The Gauss-Newton matrix for Deep Learning models and its applications

Aleksandar Botev

August 2020

Declaration

I, Aleksandar Botev confirm that the work presented in this thesis is my own and includes experiments and results done in collaboration as declared in the preface. Where information has been derived from other sources, I confirm that this has been indicated in the thesis. The thesis does not exceed the word limit of 100,000 words including footnotes, tables, figures, appendices and bibliography as per the Research Degree Regulations.

Aleksandar Botev

August 2020

Acknowledgements

I would firstly like to thank my family for all the support they have provided me with during my studies. I would also to like to thank my supervisor David Barber, for his many pieces of advice, help and support throughout my PhD program. I'm also very grateful to all my colleges from my group at University College London who have corrected me endless times and made my studies so much enjoyable - Thomas Anthony, Thomas Bird, Raza Habib, Zhen He, Louis Kirsch, Julius Kunze, Hippolyt Ritter, Harshil Shah, James Townsend, Tianlin Xu, Mingtian Zhang, Bowen Zheng. [1] Furthermore, I would like to thank all my friends, colleges and reviewers who have helped me during the writing and submitting of my publications and thesis.

I'm also very grateful to both OpenAI and Google DeepMind for the opportunity they have provided me with during my studies to learn and do research outside of academia. Lastly, I would like to thank the Centre for Doctoral Training in Financial Computing and Analytics for supporting my four years of PhD studies.

[1] Listed in alphabetical order.

Abstract

Deep Learning learning has recently become one of the most predominantly used techniques in the field of Machine Learning. Optimising these models, however, is very difficult and in order to scale the training to large datasets and model sizes practitioners use first-order optimisation methods. One of the main challenges of using the more sophisticated second-order optimisation methods is that the curvature matrices of the loss surfaces of neural networks are usually intractable, which is an open avenue for research.

In this work, we investigate the Gauss-Newton matrix for neural networks and its application in different areas of Machine Learning. Firstly, we analyse the structure of the Hessian and Gauss-Newton matrices for Feed Forward Neural Networks. Several insightful results are presented, and the relationship of these two matrices to each other and to the Fisher matrix is discussed. Based on this analysis, we develop a block-diagonal Kronecker Factored approximation to the Gauss-Newton matrix. The method is experimentally validated in the context of second-order optimisation, where it achieves competitive performance to other approaches on three datasets. In the last part of this work, we investigate the application of the proposed method for constructing an approximation to the posterior distribution of the parameters of a neural network. The approximation is constructed by adapting the well known Laplace approximation using the Kronecker factored Gauss-Newton matrix approximation. The method is compared against Dropout, a commonly used technique for uncertainty estimation, and achieves better uncertainty estimates on out of distribution data and is less susceptible to adversarial attacks. By combining the Laplace approximation with the Bayesian framework for online learning, we develop a scalable method for overcoming catastrophic forgetting. It achieves significantly better results than other approaches in the literature on several sequential learning tasks. The final chapter discusses potential future research directions that could be of interest to the curious reader.

Impact Statement

The presented work demonstrates a wide spectrum of applications for the Gauss-Newton matrix of Deep Learning models. The analysis from the Chapter 3 can be of particular use to other researchers investigating the curvature of neural networks and it could potentially give further insights into the loss surface structure. Chapter 4 proposes a practical algorithm for optimization, which can have a practical impact on practitioners both inside and outside academia. The much lower number of hyperparameters involved could also reduce the energy and hardware needed for training models, as it does not require the extensive grid search that is usually required for other methods. Chapter 5 demonstrates how to use the curvature approximation to get uncertainty estimates of the model predictions. Uncertainty is of great importance to any real-world applications where decisions based on the model output can have significant consequences, such as applications in the medical domain, human-robot interactions and self-driving cars. As such, the demonstrated benefits of using the proposed approximation scheme can be used both as a starting point for further research in this direction as well as an important improvement in practical applications. In addition, this chapter further shows improvement when our method is applied to the problem of continual learning. Our publications have already been used in other's research work. The code for the empirical experiments presented in any chapter is available at https://github.com/BB-UCL/Lasagne, with which we hope to speed up the adoption of the ideas presented.

Contents

List of Figures

List of Tables

Chapter 1

Why is the curvature of neural networks important

Machine Learning is a field, whose main goal is to extract the most relevant information for a given task from the observational data. It is tightly connected with statistics and computer science, as most of the models are based on statistical modelling assumptions and are usually highly scalable on computing hardware. However, compared to classical statistics, it is mostly focused on achieving good empirical results. In this direction, neural networks have demonstrated to be one of the most flexible models that are also easy to train. They have become the de facto standard for applications using unstructured data, such as images, raw text, sound and others. Training neural networks was very difficult in the early 90s and 2000s, but several methods have been developed since then, both for optimising them better, as well as for constructing random initialisations which allow being trained easily [104, 61, 50, 48]. Unfortunately, some of the best optimizers used in more classical Machine Learning models, could not be applied to neural networks directly. The main reason for this is that they required to compute the Hessian matrix, or some approximation of it, which will be referred to as a curvature matrix. Its size is the number of parameters squared, however, modern neural network models have on the order of millions or billions of parameters. This makes computing and storing a curvature matrix infeasible for any practical purposes. Additionally, as the name suggests, it is usually informative of the curvature of the objective loss that the model optimises. This has been investigated by researchers with the goal of better understanding what makes neural networks work as well as they do [38, 20]. In practice, however, the only thing that is possible to achieve is to either compute only

the diagonal entries of the curvature matrix or compute its projection onto some much smaller subspace. The work in this thesis tries to partially address this issue. Instead of computing the full curvature matrix, we propose to approximate only its diagonal blocks, corresponding to parameters in the same layer of the network, using Kronecker products. Although it might seem quite limited, we demonstrate that an approach like this provides a feasible alternative, that shows practical gains in several applications. Compared to the diagonal approximation, this preserves any intra-layer parameter dependencies, while ignoring any inter-layer ones.

Additionally, the curvature matrix plays an important role when used for modelling statistical dependencies between observed variables. In this scenario, the Hessian matrix is equal to the negative of the Fisher Information Matrix, also referred to as just the Fisher. This matrix is a measure of how much information the data carries over to the parameters of the model. It plays an important role in the famous Cramer-Rao bound, which provides a lower bound on the variance of any unbiased estimate of the true model parameters [109, 22]. Another important area, where the curvature matrix plays a central role, is Information Geometry. This arises when one considers the space of models that are defined by neural network as a Riemannian manifold. Specifically, when they define a distribution, this defines a Statistical manifold, where the Fisher matrix plays the role of a local metric tensor. This allows one to move along geodesics on the manifold in the direction of the gradient of any function. Since this is usually infeasible to compute, rather than computing the exact geodesic, in practice the Riemannian retraction is used [1]. If used for optimisation, the famous work of Amari proved that this retraction, termed Natural Gradient, is statistically efficient — it achieves the Cramer-Rao bound [2]. Other practical applications, where the Riemannian perspective has shown to provide significant improvements, are gradient-based Markov Chain Monte Carlo methods [75]. In practice, however, because it is infeasible to compute the curvature matrix, practitioners were restricted to only use diagonal approximations, which showed limited improvements as they ignore any intra-parameters correlations.

One of the main applications that we are interested in is constructing a Laplace approximation to the posterior density of a neural network. In this case, the curvature matrix will become the precision matrix of the approximate Gaussian distribution. This has been successfully applied before on small scale models [79]. Because

[1] The Riemannian retraction can be thought of as a first-order approximation to a geodesic differential equation.

of the size of the curvature matrix, this method is infeasible to apply even to an average modern neural network model. In the literature, a diagonal approximation has been tried, but in practice, this does not seem to give very good results. Hence, the approximation that we have developed, would enable the application of the Laplace methods with richer curvature information. In the context of having multiple sequential observations, Assumed Density Filtering is a classical method that constructs a Laplace approximation on every new observation [87]. This can be applied for the problem of online learning for neural networks, where the data from different tasks would play the role of different observations. Our experiments demonstrate that including curvature information with intra-layer dependencies improves significantly compared to the diagonal approximation or previously published methods. Furthermore, in general, for local Gaussian approximations, the curvature matrix is closely related to the evidence of the model conditioned on the data. This is a quantity of great importance in Bayesian statistics, that is used for model selection and comparisons. Unfortunately, we did not have the opportunity to apply our method to this problem.

1.1 Structure of the Thesis

The thesis is split into another five chapters. The next Chapter 2 presents a very broad introduction to topics regarding Deep Learning. This includes a short presentation of different neural network architectures as well as how they are used in practice. Also, a short introduction to optimisation methods, which are applied to Deep Learning models, is done, as it is one of the central problems that this work tries to solve in the later chapters. Finally, a short discussion of uncertainty estimation is presented, as well as its possible applications in different areas such as online learning. Chapter 3 will introduce the mathematical notations and background that will be used throughout the whole thesis. Moreover, it will present several important theoretical results regarding the structure of different curvature matrices for Feed Forward Neural Networks and their relationships with each other. This will layout the foundations that have motivated the work in Chapter 4, where our work on practical and more computationally efficient approximations to the Generalised Gauss-Newton matrix for a neural network is presented. At the end of the chapter are presented empirical evaluations and comparisons with other methods in the literature, together with further discussion on recent advancements in the same di-

rection. Chapter 5 will then build upon the practical curvature approximations developed and will show how they can be used to build an approximate posterior distribution over the parameters of a model. Several experiments have validated this idea and demonstrated that this method achieves better uncertainty estimates on multiple datasets, then competing approaches. Additionally, it would showcase how the posterior distribution can be applied to online learning and empirically demonstrate that it outperforms other methods from the literature. In the final Chapter 6, several standing problems with the presented curvature approximations will be discussed as well as potential future research directions.

All of the work in this thesis is based on the three conference papers, published during my PhD studies at University College London, listed below with the full list of collaborators:

> Botev, A., Ritter, H., and Barber, D. "Practical Gauss-Newton Optimisation for Deep Learning". In: *Proceedings of the 34th International Conference on Machine Learning*. Vol. 70. Proceedings of Machine Learning Research. International Convention Centre, Sydney, Australia, 2017, pp. 557–565
>
> Ritter, H., Botev, A., and Barber, D. "A Scalable Laplace Approximation for Neural Networks". In: *International Conference on Learning Representations*. 2018
>
> Ritter, H., Botev, A., and Barber, D. "Online Structured Laplace Approximations for Overcoming Catastrophic Forgetting". In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS'18. 2018, pp. 3742–3752

In addition, the following papers have been published during my PhD:

> Botev, A., Lever, G., and Barber, D. "Nesterov's accelerated gradient and momentum as approximations to regularised update descent". In: *2017 International Joint Conference on Neural Networks (IJCNN)*. 2017, pp. 1899–1903
>
> Botev, A., Zheng, B., and Barber, D. "Complementary Sum Sampling for Likelihood Approximation in Large Scale Classification". In: *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*. Vol. 54. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA, 2017, pp. 1030–1038

Chapter 2

Background

In the years before Deep Learning became the popular technique it is today, in the field of Machine Learning researchers used to train significantly smaller models with smaller datasets. Some of the best tools used for optimising the parameters of their models were based on second-order optimisation — techniques which required the estimation of a curvature matrix of the objective function. However, what is a curvature matrix? It is difficult to give a precise definition, but we will describe it as a matrix that tries to capture properties of the loss function by approximating in some way the matrix of second-order derivatives and can be used to speed up optimisation in practical terms. Unfortunately, none of the previous second-order techniques scale well to the modern demands of Deep Learning models despite the ever-improving computational hardware that is being used. As a result, models in more recent years are predominantly optimised using first-order methods as they scale very easily to both large models and datasets.

In light of this, this thesis is mainly concerned with investigating the Generalised Gauss-Newton matrix for Deep Learning models, a specific type of curvature matrix, as well as different practical approximations of it. Despite the original motivation, the benefits of having a curvature matrix go beyond just optimisation — it also allows us to make a richer scalable approximation to the posterior distribution of the parameters of a neural network. This approximation, although not perfect, still performs better compared to other alternatives presented in the literature. Furthermore, it has many practical applications in other areas of research, some of which will be demonstrated experimentally.

2.1 Deep Learning

Deep Learning is a field that in the past couple of years has become more and more popular and nowadays is one of the most prominent tools that is used in Machine Learning. Its development has led to achieving better state-of-the-art results in a variety of areas, and it has revolutionised the application of Machine Learning to real-world problems [70]. The most prominent initial breakthroughs were in computer vision, where it was one of the first methods to achieve near-human performance on difficult object recognition challenges such as ImageNet [66, 139, 125, 128]. Another area that has seen significant improvements using Deep Learning models is Natural Language Processing (NLP), where models are now capable of generating coherent paragraphs of text, answering questions and performing simple dialogues [136, 106, 26, 105, 17]. Reinforcement Learning has also shown significant progress, where models have achieved human performance on Atari games playing directly from pixels, manipulating robotic hands with improved dexterity and more recently by achieving a long standing challenge in AI — defeating top human players in the game of Go without any supervision [90, 3, 124, 1]. Applications like image generation, voice recognition and voice generation have been improved enough that they are being used now in our everyday life through smartphones, digital assistants and other devices. [47, 98, 134, 100]. This is just the tip of the iceberg with many more areas and applications being improved upon every year.

