 0.1 cosine decayed to 1e−6 and weight decay of 5e−3 training for 100 epochs. We use standard data augmentation with horizontal flips and random crops. Results are averaged over three seeds and the timings are obtained running all runs on a single NVIDIA A100 GPU and locally measuring the hyperparameter update time. The full results with standard errors are presented in Table 5.

31
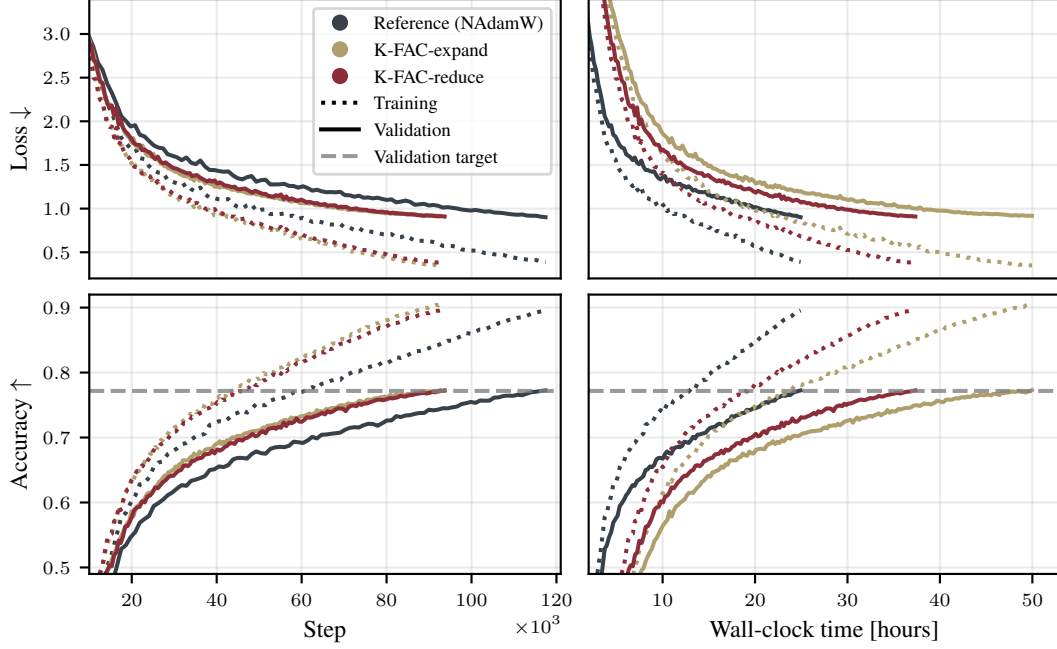
Figure 5: **Training results for a vision transformer on ImageNet.** This is similar to Figure 4, but the K-FAC statistics are updated *every* step and different hyperparameters are used. Due to K-FAC's overhead, the wall-clock time is not reduced in this setting. Moreover, the discrepancy in speed between K-FAC-expand and K-FAC-reduce becomes apparent.

Table 5: Results from Table 2 with standard errors over three random seeds. The update time is the average time per full K-FAC Laplace approximation to the marginal likelihood, which requires a full data set pass with computation and eigendecomposition of the Kronecker factors.

K-FAC	Data Augmentation	NLL $\downarrow$	Accuracy [%] $\uparrow$	Update Time [s] $\downarrow$
expand	✗	0.422 ± 0.013	88.89 ± 0.24	196 ± 22
	✓	0.244 ± 0.004	92.52 ± 0.12	
reduce	✗	0.703 ± 0.012	86.71 ± 0.13	99 ± 23
	✓	0.352 ± 0.008	93.50 ± 0.06	

32