The field of Machine Learning can be split into three major categories — Supervised Learning, Unsupervised Learning and Reinforcement Learning. Supervised Learning deals with problems where we are given a collection of N data points $\mathcal{D} = \{(\mathbf{x}^n, \mathbf{y}^n)|n = 1, \dots, N\}$, referred to as the dataset, and the goal is to learn to predict the variable $\mathbf{y}$ given $\mathbf{x}$. As an example, $\mathbf{x}$ could be an image while $\mathbf{y}$ could be a binary label of whether in the there is a dog or a cat present.

In probabilistic modelling, this would be equivalent to estimating the conditional distribution $p(\mathbf{y}|\mathbf{x})$. In Unsupervised Learning, on the other hand, we are given a dataset of observations $\mathbf{x}$, without any labels, and the goal is to learn to generate new values, which are similar to the dataset [1], or to compress the observation to a much lower-dimensional representation which captures the essential features of the data. In the probabilistic modelling literature, this would constitute directly estimating the marginal distribution $p(\mathbf{x})$ in a way that it is possible to take samples from it. Reinforcement Learning is somewhat different in that it does not have a fixed

[1] This is an oversimplification, as in general, it is possible to have data of various modalities.

dataset. Instead, it focuses on the very general problem of how does an agent learn to act in a given environment. Commonly, there is an external reward that the agent is trying to maximise. Hence, the data that it receives depends on its actions, although there are subfields which assume some form of offline supervision. There are various connections between the three categories, but this area is closest to what people outside the research community consider as Artificial Intelligence.

In this thesis, we are concerned only with Supervised Learning problems. Unless otherwise explicitly specified, it should always be assumed that the neural network model considered is trying to estimate the conditional distribution $p(\mathbf{y}|\mathbf{x})$ for a given dataset $\mathcal{D}$.

To introduce Deep Learning as a concept, we shall start with one of the first and simplest models proposed — the perceptron, which one could call a shallow network [117]. It was initially designed as a binary classification algorithm which assumed that the label can be represented by applying a threshold function to linear combination of the input features plus a bias term:

$$\mathbf{h} = \begin{cases} 1 & \text{if } \mathbf{w}^\mathsf{T}\mathbf{x} + \mathbf{b} > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{2.1}$$

In the cases where the output is an arbitrary vector rather than just a binary value, a more general computational model is used. If the input $\mathbf{x}$ is a vector in $\mathbb{R}^d$ and the target output $\mathbf{y}$ is a vector in $\mathbb{R}^k$, given a weight matrix $\mathbf{W}$ in $\mathbb{R}^{k\times d}$, a bias vector $\mathbf{b}$ in $\mathbb{R}^k$ and a non-linear activation function ϕ the output is computed as

$$\mathbf{h} = \phi(\mathbf{W}\mathbf{x} + \mathbf{b}). \tag{2.2}$$

This computationally is commonly referred to, in the context of Deep Learning, as a fully connected layer and the weight matrix and bias are its parameters. To train a layer one iteratively adjusts its parameters such that the output $\mathbf{h}$ is as close as possible to the targets $\mathbf{y}$ given $\mathbf{x}$. This procedure can be formalised as minimising the averaged square loss over the whole dataset, which is equal to

$$\frac{1}{N}\sum_n ||\mathbf{y}^n - \phi(\mathbf{W}\mathbf{x}^n + \mathbf{b})||_2^2. \tag{2.3}$$

In general, one could use any activation function ϕ; however, in practice, it is chosen based on the exact problem and some prior knowledge about the dataset. For

instance in the case of binary classification, where $\mathbf{y} = \{0, 1\}$, the standard choice for an activation function is the sigmoid — $\phi(x) = \frac{1}{1+e^{-x}}$.

A simplified description of Deep Learning models is that they are a hierarchy of multiple layers, each taking inputs from its predecessors, composing together to produce output features. Although a seemingly obvious idea, it took significant time for the research community to see its potential. The main reasons were that initially it was proven that the capabilities of the shallow perceptron are very limited [103]. Moreover, at that time hardware was not yet well developed and capable of training very deep models. In time it was theoretically proven that stacking multiple fully connected layers with non-linear activation functions has universal approximation capabilities [23]. Additionally, many more type of layers have been developed, which improved even more model capabilities. This representation of a very sophisticated model in a simple form of building blocks connected together revolutionised the field and allowed its widespread research and adoption in the research community.

The following sections formalise the construction of a deep neural network mathematically. After that, some of the most popular variants of neural networks are presented and discussed. The final section discusses how in the modern use of Deep Learning models, they are usually used together in a probabilistic framework, and their outputs represent distributions over the outputs rather than singular values.

2.1.1 Feed Forward Neural Networks

Feed Forward Neural Networks are the simplest deep learning models — a stack of multiple fully connected layers. Each one performs the computation of Equation 2.2 by taking as input the output of the previous layer. Only the final layer does not have an activation function and directly outputs its pre-activations. The reason for this will be clarified in Section 2.1.4. This can be formalised mathematically in the following way:

$$\begin{aligned} \mathbf{a}_0 &= \mathbf{x}, \\ \mathbf{h}_l &= \mathbf{W}_l \mathbf{a}_{l-1} + \mathbf{b}_l \qquad \forall l \in [1, L], \\ \mathbf{a}_l &= \begin{cases} \phi(\mathbf{h}_l) & \text{if } 0 < l < L, \\ \mathbf{h}_L & \text{if } l = L. \end{cases} \end{aligned} \tag{2.4}$$

Common activation functions ϕ used in practice are the sigmoid function described earlier, the hyperbolic tangent, the rectified linear unit (ReLu) [46] and some more sophisticated ones such as the exponential linear unit [21] and Swish [107].

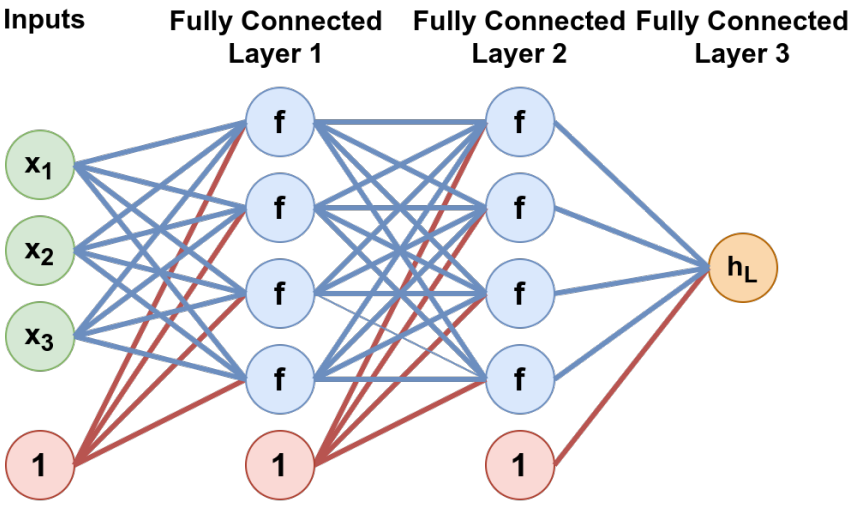


Figure 2.1: Feed Forward Neural Network. A three layer network with two hidden layers and a scalar output. The input values are marked in green, the weight matrix connections and layer activations in blue and the biases connections in red. Note that the final output does not have an activation function in our notations.

Although very simple it turns out that this model, given enough units in each layer, can approximate any function arbitrarily well [23]. However, specifically when dealing with image data, a different type of neural network have been established as the most successful.

2.1.2 Convolutional Neural Networks

Convolutional Neural Networks (CNN) have been developed specifically for solving computer vision and image recognition tasks. Rather than using a direct linear combination of the whole input image, each convolutional layer applies a discrete spatial convolution of the image with a bank of small filters, which play a similar role to the weight matrix of the fully connected layer. For each filter, this results in an image like output that also has two spatial dimensions. These image like outputs are then stacked together in the depth dimension which in turn results in another 3D array that gets propagated to the next layer. The number of stacked outputs, corresponding to the number of filters, is usually referred to as channels. The main reason for this is that the depth dimension plays a comparable role as to how the input image has red, green and blue colour channels. Similar to the fully connected layer after applying the convolutional filters, a bias term is added, and a

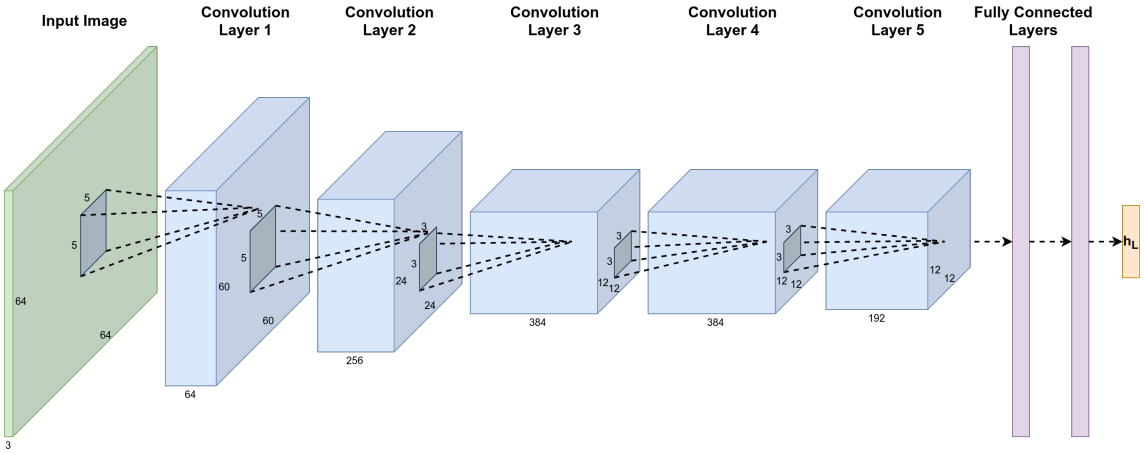


Figure 2.2: Convolutional Neural Network. A standard convolutional neural network architecture, similar to AlexNet. The input image is indicated in green, which is processed by a series of convolutional layers until finally at the end they are passed through several fully connected layers and the output is produced. The numbers show example dimensions of both the image like tensors as well as the convolution kernels that are applied.

non-linear activation function is applied. Additionally, spatial pooling is commonly used in order to shrink the spatial size of the output, while the number of channels is usually progressively increased. The sequence of convolutional layers continues until the spatial dimensions are sufficiently small. Then the full 3D array is flattened into a single vector, and a small Feed Forward Neural Network is then applied to produce the final output.

One of the main inspirations for applying convolutions was its resemblance to some functionality of the human vision as well as its capabilities of extracting features independent of where they are in the image [32]. In the literature, this is referred to as spatial translation invariance. One of the most famous CNNs, named AlexNet, for the first time, achieved better performance than other classical models on the ImageNet challenge [66]. Since its success, many more architectural developments have been done. Some improve the performance of the network across many tasks, while others are tailored to specific tasks. One example of an architectural change that generally improved the performance of Convolutional Neural Networks is the introduction of residual connections [49]. By including these connections in a network, it is now possible to train extremely deep networks and achieve better state-of-the-art results for different vision tasks, such as segmentation, as well

as for improving training times. The UNet, on the other hand, is an architecture introduced explicitly for improving the task of image segmentation [116].

Similarly, neither feed forward nor convolutional networks are well suited when working with sequential data, for which another model has been established.

2.1.3 Recurrent Neural Networks

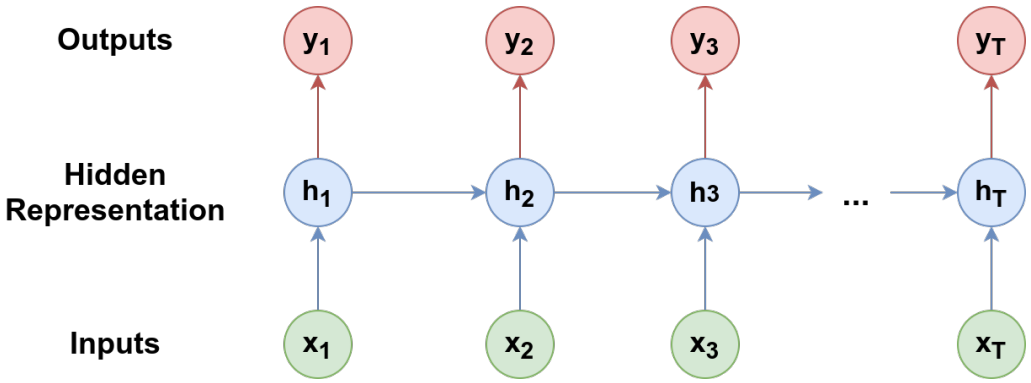


Figure 2.3: Recurrent Neural Network. The figure shows a standard recurrent network computation. The inputs are depicted in red and all of the hidden representation at each time steps in blue. The outputs and their connections are in red.

Recurrent Neural Networks (RNN) were developed as an extension to the standard Feed Forward Neural Networks for data which comes in sequences, like language or sound. Since sequential data can have variable length, it is very challenging to process such inputs with the network architectures discussed earlier. Instead of processing the whole sequence as a single vector, recurrent models process the inputs sequentially forming a chain over the sequence length as depicted in Figure 2.3. At every time step, the network has a hidden state representing all the necessary information from the past and the current inputs. It is computed by combining the hidden state at the previous time step with the inputs at the current time step using a fully connected layer and then applying a non-linear function. An important distinction with other models is that the weight matrix and bias of this layer are the same across all time steps. The actual output at time t is computed by applying a separate fully connected layer to the hidden state representation of the RNN at time t. Mathematically this can be formalised as follows:

$$\begin{aligned} \mathbf{h}_t &= \phi(\mathbf{W}_x \mathbf{x}_t + \mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{b}_h), \\ \mathbf{y}_t &= \mathbf{W}_y \mathbf{h}_t + \mathbf{b}_y. \end{aligned} \tag{2.5}$$

This specific construction is usually referred to in the literature as a vanilla Recurrent Neural Network. The reason is that the actual transformation of the pair $(\mathbf{h}_{t-1}, \mathbf{x}_t)$ into $\mathbf{h}_t$ does not need to be the same as a standard fully connected layer. Empirically, it was observed that this model struggled with learning data dependencies that span across many time steps. To address this issue, the Long-Short Term Memory (LSTM) [55] and Gated Recurrent Unit (GRU) [19] were developed, which provided alternative forms of recurrent transition functions. They showed significantly better results in tasks that the vanilla architecture was not capable of solving. More recently, the self-attention mechanism has been used to construct sequential models which empirically outperforms even the LSTM and GRU on a variety of tasks [130]. Stacking multiple such layers in a hierarchical fashion lead to the famous Transformer architecture, which is currently a basis for many of the state-of-the-art models in NLP [26, 105].

2.1.4 Modelling distributions

Originally neural networks were used to provide point estimates of the target values — the output was treated as a direct prediction and training them was done by regression. A more powerful paradigm is to use the network in a probabilistic modelling framework and use its output to parameterise a distribution over the output domain. Firstly, this allows the model to assign uncertainty to the value it is predicting. More importantly, it allows training of neural networks using Maximum Likelihood Estimation (MLE), which has several important properties such as consistency and efficiency [24, 59]. Denoting the collection of all network parameters as θ in the Supervised Learning setting this corresponds to maximising

$$\sum_n \log p(\mathbf{y}^n|\mathbf{x}^n, \theta). \tag{2.6}$$

Whereas mentioned earlier, the pairs $(\mathbf{x}^n, \mathbf{y}^n)$ are taken from the dataset $\mathcal{D}$. The standard squared loss is a special case of MLE when the distribution $p(\mathbf{y}|\mathbf{x}, \theta)$ is a Gaussian with mean $\mathbf{h}_L$ and identity covariance. Experimentation has shown, however, that Deep Learning models trained using MLE often tend to overfit the training data and perform poorly when presented with out of sample data. In light of this, it is common to additionally introduce a prior distribution over the parameters and instead of using MLE the parameters are selected by maximising the logarithm

of the posterior distribution:

$$\begin{aligned} p(\theta|\mathcal{D}) &= \frac{p(\mathcal{D}|\theta)p(\theta)}{p(\mathcal{D})} = \frac{\prod_n p(\mathbf{y}^n|\mathbf{x}^n,\theta)p(\theta)}{p(\mathcal{D})}, \\ \log p(\theta|\mathcal{D}) &= \sum_n \log p(\mathbf{y}^n|\mathbf{x}^n,\theta) + \log p(\theta) + \text{const.} \end{aligned} \tag{2.7}$$

This is a well known procedure in statistics called Maximum A Posteriori (MAP) estimation. The most common prior distribution used over the parameters of neural networks is the isotropic Gaussian.

The specific choice of the type of distribution that the network outputs is problem dependent. As an example in the case of binary classification, the standard choice would be a Bernoulli distribution, in the case of multi-class classification would be a categorical distribution, in the case when dealing with continuous values the most common choice is a Gaussian. All of the distributions used in practice are chosen from the exponential family, and commonly the network parameterises them by treating its final layer outputs $\mathbf{h}_L$ as the natural parameters of the distribution. This is the main reason why in our treatment of the model the last layer does not have an activation function, as usually it is dictated by how the natural parameters map to the actual probability density function. In the case of a Bernoulli distribution, the natural parameter is the logit, while the probability of the variable being equal to one is the sigmoid function of the logit.

2.2 Optimization

So far, we have introduced what Deep Learning models are and how in practice they can be used as part of a probabilistic framework, which defines a clear objective to be optimised. The next step is to figure out given this objective how do we find the best parameters of the network. For complicated models such as neural networks, this is very challenging as the resulting problem can be highly non-linear. In all problems considered it would be assumed that the objective function $\mathcal{E}(\theta, \mathcal{D})$ is differentiable with respect to the parameters of the model θ, as well as that the goal is to minimise it. In the case of maximisation, as in MLE and MAP estimation, the objective is defined as the negative of the original function. By definition, the gradient points in the direction of steepest ascent of the function value, hence a simple idea is to make small steps in the opposite direction. This method is called Gradient Descent (GD)

and given some initial parameters θ_0 it iteratively updates them via the equation

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta \mathcal{E}(\theta_t). \tag{2.8}$$

The scalar α is called the learning rate and defines how large the step size is at every iteration. For every smooth function, there exists a small enough α such that this procedure decreases the objective. Gradient Descent is not limited to Deep Learning and can be applied to any problem and any model as long as it is possible to calculate a gradient. Its properties have been extensively studied in the literature [15, 92, 80]. A computational drawback of this approach, however, is that computing the gradient of the objective function requires its evaluation on the whole dataset. For smaller datasets of a few thousand examples, this is not an issue, but when dealing with bigger datasets, this quickly becomes a significant bottleneck even for modern hardware. A way around the issue is to notice that the part of the loss $\mathcal{E}$ that depends on each data point is additive and independent of the others (this is true for the log-likelihood described earlier and all commonly used objective functions). In this case, that part of the loss function can be expressed as an expectation over the empirical distribution. Consequently, one can use Monte Carlo to approximate the gradient by sampling only a smaller subset of the full dataset, referred to as a minibatch. The procedure described is unbiased in the following sense

$$\mathbb{E}_{(i_1,\ldots,i_B)}\left[\frac{1}{B}\sum_b f(\mathbf{y}^{i_b}, \mathbf{x}^{i_b})\right] = \mathbb{E}_{i\sim U[1,N]}\left[f(\mathbf{y}^i, \mathbf{x}^i)\right] = \frac{1}{N}\sum_n f(\mathbf{y}^n, \mathbf{x}^n). \tag{2.9}$$

Although in expectation of the gradient is correct, this introduces noise in the estimated gradients; hence the method is called Stochastic Gradient Descent (SGD). Despite that the algorithm is now stochastic, if the learning rate is decreased appropriately, satisfying the well known Robbins-Monro conditions [115], it is still guaranteed to converge to a local minimum. This algorithm underpins most of the modern optimisation methods used to train Deep Learning models. The next section will discuss some of the more advanced techniques used for stochastic optimisation with neural networks.

2.2.1 First-Order Optimisers

First-order algorithms are probably the most well known and studied algorithms in the literature. The term comes from the fact that their update rules use only first-order information — the gradient. To distinguish it from the adaptive methods

of the next section, we will require that every update can be expressed as a linear combination of all previous gradients. It is clear that Gradient Descent is part of this family of algorithms, as well as its stochastic variant. The most widely used algorithm of this type for training Deep Learning models is the Heavy Ball method, which is often also called Gradient Descent with Momentum [104]. The initial inspiration for the method was to introduce an auxiliary variable which plays a similar role to that of momentum in a physical system. Then one can treat the objective function as the potential energy and simulate Newtonian dynamics with friction. The resulting update rules are:

$$\begin{aligned} \mathbf{v}_{t+1} &= \beta \mathbf{v}_t - \alpha \nabla_\theta \mathcal{E}(\theta_t), \\ \theta_{t+1} &= \theta_t + \mathbf{v}_t, \end{aligned} \tag{2.10}$$

where β is called the momentum coefficient[2]. Analogously to Gradient Descent, the method can be extended to use sub-sampled minibatches rather than the full dataset in order to scale to large datasets. Practitioners have found this approach to be very successful, and with proper tuning of the learning rate and momentum coefficient, it achieves state-of-the-art results on many problems. Its convergence properties in the stochastic setting have also been extensively studied [135, 76, 108].

Another important first-order method is the Accelerated Gradient Descent from the seminal work of Nesterov [95]. The authors proved that in the deterministic case, e.g. using the full gradient, it achieves the optimal quadratic convergence rate. There has been further theoretical analysis of using similar ideas in the stochastic setting. However, when applying these methods to large Deep Learning models, they rarely perform better than Gradient Descent with Momentum, hence why they are less commonly used in practice.

2.2.2 Adaptive First-Order Optimizers

In comparison to the classical first-order methods, adaptive first-order methods additionally have some form of rescaling of the gradient elementwise[3]. Most often the term is derived from the elementwise squared values of the gradients as in RmsProp, AdaDelta and Adam [138, 61]. The most widely used method from these algorithms

[2]The term $1 - \beta$ would correspond to the friction coefficient in the physics interpretation.

[3]This is equivalent to a diagonal preconditioner, but we want to distinguish these methods from second-order optimizers explicitly.

for training Deep Learning models is Adam, arguably on par with the Heavy Ball method in terms of its use. Its updates are based on tracking a moving average of the first moment of the gradients and rescaling by the square root of a moving average of the second moments of the gradients:

$$\begin{aligned} \boldsymbol{\mu}_{t+1} &= \beta_1 \boldsymbol{\mu}_t + (1-\beta_1)\nabla_\theta \mathcal{E}(\theta_t), \\ \tilde{\boldsymbol{\mu}}_{t+1} &= \frac{\boldsymbol{\mu}_{t+1}}{1-\beta_1^t}, \\ \mathbf{v}_{t+1} &= \beta_2 \mathbf{v}_t + (1-\beta_2)\nabla_\theta \mathcal{E}(\theta_t)^2, \\ \tilde{\mathbf{v}}_{t+1} &= \frac{\mathbf{v}_{t+1}}{1-\beta_2^t}, \\ \theta_{t+1} &= \theta_t - \alpha \frac{\tilde{\boldsymbol{\mu}}_{t+1}}{\sqrt{\tilde{\mathbf{v}}_{t+1}} + \epsilon}. \end{aligned} \tag{2.11}$$

In the expressions above all mathematical operations are applied elementwise. In practice, it performs consistently very well and with very little fine-tuning, despite some issues with its behaviour in particular cases having been raised [111].

2.2.3 Second-Order Optimizers

Second-order methods have a long history in optimisation of deterministic functions. The main difference with first-order methods is that they uses the second-order derivatives matrix — the Hessian — to construct their steps. The core idea of these methods is to construct a local quadratic approximation by Taylor expanding the objective function around the current parameter setting and then solving it:

$$\mathcal{E}(\theta+\delta) \approx \mathcal{E}(\theta) + \delta^\mathsf{T} \nabla_\theta \mathcal{E} + \frac{1}{2}\delta^\mathsf{T} \nabla^2_{\theta_t} \mathcal{E} \delta. \tag{2.12}$$

The solution is simply $\delta^* = -\left(\nabla^2_{\theta_t}\mathcal{E}\right)^{-1} \nabla_\theta \mathcal{E}$ which is the update for the classical Newton's method. However, in this formulation, if the Hessian matrix is not Positive Semi-Definite (PSD) then this local quadratic function does not have a minimal value and minimising it might lead to steps in directions that do not decrease the loss. Additionally, this can converge to a saddle point rather than a minimum. Several algorithmic developments address these issues, such as line search and trust-region methods. An alternative approach is not to use the Hessian matrix at all, but find a matrix that has similar properties and is PSD by construction. This is what will broadly be called a curvature matrix, and its estimated value would be denoted with $\mathbf{C}$. Replacing the Hessian in the quadratic Taylor expansion by this matrix

guarantees that the resulting function has a minimum and we can then update the parameters using the solution:

$$\theta_{t+1} = \theta_t - \alpha \mathbf{C}_t^{-1} \nabla_\theta \mathcal{E}(\theta_t). \tag{2.13}$$

Practical algorithms of this kind execute more sophisticated updates, for instance, by automatically choosing the step size based on different criteria [132, 4]. The adaptive first-order methods can be thought of as building a diagonal curvature matrix based only on gradient information. One example of such a curvature matrix is the Gauss-Newton, which initially has been proposed only for non-linear least squares. It will be of central interest of this work and will be discussed in great details in Chapter 3. Another family of algorithms that construct an alternative matrix to the Hessian are the so called Quasi-Newton methods. They build up iteratively a curvature matrix using properties of the Hessian. The most notable ones are the Broyden-Fletcher-Goldfarb-Shanno (BFGS) and its limited memory variant LBFGS [12, 97]. Due to its significantly lower memory usage, for a while, LBFGS was one of the most popular algorithms for non-linear optimisation. The main challenge of applying any of these methods is that they often require significantly more memory than first-order methods, and they are not developed to deal with stochasticity in either the gradient or the curvature matrix. The second point is of crucial importance as it makes them unpractical for large datasets. In recent years there has been an increasing interest in developing Stochastic Quasi-Newton methods which can be applicable to large models such as neural networks [14, 91, 131]. Nevertheless, there has not yet been developed any Stochastic Quasi-Newton algorithm which manages to achieve better or faster performance on Deep Learning models when compared to the more popular first-order methods.

Recently there has been increasing interest in developing second-order optimisers specifically for Deep Learning. A Hessian-free algorithm has been developed, that uses automatic differentiation to compute Hessian-vector products cheaply and performs the gradient preconditioning via a truncated conjugate-gradient method [81]. The results from this work demonstrated that second-order methods are capable of training neural networks on problems where first-order methods struggled to find good solutions. Unfortunately, the method requires very large batch sizes and is still not performing well in terms of computation time. Finally, as will be shown in this work, several methods build a block-diagonal approximation to a curvature matrix. A detailed discussion of such methods is deferred to later chapters.

2.3 Model Uncertainty and its applications

The standard setting in which Deep Learning models are trained is MLE or MAP where the result of the training procedure is just a single parameter value. Although these results show excellent performance on data similar to the training data they have been observed to be overconfident in predictions on data that is very different from what they have been trained on. Consider, an image classifier that has been trained to recognise car models from images. When, however, presented with a plane, it would still provide us with a prediction of which car model it is. It would be beneficial if the model can also provide us with a level of uncertainty that it has about its prediction, which in this example is likely to be high as it knows nothing about planes. The measure of uncertainty is especially important when trying to apply models in real-world applications, where decisions based on their predictions can be of great importance. Examples of such applications are autonomous vehicles and medical treatment for patients, where real lives could be at stake if the wrong decisions are made. The uncertainty in the model predictions can be split into types:

- aleatoric uncertainty — this arises from noise in the data generation process, e.g. from corrupted labels or inherited noise in measurement instruments.
- epistemic uncertainty — this arises from the uncertainty of the model parameters, e.g. there are many parameters values that could explain the observed data

A trained model usually captures well aleatoric uncertainty as the training data is expected to be noisy and contain information about the noise. However, the point estimate that is the result of standard training is not capable of expressing any form of epistemic uncertainty. The Bayesian framework provides a principled way of avoiding such problems by treating the parameters of the model as an unknown quantity and integrating over all possible values of the posterior distribution. For many practical models, the full posterior distribution is intractable and instead some approximation of it has to be made in order to perform the integration. Variational approaches try to find simpler tractable distributions that well approximate it via divergence minimisation [62]. Another common approach is to introduce, during training, stochastic noise into the model and perform training with it [126, 35, 5]. At test-time instead of removing this noise process, it is possible to perform Monte Carlo sampling and get multiple predictions over the outputs. One can relate this procedure to variational approaches under conditions which, however,

are not met in practice [57]. The most popular strategy to get epistemic uncertainty is to find multiple MAP solutions by training a model from different initial points and constructing an ensemble from the resulting parameters [28, 73, 25, 67]. This is both the simplest and the empirically best performing technique. It is somewhat orthogonal to the other approaches as it can be combined with them, by making a mixture of approximate distributions (each mixture corresponds to a single ensemble member). Hence, in this work, ensemble methods will not be discussed in any more details. Other applications of prediction uncertainty is active learning, where the uncertainty of the model can be used to select the optimal unlabelled data which to be given to a human to annotate.

The applications of having an approximate posterior distribution over the parameters go beyond using them for predictive uncertainty. Online learning is a field where the model observes multiple tasks in a row without being able to return to the previous tasks data. This is similar to how people often learn different tasks in sequence and at the end are capable of then executing each one successfully. In practice, it is challenging to train Deep Learning models in this setting as they tend to quickly forget the information that they have learned in earlier tasks. By having an approximate distribution to the posterior and continuously updating it based on the new task, the model is capable of retaining information from all previous tasks. Another avenue where the posterior distribution can be useful is Reinforcement Learning, where it can be used in order to achieve much better exploration lower regret via approximate Thompson Sampling [122, 36, 16, 60, 56].

Chapter 3

Curvature Matrices of Feed Forward Neural Networks

The chapter lays out all theoretical foundations which will be the core of any of the practical algorithms developed in the later chapters. The first three sections present the exact problem setup and mathematical background that the reader should be familiar with in order to understand the derivations thoroughly thereafter. Firstly, the Hessian matrix of a Feed Forward Neural Network will be analysed, as it is the basis of all approximate curvature matrices. As demonstrated in Lemma 3.15 for a single input, it has a very particular structure. This result will be used to prove several interesting facts about the objective function of neural networks when using piecewise linear activation functions. However, since it is not Positive Semi-Definite in general, Section 3.5 will analyse the Generalised Gauss-Newton matrix and its relationship to the Hessian and Fisher matrices. The final section of the chapter discusses the implication of having a finite dataset on the rank of this curvature matrix.

The work in this chapter was done in collaboration with my supervisor Prof. David Barber at University College London. Most of the results have been previously publish [10], however, I have tried to formalise them better and describe the intuitions behind them in much more details than it was possible in the short page limit of a conference publication.

3.1 Modelling formulation and assumptions

The main focus of all theoretical results will be training a Deep Learning model on a Supervised Learning task. In light of this, it is assumed that a dataset $\mathcal{D}$ is given, consisting of N data points. Each pair $(\mathbf{x}^n, \mathbf{y}^n)$, where $\mathbf{x}^n \in \mathbb{R}^{d_x}$ and $\mathbf{y}^n \in \mathbb{R}^{d_y}$, are assumed to have been sampled from some unknown distribution $p_{\text{true}}(\mathbf{x}, \mathbf{y})$. The upper indexing with the variable n would be used to indicate indexing over the data points in $\mathcal{D}$. The empirical distribution of the data is defined as a mixture of delta functions around each data point and will be denoted as $p_d(\mathbf{x}, \mathbf{y})$:

$$p_d(\mathbf{x}, \mathbf{y}) = \frac{1}{N} \sum_n \delta(\mathbf{x} - \mathbf{x}^n)\delta(\mathbf{y} - \mathbf{y}^n). \tag{3.1}$$

The particular objective function that will be optimised is assumed to be provided as part of the modelling problem. There can be many variants of the exact functional form, but the discussion will be restricted to loss functions of the form

$$\mathcal{E}(\theta, \mathcal{D}) = \frac{1}{N} \sum_n \mathcal{L}_\theta(\mathbf{x}^n, \mathbf{y}^n) + \overline{\lambda}\Omega(\theta). \tag{3.2}$$

The loss for a single data point would be denoted by $\mathcal{L}^n$, keeping in mind that it depends on θ, $\mathbf{x}^n$ and $\mathbf{y}^n$. In the cases where the derivation is invariant to the specific choice of the data point values, the index n will be dropped, and only $\mathcal{L}$ will be used. The average values over the whole dataset of quantities that are data point dependent will be denoted via an over-line, for instance the average dataset loss is

$$\overline{\mathcal{L}} = \frac{1}{N} \sum_n \mathcal{L}_\theta(\mathbf{x}^n, \mathbf{y}^n). \tag{3.3}$$

Additionally, the regulariser $\Omega(\theta)$ should be a convex function of the parameters. In the case when the neural network is part of a statistical framework, it is assumed that it approximates the conditional distribution $p(\mathbf{y}|\mathbf{x})$. To distinguish this from the true data distribution, this approximation will be denoted as $p_\theta(\mathbf{y}|\mathbf{x})$. In this setting, since the objective is minimised, the loss for a single data point is the negative log-likelihood:

$$\mathcal{L}^n = -\log p_\theta(\mathbf{y}^n|\mathbf{x}^n). \tag{3.4}$$

In the case of MAP estimation, the regulariser is nothing more than the negative log prior:

$$\Omega(\theta) = -\log p(\theta), \tag{3.5}$$

and then its coefficient $\overline{\lambda}$ is equal to $\frac{1}{N}$. The construction defines the objective as the log-posterior rescaled by a factor of one over the number of data points. In all practical cases considered in later chapters, the prior will be an isotropic Gaussian distribution, whose covariance is the identity multiplied by σ^2. The full objective function then is

$$\mathcal{E}(\theta, \mathcal{D}) = \frac{1}{N} \sum_n \mathcal{L}^n + \frac{\lambda}{2} ||\theta||_2^2, \tag{3.6}$$

where for convenience the new variable λ is defined as $\frac{\overline{\lambda}}{\sigma^2}$.

3.2 The Kronecker product and related properties

The Kronecker product plays an essential role in the structure of many of the curvature matrices and their approximations. As such, the reader needs to familiarise himself with it and its various properties in order to be able to understand the mathematical derivations easily.

Definition 3.1. *Given a $m \times n$ matrix $\mathbf{A}$ and a $p \times q$ matrix $\mathbf{B}$ the Kronecker product between $\mathbf{A}$ and $\mathbf{B}$, denoted as $\mathbf{A} \otimes \mathbf{B}$, is a block matrix consisting of $m \times n$ blocks each of size $p \times q$ and the ij^{th} block is equal to $\mathbf{A}_{ij}\mathbf{B}$.*

For clarity this is depicted below

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} \mathbf{A}_{11}\mathbf{B} & \mathbf{A}_{12}\mathbf{B} & \dots & \mathbf{A}_{1,n}\mathbf{B} \\ \mathbf{A}_{21}\mathbf{B} & \mathbf{A}_{22}\mathbf{B} & \dots & \mathbf{A}_{2,n}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}_{m1}\mathbf{B} & \mathbf{A}_{m2}\mathbf{B} & \dots & \mathbf{A}_{m,n}\mathbf{B} \end{bmatrix}. \tag{3.7}$$

3.2.1 Properties

From the definition, it is clear that the Kronecker product is a bilinear map between the two matrices $\mathbf{A}$ and $\mathbf{B}$. As such one can prove the following associative properties:

- $(\mathbf{A} + \mathbf{C}) \otimes \mathbf{B} = \mathbf{A} \otimes \mathbf{B} + \mathbf{C} \otimes \mathbf{B}$.
- $\mathbf{A} \otimes (\mathbf{B} + \mathbf{C}) = \mathbf{A} \otimes \mathbf{B} + \mathbf{A} \otimes \mathbf{C}$.
- $(\alpha\mathbf{A}) \otimes \mathbf{B} = \mathbf{A} \otimes (\alpha\mathbf{B}) = \alpha\mathbf{A} \otimes \mathbf{B}$.
- $(\mathbf{A} \otimes \mathbf{B}) \otimes \mathbf{C} = \mathbf{A} \otimes (\mathbf{B} \otimes \mathbf{C}) = \mathbf{A} \otimes \mathbf{B} \otimes \mathbf{C}$.

A few more properties of the Kronecker product that are used in this work are listed below.

Property 3.2. *If* $\mathbf{a}$ *and* $\mathbf{b}$ *are vectors then their outer product can be written in terms of a Kronecker product in the following way:*

$$\mathbf{a}\,\mathbf{b}^\mathsf{T} = \mathbf{a} \otimes \mathbf{b}^\mathsf{T} = \mathbf{b}^\mathsf{T} \otimes \mathbf{a}.$$

Proof. Directly follows from Definition 3.1. □

Property 3.3. *Given a* $m \times n$ *matrix* $\mathbf{A}$ *and a* $p \times q$ *matrix* $\mathbf{B}$ *the transpose of the Kronecker product between* $\mathbf{A}$ *and* $\mathbf{B}$ *is the Kronecker product of the individual matrices transposed:*

$$(\mathbf{A} \otimes \mathbf{B})^\mathsf{T} = \mathbf{A}^\mathsf{T} \otimes \mathbf{B}^\mathsf{T}.$$

Proof. Directly follows from Definition 3.1. □

Property 3.4. *Given a* $m \times m$ *diagonal matrix* $\mathbf{A}$ *and a* $p \times p$ *diagonal matrix* $\mathbf{B}$ *then the Kronecker product* $\mathbf{A} \otimes \mathbf{B}$ *is a diagonal matrix as well.*

Proof. Directly follows from Definition 3.1. □

Property 3.5 (Mixed Matrix Property)**.** *Given a* $m \times n$ *matrix* $\mathbf{A}$*,* $n \times r$ *matrix* $\mathbf{C}$ *and a* $p \times q$ *matrix* $\mathbf{B}$*,* $q \times s$ *matrix* $\mathbf{D}$ *the following identity, referred to as the mixed-product property, holds:*

$$(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) = (\mathbf{AC}) \otimes (\mathbf{BD}).$$

Proof. Since both $(\mathbf{A} \otimes \mathbf{B})$ and $(\mathbf{C} \otimes \mathbf{D})$ are block-matrices with compatible block sizes, we can directly analyse for the resulting $m \times r$ ij^{th} block of the product:

$$\mathbf{P}_{ij} = \sum_k (\mathbf{A}_{ik}\mathbf{B})(\mathbf{C}_{kj}\mathbf{D}) = \left(\sum_k \mathbf{A}_{ik}\mathbf{C}_{kj}\right)\mathbf{BD} = (\mathbf{AC})_{ij}\mathbf{BD}.$$

□

Property 3.6. *Given a* $m \times m$ *unitary matrix* $\mathbf{A}$ *and a* $p \times p$ *unitary matrix* $\mathbf{B}$ *then the Kronecker product* $\mathbf{A} \otimes \mathbf{B}$ *is also a unitary matrix.*

Proof. $(\mathbf{A} \otimes \mathbf{B})^\mathsf{T}(\mathbf{A} \otimes \mathbf{B}) = (\mathbf{A}^\mathsf{T} \otimes \mathbf{B}^\mathsf{T})(\mathbf{A} \otimes \mathbf{B}) = (\mathbf{A}^\mathsf{T}\mathbf{A}) \otimes (\mathbf{B}^\mathsf{T}\mathbf{B}) = \mathbf{I} \otimes \mathbf{I} = \mathbf{I}$. □

Property 3.7. *Given a $m \times n$ matrix $\mathbf{A}$ with Singular Value Decomposition $\mathbf{U_A \Sigma_A V_A^T}$ and a $p \times q$ matrix $\mathbf{B}$ with Singular Value Decomposition $\mathbf{U_B \Sigma_B V_B^T}$ then the Singular Value Decomposition of the Kronecker product $\mathbf{A} \otimes \mathbf{B}$ is:*

$$(\mathbf{A} \otimes \mathbf{B}) = (\mathbf{U_A} \otimes \mathbf{U_B})(\mathbf{\Sigma_A} \otimes \mathbf{\Sigma_B})(\mathbf{V_A} \otimes \mathbf{V_B})^\mathsf{T}.$$

Proof. The factorisation follows from Property 3.5, the fact that $\mathbf{U_A} \otimes \mathbf{U_B}$ and $\mathbf{V_A} \otimes \mathbf{V_B}$ are unitary matrices follow from Property 3.6, finally $\mathbf{\Sigma_A} \otimes \mathbf{\Sigma_B}$ is diagonal from Property 3.4. □

Property 3.8. *Given a $m \times m$ square matrix $\mathbf{A}$ and a $p \times p$ square matrix $\mathbf{B}$ then the Kronecker Product $\mathbf{A} \otimes \mathbf{B}$ raised to the power n is equal to the Kronecker product of each individual matrix raised to the power n:*

$$(\mathbf{A} \otimes \mathbf{B})^n = \mathbf{A}^n \otimes \mathbf{B}^n.$$

Proof. Directly follows from Property 3.7. □

Property 3.9. *Given a $m \times m$ square matrix $\mathbf{A}$ and a $p \times p$ square matrix $\mathbf{B}$ then the Kronecker product $\mathbf{A} \otimes \mathbf{B}$ has a rank equal to the product of the rank of $\mathbf{A}$ and the rank of $\mathbf{B}$.*

$$\mathbf{rank}\,(\mathbf{A} \otimes \mathbf{B}) = \mathbf{rank}\,(\mathbf{A})\,\mathbf{rank}\,(\mathbf{B})\,.$$

Proof. Directly follows from Property 3.7. □

Property 3.10. *Given a $m \times m$ square matrix $\mathbf{A}$ and a $p \times p$ square matrix $\mathbf{B}$ then the determinant of the Kronecker product $\mathbf{A} \otimes \mathbf{B}$ is equal to the determinant of $\mathbf{A}$ to the power p times the determinant of $\mathbf{B}$ to the power m.*

$$\mathbf{det}\,(\mathbf{A} \otimes \mathbf{B}) = \mathbf{det}\,(\mathbf{A})^p\,\mathbf{det}\,(\mathbf{B})^m\,.$$

Proof. From Property 3.7 it follows that the determinant is equal to the determinant of the diagonal matrix $\mathbf{\Sigma_A} \otimes \mathbf{\Sigma_B}$. The elements of this matrix are $[\mathbf{\Sigma_A}]_{ii}[\mathbf{\Sigma_B}]_{jj}$ for every pair of indices i and j. Hence, each element $[\mathbf{\Sigma_A}]_{ii}$ appears as many times as the dimensionality of $\mathbf{B}$, which is p, and analogously each $[\mathbf{\Sigma_B}]_{jj}$ appears exactly m times. □

Definition 3.11. *The Khatri-Rao product of two block matrices* $\mathbf{A}$ *and* $\mathbf{B}$*, each one consisting of* $m \times n$ *blocks, is a block matrix whose* k, l *block is the Kronecker product of the* k, l *blocks of* $\mathbf{A}$ *and* $\mathbf{B}$*. The operation will be denoted by* $\mathbf{A} * \mathbf{B}$*.*

For clarity the definition is depicted below:

$$\mathbf{A} * \mathbf{B} = \begin{bmatrix} \mathbf{A}_{1,1} \otimes \mathbf{B}_{1,1} & \mathbf{A}_{1,2} \otimes \mathbf{B}_{1,2} & \dots & \mathbf{A}_{1,n} \otimes \mathbf{B}_{1,n} \\ \mathbf{A}_{2,1} \otimes \mathbf{B}_{2,1} & \mathbf{A}_{2,2} \otimes \mathbf{B}_{2,2} & \dots & \mathbf{A}_{2,n} \otimes \mathbf{B}_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}_{m,1} \otimes \mathbf{B}_{m,1} & \mathbf{A}_{m,2} \otimes \mathbf{B}_{m,2} & \dots & \mathbf{A}_{m,n} \otimes \mathbf{B}_{m,n} \end{bmatrix}. \tag{3.8}$$

3.2.2 The vectorization operator

When dealing with parameters which are matrices, it is sometimes convenient to still treat them as vectors. For this purpose we introduce the vectorization operator:

Definition 3.12. *The vectorization operator is defined as an isomorphism between the space of real matrices of size* $m \times n$ *to the space of real vectors of size* mn *by stacking the columns of the matrix sequentially into a single vector. Formally* $\mathbf{vec}_{m,n}() : \mathbb{R}^{m \times n} \to \mathbb{R}^{mn}$ *such that:*

$$\mathbf{vec}_{m,n}(\mathbf{A}) = \begin{bmatrix} \mathbf{A}_{\cdot,1} \\ \dots \\ \mathbf{A}_{\cdot,N} \end{bmatrix}.$$

For convenience, when the input to the operator is a vector rather than a matrix, we define it as the identity map.

Since $\mathbf{vec}_{m,n}()$ is an isomorphic map, note that its inverse exists. The operator has one important property that would be often used throughout the thesis:

Property 3.13. *Given a vector* $\mathbf{a} \in \mathbb{R}^m$ *and a vector* $\mathbf{b} \in \mathbb{R}^n$ *the following equality holds:*

$$\mathbf{vec}_{m,n}(\mathbf{a}\,\mathbf{b}^\mathsf{T}) = \mathbf{a} \otimes \mathbf{b}\,.$$

Proof. Follows directly from Definition 3.12. □

Property 3.14. *Given a $m \times n$ matrix $\mathbf{A}$, a $p \times q$ matrix $\mathbf{B}$ and a $n \times p$ matrix $\mathbf{X}$ the following equality holds:*

$$\mathbf{vec}_{m,q}(\mathbf{AXB}) = (\mathbf{B}^\mathsf{T} \otimes \mathbf{A})\mathbf{vec}_{n,p}(\mathbf{X}).$$

Proof. Let's consider the i^{th} block of the right hand side, which is a vector of size m:

$$\mathbf{P}_i = [\mathbf{B}_{11}\mathbf{A}, \mathbf{B}_{21}\mathbf{A}, \ldots, \mathbf{B}_{n,i}\mathbf{A}]\mathbf{vec}\,(\mathbf{X}) = \sum_j \mathbf{B}_{j,i}\mathbf{A}\mathbf{X}_{j,\cdot} = \mathbf{AXB}_{\cdot,i}.$$

which is indeed the i^{th} column of the matrix $\mathbf{AXB}$. □

Notations note: Through the text whenever the matrix sizes m and n are clear from the context the subscripts will be omitted and we will use $\mathbf{vec}\,(\cdot)$ instead of $\mathbf{vec}_{m,n}(\cdot)$.

3.3 Neural network notations

As discussed in the Section 2.1, there are many different Deep Learning models used in practice. However, in this chapter, the main focus will be on investigating the curvature matrices of Feed Forward Neural Networks. The reason for this choice is that they are the most simple neural networks, whose properties, however, are similar to those of more complicated architectures. Also, the fact that they have the potential to be universal function approximators implies that some of the conclusions drawn from them can be extended to other architectures as well. Therefore, from this point forward unless explicitly specified, it is always assumed that the model under consideration is a Feed Forward Neural Network.

The original notations introduced in Equation 2.4 are somewhat inconvenient for presenting our results. In this regard, let us introduce the concatenated weight matrix $\hat{W}_l$ for each layer l. It is defined by stacking the bias as an additional last column to the standard weight matrix:

$$\hat{\mathbf{W}}_l = [\mathbf{W}; \mathbf{b}]\,. \tag{3.9}$$

Equipped with this, the original recursive definition of a Feed Forward Neural Network can be expressed in an alternative form, by additionally appending a unit with

a value of 1 to all activations except for the final layer:

$$\begin{aligned}
\hat{\mathbf{a}}_0 &= \begin{bmatrix} \mathbf{a}_0 \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}, \\
\mathbf{h}_l &= \hat{\mathbf{W}}_l \hat{\mathbf{a}}_{l-1} \qquad\qquad \forall l \in [1, L], \\
\hat{\mathbf{a}}_l &= \begin{cases} \begin{bmatrix} \mathbf{a}_l \\ 1 \end{bmatrix} = \begin{bmatrix} f(\mathbf{h}_l) \\ 1 \end{bmatrix} & \text{if } l < L, \\ \mathbf{a}_L = \mathbf{h}_L & \text{if } 0 < l = L. \end{cases}
\end{aligned} \tag{3.10}$$

When taking derivatives, it is advantageous to represent parameters as vectors rather than matrices or other tensors. Hence, the flattened parameters for layer l are defined via the vectorization operator, see Section 3.2.2:

$$\mathbf{w}_l = \mathbf{vec}\left(\hat{\mathbf{W}}^l\right). \tag{3.11}$$

This further allows us to define the equation for the pre-activations $\mathbf{h}_l$ as a linear map of $\mathbf{w}_l$ using the properties of the Kronecker product:

$$\begin{aligned}
\mathbf{h}_l &= \hat{\mathbf{W}}_l \hat{\mathbf{a}}_{l-1} = \mathbf{vec}\left(\hat{\mathbf{W}}_l \hat{\mathbf{a}}_{l-1}\right) \\
&= \mathbf{vec}\left(\mathbf{I}\hat{\mathbf{W}}_l \hat{\mathbf{a}}_{l-1}\right) = \left(\hat{\mathbf{a}}_{l-1}^\mathsf{T} \otimes \mathbf{I}\right) \mathbf{vec}\left(\hat{\mathbf{W}}_l\right) \\
&= \left(\hat{\mathbf{a}}_{l-1}^\mathsf{T} \otimes \mathbf{I}\right) \mathbf{w}_l.
\end{aligned} \tag{3.12}$$

It is now possible to formally define the collection of all parameters as the concatenation of all layer's $\mathbf{w}_l$ variables — $\theta = [\mathbf{w}_1^\mathsf{T}; \mathbf{w}_2^\mathsf{T}; \ldots; \mathbf{w}_L^\mathsf{T}]^\mathsf{T}$.

Finally, we want to highlight an important distinctions, when manipulating derivative expressions, between the gradient and Jacobian matrix. In Machine Learning in most cases practitioners are working with gradients, they are commonly treated as vectors [1]. However, when writing the multi-dimensional chain rule, it is much more suitable to use the correct mathematical definition of Jacobian matrices. In order to emphasis and make this distinction clear for the reader we will use the following convention:

> Given a vector valued function $\mathbf{f} : \mathbb{R}^d \to \mathbb{R}^k$ we will denote the Jacobian matrix of $\mathbf{f}$ with respect to its inputs $\mathbf{x}$ with $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ defined as
>
> $$\left[\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right]_{i,j} = \frac{\partial \mathbf{f}_i}{\partial \mathbf{x}_j}.$$

[1] Formally, gradients are covectors.

In the special case, where $\mathbf{f}$ is a scalar function, e.g. $k = 1$, then we will denote the vector of partial derivatives of $\mathbf{f}$ with respect to its inputs $\mathbf{x}$ with $\nabla_{\mathbf{x}}\mathbf{f}$ defined as

$$[\nabla_{\mathbf{x}}\mathbf{f}]_j = \frac{\partial \mathbf{f}}{\partial \mathbf{x}_j}.$$

Importantly, when $\mathbf{f}$ is a scalar function the Jacobian represents a row vector (its a $1 \times d$ matrix) while the gradient is a column vector. Using these notations, we are finally ready to dive into our theoretical analysis.

3.4 The structure of the sample Hessian matrix

A central idea throughout this thesis is the approximation of the Hessian matrix of the objective function with respect to the parameters θ of a Feed Forward Neural Network. From the assumption in Equation 3.2 the following equality for the Hessian follows:

$$\frac{\partial^2 \mathcal{E}}{\partial \theta^2} = \frac{1}{N}\sum_n \frac{\partial^2 \mathcal{L}^n}{\partial \theta^2} + \lambda \frac{\partial^2 \Omega}{\partial \theta^2}. \tag{3.13}$$

Since the regulariser is convex, it is assumed that there exists a known analytical expression for its second-order derivative. This is true for all practical cases considered later, where this is nothing more than the log-density of an isotropic Gaussian and its Hessian matrix is a scaled identity. In this regard, the expression for our analysis is the *sample* Hessian, e.g. the matrix of second-order derivatives for a single data point:

$$\mathbf{H}_\theta = \frac{\partial^2 \mathcal{L}}{\partial \theta^2}. \tag{3.14}$$

Using the chain rule and Property 3.5 we can write the Jacobian of the objective $\mathcal{L}$ with respect to $\mathbf{w}_l$ as:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_l} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_l}\frac{\partial \mathbf{h}_l}{\partial \mathbf{w}_l} = \left(1 \otimes \frac{\partial \mathcal{L}}{\partial \mathbf{h}_l}\right)\left(\hat{\mathbf{a}}_{l-1}^\mathsf{T} \otimes \mathbf{I}\right) = \hat{\mathbf{a}}_{l-1}^\mathsf{T} \otimes \frac{\partial \mathcal{L}}{\partial \mathbf{h}_l}. \tag{3.15}$$

Since $\hat{\mathbf{a}}_{l-1}$ is the input to layer l observe that for $k \geq l$ it does not depend on either $\mathbf{w}_l$ or $\mathbf{w}_k$ and so the following equalities hold:

$$\begin{aligned} \frac{\partial \hat{\mathbf{a}}_{l-1}}{\partial \mathbf{w}_k} &= 0, \\ \frac{\partial}{\mathbf{w}_k}\left(\frac{\partial \hat{\mathbf{a}}_{k-1}}{\partial \mathbf{h}_l}\right) &= 0. \end{aligned} \tag{3.16}$$

To find the diagonal blocks of $\mathbf{H}_\theta$ we differentiate the Jacobian one more time with respect to $\mathbf{w}_l$:

$$\begin{aligned}\frac{\partial^2 \mathcal{L}}{\partial \mathbf{w}_l{}^2} &= \frac{\partial}{\mathbf{w}_l}\left(\hat{\mathbf{a}}_{l-1}^\intercal \otimes \frac{\partial \mathcal{L}}{\partial \mathbf{h}_l}\right) = \hat{\mathbf{a}}_{l-1}^\intercal \otimes \frac{\partial^2 \mathcal{L}}{\partial \mathbf{w}_l \partial \mathbf{h}_l} = \hat{\mathbf{a}}_{l-1}^\intercal \otimes \left(\frac{\partial \mathbf{h}_l}{\partial \mathbf{w}_l}^\intercal \frac{\partial^2 \mathcal{L}}{\partial \mathbf{h}_l{}^2}\right) \\ &= \hat{\mathbf{a}}_{l-1}^\intercal \otimes \left(\left(\hat{\mathbf{a}}_{l-1}^\intercal \otimes I\right)^\intercal \frac{\partial^2 \mathcal{L}}{\partial \mathbf{h}_l{}^2}\right) = \hat{\mathbf{a}}_{l-1}^\intercal \otimes \left(\hat{\mathbf{a}}_{l-1} \otimes \frac{\partial^2 \mathcal{L}}{\partial \mathbf{h}_l{}^2}\right) \\ &= \hat{\mathbf{a}}_{l-1}^\intercal \otimes \hat{\mathbf{a}}_{l-1} \otimes \frac{\partial^2 \mathcal{L}}{\partial \mathbf{h}_l{}^2} = \left(\hat{\mathbf{a}}_{l-1}\hat{\mathbf{a}}_{l-1}^\intercal\right) \otimes \frac{\partial^2 \mathcal{L}}{\partial \mathbf{h}_l{}^2}.\end{aligned} \tag{3.17}$$

To find the off diagonal blocks of $\mathbf{H}_\theta$ we differentiate the Jacobian one more time with respect to $\mathbf{w}_k$ for $k > l$:

$$\begin{aligned}\frac{\partial^2 \mathcal{L}}{\partial \mathbf{w}_k \partial \mathbf{w}_l} &= \frac{\partial}{\mathbf{w}_k}\left[\hat{\mathbf{a}}_{l-1}^\intercal \otimes \frac{\partial \mathcal{L}}{\partial \mathbf{h}_l}\right] = \hat{\mathbf{a}}_{l-1}^\intercal \otimes \frac{\partial}{\mathbf{w}_k}\left[\frac{\partial \mathcal{L}}{\partial \mathbf{h}_k}\frac{\partial \mathbf{h}_k}{\partial \hat{\mathbf{a}}_{k-1}}\frac{\partial \hat{\mathbf{a}}_{k-1}}{\partial \mathbf{h}_l}\right] \\ &= \hat{\mathbf{a}}_{l-1}^\intercal \otimes \frac{\partial}{\mathbf{w}_k}\left[\frac{\partial \mathcal{L}}{\partial \mathbf{h}_k}\hat{\mathbf{W}}^k\frac{\partial \hat{\mathbf{a}}_{k-1}}{\partial \mathbf{h}_l}\right] = \hat{\mathbf{a}}_{l-1}^\intercal \otimes \frac{\partial}{\mathbf{w}_k}\left[\left(\frac{\partial \hat{\mathbf{a}}_{k-1}}{\partial \mathbf{h}_l}^\intercal \otimes \frac{\partial \mathcal{L}}{\partial \mathbf{h}_k}\right)\mathbf{w}_k\right] \\ &= \hat{\mathbf{a}}_{l-1}^\intercal \otimes \left[\frac{\partial^2 \mathcal{L}}{\partial \mathbf{w}_k \partial \mathbf{h}_k}\hat{\mathbf{W}}^k\frac{\partial \hat{\mathbf{a}}_{k-1}}{\partial \mathbf{h}_l} + \left(\frac{\partial \hat{\mathbf{a}}_{k-1}}{\partial \mathbf{h}_l}^\intercal \otimes \frac{\partial \mathcal{L}}{\partial \mathbf{h}_k}\right)^\intercal\right] \\ &= \hat{\mathbf{a}}_{l-1}^\intercal \otimes \left[\frac{\partial \mathbf{h}_k}{\partial \mathbf{w}_k}^\intercal \frac{\partial^2 \mathcal{L}}{\partial \mathbf{h}_k{}^2}\frac{\partial \mathbf{h}_k}{\partial \hat{\mathbf{a}}_{k-1}}\frac{\partial \hat{\mathbf{a}}_{k-1}}{\partial \mathbf{h}_l} + \left(\frac{\partial \hat{\mathbf{a}}_{k-1}}{\partial \mathbf{h}_l} \otimes \frac{\partial \mathcal{L}}{\partial \mathbf{h}_k}^\intercal\right)\right] \\ &= \hat{\mathbf{a}}_{l-1}^\intercal \otimes \left[\left(\hat{\mathbf{a}}_{k-1}^\intercal \otimes I\right)^\intercal \frac{\partial^2 \mathcal{L}}{\partial \mathbf{h}_k{}^2}\frac{\partial \mathbf{h}_k}{\partial \mathbf{h}_l} + \left(\frac{\partial \hat{\mathbf{a}}_{k-1}}{\partial \mathbf{h}_l} \otimes \frac{\partial \mathcal{L}}{\partial \mathbf{h}_k}^\intercal\right)\right] \\ &= \hat{\mathbf{a}}_{l-1}^\intercal \otimes \left[\hat{\mathbf{a}}_{k-1} \otimes \frac{\partial^2 \mathcal{L}}{\partial \mathbf{h}_k{}^2}\frac{\partial \mathbf{h}_k}{\partial \mathbf{h}_l} + \frac{\partial \hat{\mathbf{a}}_{k-1}}{\partial \mathbf{h}_l} \otimes \frac{\partial \mathcal{L}}{\partial \mathbf{h}_k}^\intercal\right] \\ &= \left(\hat{\mathbf{a}}_{k-1}\hat{\mathbf{a}}_{l-1}^\intercal\right) \otimes \frac{\partial^2 \mathcal{L}}{\partial \mathbf{h}_k \partial \mathbf{h}_l} + \hat{\mathbf{a}}_{l-1}^\intercal \otimes \frac{\partial \hat{\mathbf{a}}_{k-1}}{\partial \mathbf{h}_l} \otimes \frac{\partial \mathcal{L}}{\partial \mathbf{h}_k}^\intercal.\end{aligned} \tag{3.18}$$

Combining the above results proves the following lemma

Lemma 3.15. *Using the notations from Section 3.10 and the following definitions:*

- $\tilde{\mathbf{a}}$ *as the concatenation of all layers inputs* — $[\hat{\mathbf{a}}_0^\intercal; \hat{\mathbf{a}}_1^\intercal; \dots; \hat{\mathbf{a}}_{L-1}^\intercal]^\intercal$.
- $\tilde{\mathbf{h}}$ *as the concatenation of all layers pre-activations* — $[\mathbf{h}_1^\intercal; \mathbf{h}_2^\intercal; \dots; \mathbf{h}_L^\intercal]^\intercal$.
- $\mathbf{H}^{FO}$ *as the block diagonal matrix containing the information in the Hessian matrix that depends only on first order information, called The*

Hessian First Order Component. Its k, l block is [a]*:*

$$\mathbf{H}_{k,l}^{FO} = (1 - \delta_l^k) \left(\hat{\mathbf{a}}_{l-1}^\mathsf{T} \otimes \frac{\partial \hat{\mathbf{a}}_{k-1}}{\partial \mathbf{h}_l} \otimes \frac{\partial \mathcal{L}}{\partial \mathbf{h}_k}^\mathsf{T} \right).$$

Then the single sample Hessian matrix of a Feed Forward Neural Network with respect to its parameters θ is equal to:

$$\mathbf{H}_\theta = \left(\tilde{\mathbf{a}} \tilde{\mathbf{a}}^\mathsf{T} \right) * \frac{\partial^2 \mathcal{L}}{\partial \tilde{\mathbf{h}}^2} + \mathbf{H}^{FO}.$$

[a] Here δ_l^k is the indicator function, which evaluates to one only when $k = l$ and is zero otherwise.

3.4.1 The Block Diagonal Hessian Recursion

From the previous derivation, it is easy to notice that the only second-order terms that appear in the expressions of the *sample* Hessian matrix are of the form $\frac{\partial^2 \mathcal{L}}{\partial \mathbf{h}_l{}^2}$. These terms will be referred to as the pre-activation Hessian matrices, and for each layer l will be denoted with $\mathcal{H}_l$. In addition, the matrix of the outer product of the inputs to layers l and k will be denoted as $\mathcal{A}_{l,k}$. Summarising these definitions:

$$\begin{aligned} \mathcal{H}_l &= \frac{\partial^2 \mathcal{L}}{\partial \mathbf{h}_l{}^2}, \\ \mathcal{A}_{l,k} &= \hat{\mathbf{a}}_l \hat{\mathbf{a}}_k^\mathsf{T}. \end{aligned} \tag{3.19}$$

From this the following relationship follows:

$$\begin{aligned} \mathcal{H}_l = \frac{\partial^2 \mathcal{L}}{\partial \mathbf{h}_l{}^2} &= \frac{\partial}{\partial \mathbf{h}_l} \left(\frac{\partial \mathcal{L}}{\partial \mathbf{h}_l} \right) = \frac{\partial}{\partial \mathbf{h}_l} \left(\frac{\partial \mathcal{L}}{\partial \mathbf{h}_{l+1}} \frac{\partial \mathbf{h}_{l+1}}{\partial \mathbf{a}_l} \frac{\partial \mathbf{a}_l}{\partial \mathbf{h}_l} \right) \\ &= \frac{\partial^2 \mathcal{L}}{\partial \mathbf{h}_l \partial \mathbf{h}_{l+1}} \mathbf{W}_{l+1} \frac{\partial \mathbf{a}_l}{\partial \mathbf{h}_l} + \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{l+1}} \frac{\partial \mathbf{a}_l}{\partial \mathbf{h}_l} \frac{\partial}{\partial \mathbf{h}_l} \left(\frac{\partial \mathbf{a}_l}{\partial \mathbf{h}_l} \right) \\ &= \frac{\partial \mathbf{a}_l}{\partial \mathbf{h}_l}^\mathsf{T} \mathbf{W}_{l+1}^\mathsf{T} \mathcal{H}_{l+1} \mathbf{W}_{l+1} \frac{\partial \mathbf{a}_l}{\partial \mathbf{h}_l} + \sum_k \frac{\partial \mathcal{L}}{\partial [\mathbf{a}_l]_k} \frac{\partial^2 [\mathbf{a}_l]_k}{\partial \mathbf{h}_l{}^2}. \end{aligned} \tag{3.20}$$

Importantly, since $[\mathbf{a}_l]_k$ depends only on $[\mathbf{h}_l]_k$ almost all entries in the last term are equal to zero:

$$\begin{aligned}\left[\sum_k \frac{\partial \mathcal{L}}{\partial [\mathbf{a}_l]_k} \frac{\partial^2 [\mathbf{a}_l]_k}{\partial \mathbf{h}_l{}^2}\right]_{ij} &= \sum_k \frac{\partial \mathcal{L}}{\partial [\mathbf{a}_l]_k} \frac{\partial^2 [\mathbf{a}_l]_k}{\partial [\mathbf{h}_l]_i \partial [\mathbf{h}_l]_j} \\ &= \sum_k \frac{\partial \mathcal{L}}{\partial [\mathbf{a}_l]_k} \delta_i^k \delta_j^k \frac{\partial^2 [\mathbf{a}_l]_k}{\partial [\mathbf{h}_l]_i \partial [\mathbf{h}_l]_j} \\ &= \delta_j^i \frac{\partial \mathcal{L}}{\partial [\mathbf{a}_l]_i} \frac{\partial^2 [\mathbf{a}_l]_i}{\partial [\mathbf{h}_l]_i \partial [\mathbf{h}_l]_i}.\end{aligned} \tag{3.21}$$

Combining the previous two results proves the following lemma:

Lemma 3.16. *Given the activation function ϕ, its first and second derivative ϕ' and ϕ'' and defining the diagonal matrices $\mathbf{B}_l$ and $\mathbf{D}_l$ as* [a]*:*

$$\begin{aligned}\mathbf{B}_l &= \frac{\partial \mathbf{a}_l}{\partial \mathbf{h}_l} = \mathbf{diag}\left(\phi'(\mathbf{h}_l)\right), \\ \mathbf{D}_l &= \mathbf{diag}\left(\frac{\partial \mathcal{L}}{\partial \mathbf{a}_l} \circ \phi''(\mathbf{h}_l)\right).\end{aligned} \tag{3.22}$$

Then the pre-activation sample *Hessian for every layer of a Feed Forward Neural Network satisfies the following recursive relationship:*

$$\mathcal{H}_l = \mathbf{B}_l \mathbf{W}_{l+1}^\intercal \mathcal{H}_{l+1} \mathbf{W}_{l+1} \mathbf{B}_l + \mathbf{D}_l. \tag{3.23}$$

[a] Where $\circ$ stands for elementwise product, also called the Hadamard product.

The lemma provides a practical approach for computing all of the diagonal blocks of the *sample* Hessian matrix. It starts by initially computing $\mathcal{H}_L$ for the outputs of the network. This depends on the exact choice of objective function $\mathcal{L}$ and has an analytic expression for standard losses. Then the recursive rule from the lemma is applied in a single backward pass through the network to compute the pre-activation *sample* Hessian for every layer. Finally, according to Lemma 3.15 by performing a Kronecker product of the computed matrices with $\mathcal{A}_{l,l}$ the diagonal blocks of $\mathbf{H}_\theta$ is recovered. It is important to remember, however, that this procedure computes the blocks of the Hessian for a single input x, and it must be repeated for every data point if the goal is to calculate the full Hessian. A similar observation, but restricted to the diagonal entries rather than the more general block-diagonal case, has been previously presented in [120].

3.4.2 The special case of piecewise linear activation functions

In recent years piecewise linear activation functions, such as the Rectified Linear Unit $\phi(x) = \max(x, 0)$, have gained significant popularity among practitioners [50]. Since the second derivative ϕ'' of a piecewise linear function is zero everywhere where it exists, the matrices $\mathbf{D}_l$ from Lemma 3.16 will be zero. Assuming that $\mathcal{H}_L$ is Positive Semi-Definite, that is $\mathbf{v}^\mathsf{T}\mathcal{H}_L\mathbf{v} \geq 0 \quad \forall \mathbf{v}$, then via induction it follows that the pre-activation *sample* Hessian matrices for every layer are PSD as well:

$$\begin{aligned} \mathbf{v}^\mathsf{T}\mathcal{H}_l\mathbf{v} &= \mathbf{v}^\mathsf{T}\mathbf{B}_l\mathbf{W}_{l+1}^\mathsf{T}\mathcal{H}_{l+1}\mathbf{W}_{l+1}\mathbf{B}_l\mathbf{v}, \\ \mathbf{u} &= \mathbf{W}_{l+1}\mathbf{B}_l\mathbf{v}, \\ \mathbf{v}^\mathsf{T}\mathcal{H}_l\mathbf{v} &= \mathbf{u}^\mathsf{T}\mathcal{H}_{l+1}\mathbf{u}, \\ \mathbf{u}^\mathsf{T}\mathcal{H}_{l+1}\mathbf{u} \geq 0 \quad \forall \mathbf{u} \implies \mathbf{v}^\mathsf{T}\mathcal{H}_l\mathbf{v} &\geq 0 \quad \forall \mathbf{v}. \end{aligned} \tag{3.24}$$

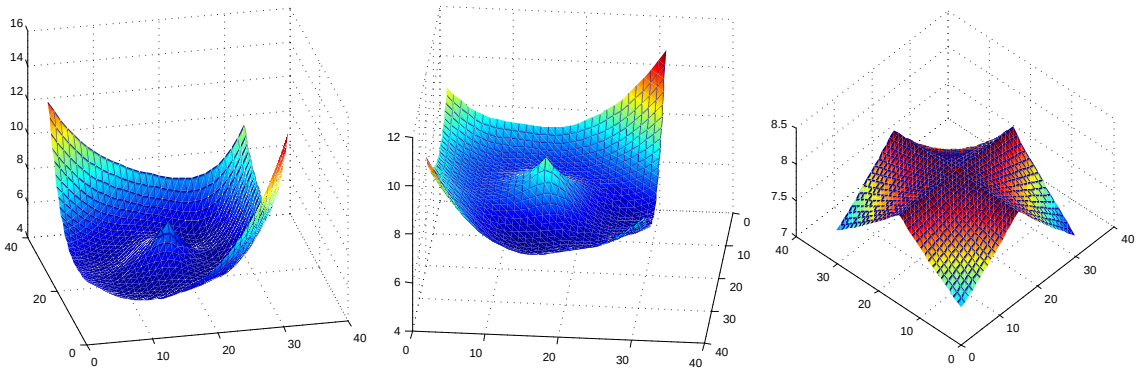
Furthermore, from the fact that all of the matrices $\mathcal{H}_l$ are PSD it also follows that the diagonal blocks of $\mathbf{H}_\theta$ are PSD as well:

$$\begin{aligned} \mathbf{vec}\left(\mathbf{V}\right)^\mathsf{T}\mathbf{H}_{l,l}\mathbf{vec}\left(\mathbf{V}\right) &= \mathbf{vec}\left(\mathbf{V}\right)^\mathsf{T}\left(\mathcal{A}_{l,l} \otimes \mathcal{H}_l\right)\mathbf{vec}\left(\mathbf{V}\right) \\ &= \mathbf{vec}\left(\mathbf{V}\right)^\mathsf{T}\mathbf{vec}\left(\mathcal{H}_l\mathbf{V}\mathcal{A}_{l,l}\right) \\ &= \mathbf{trace}\left(\mathbf{V}^\mathsf{T}\mathcal{H}_l\mathbf{V}\hat{\mathbf{a}}_{l-1}\hat{\mathbf{a}}_{l-1}^\mathsf{T}\right) \\ &= \left(\mathbf{V}\hat{\mathbf{a}}_{l-1}\right)^\mathsf{T}\mathcal{H}_l\left(\mathbf{V}\hat{\mathbf{a}}_{l-1}\right) \geq 0. \end{aligned} \tag{3.25}$$

which is summarised in the following lemma:

Lemma 3.17. *For a Feed Forward Neural Network if the second derivative of its activation functions ϕ is zero everywhere where it exists, and the Hessian of the objective function with respect to the outputs of the network is Positive Semi-Definite then all of the diagonal blocks of the Hessian matrix corresponding to parameters of a single layer $\mathbf{w}_l$ are Positive Semi-Definite.*

A corollary of this results is that if all of the parameters of the network except for $\mathbf{w}_l$ are fixed, the objective function is locally convex with respect to $\mathbf{w}_l$ wherever it is twice differentiable. Note that this does not imply that the objective is convex everywhere with respect to $\mathbf{w}_l$ as the surface will contain ridges along which it is not differentiable, corresponding to boundary points where the activation function changes regimes. In light of this, consider the following lemma:



(a) (b) (c)

Figure 3.1: Loss surface for piecewise linear activation functions. The network has two layers, uses a ReLU activation function and the objective function is a squared error loss. (a) The objective function $\mathcal{E}$ as we vary the first layer weight matrix along two randomly chosen direction matrices $\mathbf{U}$ and $\mathbf{V}$ via $\mathbf{W}_1(x, y) = x\mathbf{U} + y\mathbf{V}$, $(x, y) \in \mathbb{R}^2$. (b) The objective function as instead we vary the weights of the second layer $\mathbf{W}_2$ (c) The objective function as we vary jointly the weights of the first and second layer via $\mathbf{W}_1 = x\mathbf{U}$, $\mathbf{W}_2 = y\mathbf{V}$. The surfaces contain no smooth local maxima. (Best viewed in colour.)

Lemma 3.18. *Under the conditions from Lemma 3.17 the overall objective function $E(\theta, \mathcal{D})$ has no differentiable strict local maxima with respect to the parameters θ of the network.*

Proof. For a point to be a strict and local maxima, it is required that all eigenvalues of $\mathbf{H}_\theta$ are negative whenever it exists. For a symmetric matrix the sum of eigenvalues is equal to the trace of the matrix, so to prove the lemma it is sufficient to show that $\mathbf{trace}\,(\mathbf{H}_\theta) \geq 0$. However, the trace is just the sum of the traces of the diagonal blocks:

$$\mathbf{trace}\,(\mathbf{H}_\theta) = \sum_l \mathbf{trace}\left(\frac{\partial^2 \mathcal{L}}{\partial \mathbf{w}_l{}^2}\right) \geq 0.$$

Where the inequality comes from the result of Lemma 3.17 and the fact that the sum of the eigenvalues of any Positive Semi-Definite matrix is non-negative. □

Hence if there are any local maxima, it must lie on the non-differentiable boundary points of the activation functions. A visual depiction of this fact is shown in Figure 3.1 for a two layer neural network with squared error loss.

3.5 The Generalised Gauss-Newton matrix

As described in Section 2.2.3, when the objective function is non-convex, Newton's method could lead to unbounded updates of the parameters. Trying to address this limitation for the problem of solving non-linear least squares, whose loss function is

$$\mathcal{L} = \frac{1}{2}||\mathbf{h}_\theta(\mathbf{x}) - \mathbf{y}||_2^2, \tag{3.26}$$

the Gauss-Newton matrix was introduced with the following definition:

$$\mathbf{G} = \frac{\partial \mathbf{h}}{\partial \theta}^\intercal \frac{\partial \mathbf{h}}{\partial \theta}. \tag{3.27}$$

There are two different modern views from which this curvature matrix can be motivated [82]. One is to assume that the whole function $\mathbf{h}_\theta(\mathbf{x})$ is linearised locally around the value of the parameters:

$$\hat{\mathbf{h}}_\theta(x) = \mathbf{h}_{\theta_*}(\mathbf{x}) + (\theta - \theta_*)\frac{\partial \mathbf{h}_\theta(\mathbf{x})}{\partial \theta}. \tag{3.28}$$

By plugging $\hat{\mathbf{h}}_\theta$ instead of $\mathbf{h}_\theta$ in the least squares objective it is easy to see that in this case the Gauss-Newton matrix is the Hessian. An alternative perspective is that considering the analytical expression of the Hessian matrix, it is possible to select and discard those terms that are not guaranteed to be Positive Semi-Definite. In the case of the least squares objective $\frac{\partial^2 \mathcal{L}}{\partial \mathbf{h}^2} = \mathbf{I}$ and it follows that

$$\begin{aligned}\frac{\partial^2 \mathcal{L}}{\partial \theta^2} &= \frac{\partial \mathbf{h}}{\partial \theta}^\intercal \frac{\partial^2 \mathcal{L}}{\partial \mathbf{h}^2}\frac{\partial \mathbf{h}}{\partial \theta} + \sum_k \frac{\partial^2 \mathbf{h}_k}{\partial \theta^2}\frac{\partial \mathcal{L}}{\partial \mathbf{h}_k} \\ &= \frac{\partial \mathbf{h}}{\partial \theta}^\intercal \frac{\partial \mathbf{h}}{\partial \theta} + \sum_k \frac{\partial^2 \mathbf{h}_k}{\partial \theta^2}(\mathbf{h}_k - \mathbf{y}_k).\end{aligned} \tag{3.29}$$

Clearly, the second expression depends on the gradient of the loss with respect to $\mathbf{h}$ and in general, it can have arbitrary values. Hence, to construct the Gauss-Newton one discards this expression. Regardless of the chosen view, the quadratic approximation constructed using this curvature matrix has the unique property that it takes the minimal value of zero on an $(n-1)$-dimensional manifold [18]. It has been observed that this definition can be generalised to more than just non-linear least squares [121]. The only requirement is that the loss function $\mathcal{L}$ is convex with respect to $\mathbf{h}_\theta(\mathbf{x})$. In order to define the Gauss-Newton matrix even when this condition is not satisfied, the following definition will be used:

Definition 3.19. *Given a function* $\mathbf{h}_\theta(\mathbf{x})$ *mapping* $\mathbf{x}$ *to* $\mathbf{z}$ *and a loss function* $\mathcal{L}(\mathbf{z}, \mathbf{y})$ *for which there is an associated Positive Semi-Definite curvature matrix* $\mathcal{G}(\mathbf{z}, \mathbf{y})$ *with respect to* $\mathbf{z}$*, then the Generalised Gauss-Newton matrix (GGN) of the loss function* $\mathcal{L}(\mathbf{h}_\theta(\mathbf{x}), \mathbf{y})$ *with respect to the parameters* θ *is*

$$\mathbf{G} = \frac{\partial \mathbf{h}}{\partial \theta}^\mathsf{T} \mathcal{G} \frac{\partial \mathbf{h}}{\partial \theta}.$$

In the special case when $\mathcal{L}$ *is convex with respect to* $\mathbf{z}$ *we can uniquely pick the associated curvature matrix* $\mathcal{G}$ *to be the Hessian* $\frac{\partial^2 \mathcal{L}}{\partial \mathbf{z}^2}$.

The reader might be curious what is so special about the choice $\mathcal{G} = \frac{\partial^2 \mathcal{L}}{\partial \mathbf{h}^2}$ that it is worth mentioning explicitly in the definition. This in fact is the *most* natural choice for the Gauss-Newton matrix with respect to the z. The motivation for this is, that in the first place we would like to use the Hessian matrix, however it is not Positive Semi-Definite. On the other hand, if θ are a local minimizer of $\mathcal{L}$ then indeed the Hessian matrix is Positive Semi-Definite. Comparing the top line of Equation 3.29 and the definition of the Generalised Gauss-Newton, it is easy to see that if $\frac{\partial \mathcal{L}}{\partial \mathbf{h}} = 0$ and $\mathcal{G} = \frac{\partial^2 \mathcal{L}}{\partial \mathbf{h}^2}$ then indeed it follows that $\mathbf{G} = \frac{\partial^2 \mathcal{L}}{\partial \theta^2}$. In order this hold for the full curvature matrix, after taking expectation over the full dataset, this would require the model to have perfectly fit all of the data, that is $\frac{\partial \mathcal{L}}{\partial \mathbf{h}} = 0$ for any inputs. This can still be a motive for this choice, but it is a bit restrictive, as it implies significant overfitting to the data. Instead, we will motivate the choice using the following lemma:

Lemma 3.20. *If the objective function* $\mathcal{L}$ *is a conditional negative log-likelihood function, and there exist parameters* θ_* *under which the model predictive distribution* $p_\theta(\mathbf{y}|\mathbf{h}_\theta(\mathbf{x}))$ *coincides with the true data generating distribution* $p_{\boldsymbol{true}}(\mathbf{y}|\mathbf{x})$*, and for the* sample *Generalised Gauss-Newton, as in Definition 3.19, it has been chosen* $\mathcal{G} = \frac{\partial^2 \mathcal{L}}{\partial \mathbf{h}^2}$*, then the following identity holds:*

$$\mathbb{E}_{p_{\boldsymbol{true}}(\mathbf{x},\mathbf{y})}\left[\mathbf{G}(\theta_*)\right] = \mathbb{E}_{p_{\boldsymbol{true}}(\mathbf{x},\mathbf{y})}\left[\left.\frac{\partial^2 \mathcal{L}}{\partial \theta^2}\right|_{\theta_*}\right].$$

Proof. Examining Equation 3.29 and the definition of the Generalised Gauss-Newton to prove the lemma it is sufficient to show that $\mathbb{E}_{p_{\mathbf{true}}(\mathbf{x},\mathbf{y})}\left[\sum_k \frac{\partial^2 \mathbf{h}_k}{\partial \theta^2} \frac{\partial \mathcal{L}}{\partial \mathbf{h}_k}\right]$. Expanding

this expression, and pushing one of the expectations inside the summation we get:

$$\mathbb{E}_{p_{\mathbf{true}}(\mathbf{x},\mathbf{y})}\left[\sum_k \frac{\partial^2 \mathbf{h}_k}{\partial \theta^2}\frac{\partial \mathcal{L}}{\partial \mathbf{h}_k}\right] = \mathbb{E}_{p_{\mathbf{true}}(\mathbf{x})}\left[\sum_k \frac{\partial^2 \mathbf{h}_k}{\partial \theta^2}\mathbb{E}_{p_{\mathbf{true}}(\mathbf{y}|\mathbf{x})}\left[\frac{\partial \mathcal{L}}{\partial \mathbf{h}_k}\right]\right].$$

However, from the lemma we have assumed that $p_{\mathbf{true}}(\mathbf{y}|\mathbf{x}) = p_\theta(\mathbf{y}|\mathbf{h}_\theta(\mathbf{x}))$ at θ_* and that $\mathcal{L} = -\log p_\theta(\mathbf{y}|\mathbf{h}_\theta(\mathbf{x}))$. Hence the inner expectation becomes:

$$\mathbb{E}_{p_{\mathbf{true}}(\mathbf{y}|\mathbf{x})}\left[\frac{\partial \mathcal{L}}{\partial \mathbf{h}_k}\right] = \mathbb{E}_{p_\theta(\mathbf{y}|\mathbf{h}_{\theta_*}(\mathbf{x}))}\left[\nabla - \log p_\theta(\mathbf{y}|\mathbf{h}_{\theta_*}(\mathbf{x}))\right] = 0,$$

where we used the well known fact that the expectation of the gradient of the log-likelihood is zero. This concludes the proof. □

Hence the curvature matrix for the network output, which would be denoted as $\mathcal{G}_L$, would always be chosen to be equal to $\mathcal{H}_L = \frac{\partial^2 \mathcal{L}}{\partial \mathbf{h}^2}$.

Similarly to the Hessian, the Generalised Gauss-Newton matrix has a Khatri-Rao structure:

Lemma 3.21. *Using the notations from Lemma 3.15 the single* sample *Generalised Gauss-Newton matrix for a Feed Forward Neural Network with respects to its parameters θ is equal to*

$$\mathbf{G}_\theta = \left(\tilde{\mathbf{a}}\tilde{\mathbf{a}}^\intercal\right) * \left(\frac{\partial \mathbf{h}_L}{\partial \tilde{\mathbf{h}}}^\intercal \mathcal{G}_L \frac{\partial \mathbf{h}_L}{\partial \tilde{\mathbf{h}}}\right).$$

Proof. Calculating the k,l block of the GGN starting from the definition:

$$\begin{aligned}
\mathbf{G}_{k,l} &= \frac{\partial \mathbf{h}_L}{\partial \mathbf{w}_k}^\intercal \mathcal{G}_L \frac{\partial \mathbf{h}_L}{\partial \mathbf{w}_l} = \frac{\partial \mathbf{h}_k}{\partial \mathbf{w}_k}^\intercal \frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_k}^\intercal \mathcal{G}_L \frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_l}\frac{\partial \mathbf{h}_l}{\partial \mathbf{w}_l} \\
&= (\hat{\mathbf{a}}_{k-1}^\intercal \otimes I)^\intercal \frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_k}^\intercal \mathcal{G}_L \frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_l}(\hat{\mathbf{a}}_{l-1}^\intercal \otimes \mathbf{I}) \\
&= \left(\hat{\mathbf{a}}_{k-1}\hat{\mathbf{a}}_{l-1}^\intercal\right) \otimes \left(\frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_k}^\intercal \mathcal{G}_L \frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_l}\right).
\end{aligned}$$

□

It can be observed that all of the expressions in the Generalised Gauss-Newton matrix depend on terms of the form $\frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_l}^\intercal \mathcal{G}_L \frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_l}$. In an analogous way to the pre-activation Hessian, these terms will be called the pre-activation Gauss-Newton matrices, and for each layer l will be denoted with $\mathcal{G}_l$.

Lemma 3.22. *Using the notations and definitions from Lemma 3.16 the pre-activation Gauss-Newton for every layer of a Feed Forward Neural Network satisfy the following recursive relationship:*

$$\mathcal{G}_l = \mathbf{B}_l \mathbf{W}_{l+1}^\mathsf{T} \mathcal{G}_{l+1} \mathbf{W}_{l+1} \mathbf{B}_l. \tag{3.30}$$

Proof. Directly from the definition of $\mathcal{G}_l$ and $\mathbf{B}_l$ we can derive

$$\begin{aligned} \mathcal{G}_l &= \frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_l}^\mathsf{T} \mathcal{G}_L \frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_l} = \left(\frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_{l+1}} \frac{\partial \mathbf{h}_{l+1}}{\partial \mathbf{a}_l} \frac{\partial \mathbf{a}_l}{\partial \mathbf{h}_l} \right)^\mathsf{T} \mathcal{G}_L \left(\frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_{l+1}} \frac{\partial \mathbf{h}_{l+1}}{\partial \mathbf{a}_l} \frac{\partial \mathbf{a}_l}{\partial \mathbf{h}_l} \right) \\ &= \mathbf{B}_l \mathbf{W}_{l+1}^\mathsf{T} \left(\frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_{l+1}}^\mathsf{T} \mathcal{G}_L \frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_{l+1}} \right) \mathbf{W}_{l+1} \mathbf{B}_l = \mathbf{B}_l \mathbf{W}_{l+1}^\mathsf{T} \mathcal{G}_{l+1} \mathbf{W}_{l+1} \mathbf{B}_l. \end{aligned} \tag{3.31}$$

□

This result might look strikingly similar to the result from Lemma 3.16, the only noticeable difference is the matrix D_l, which is a diagonal matrix containing the elementwise second-order derivatives of the activation function ϕ. Hence, if ϕ is a piecewise linear function, implying that $D_l = 0$, then two recursions become identical. The following lemma formalises this:

Lemma 3.23. *For a Feed Forward Neural Network if all of the activation functions ϕ have zero second order derivatives whenever they are defined and $\mathcal{G}_L$ has been chosen to be equal to $\mathcal{H}_L$ then for every layer the pre-activation* sample *Gauss-Newton matrix and pre-activation* sample *Hessian matrix are equal. Furthermore the difference between the* sample *Hessian and the* sample *Generalised Gauss-Newton matrix with respect to the parameters θ is the Hessian First Order Component:*

$$\mathbf{H}_\theta = \mathbf{G}_\theta + \mathbf{H}^{FO}.$$

A corollary of this result, since it is valid for any data point, is that the diagonal blocks of the expected Hessian and Generalised Gauss-Newton over the whole data set are also equal, sine the diagonal blocks of the Hessian First Order Component are always zero. This demonstrates how closely the Generalised Gauss-Newton approximates the Hessian even in the case of complicated models like neural networks. Even for non-linear activation functions, if they are behaving almost linearly, it would be expected that this result holds approximately. Although these results apply only to

the matrices for a single data point, they would play an important role in motivating and developing their practical approximations in Chapter 4.

3.5.1 Relationship with the Fisher matrix

Somewhat similar results to the ones for the *sample* Generalised Gauss-Newton matrix has been previously demonstrated for the *sample* Fisher matrix. In most Supervised Learning problems, the two matrices are equal, and hence in those cases, all of the results above apply to both. This section provides a short review of the Fisher matrix and its relationship to the Generalised Gauss-Newton.

For general probability distribution $p_\theta(x)$ the Fisher information matrix of the parameters θ is defined as

$$\mathbf{F} = \mathbb{E}_{p_\theta(\mathbf{x})}\left[\nabla_\theta \log p_\theta(\mathbf{x}) \nabla_\theta \log p_\theta(\mathbf{x})^\mathsf{T}\right]. \tag{3.32}$$

This can equivalently be expressed as the negative of the expectation of the Hessian of the log-likelihood function [82]:

$$\mathbf{F} = -\mathbb{E}_{p_\theta(\mathbf{x})}\left[\nabla^2_\theta \log p_\theta(\mathbf{x})\right]. \tag{3.33}$$

For clarity the proof of this equivalence is given below

$$\begin{aligned}
&\nabla_\theta \mathbb{E}_{p_\theta(\mathbf{x})}\left[\nabla_\theta \log p_\theta(\mathbf{x})\right] = \mathbb{E}_{p_\theta(\mathbf{x})}\left[\nabla_\theta \log p_\theta(\mathbf{x}) \nabla_\theta \log p_\theta(\mathbf{x})^\mathsf{T}\right] + \mathbb{E}_{p_\theta(\mathbf{x})}\left[\nabla^2_\theta \log p_\theta(\mathbf{x})\right], \\
&\mathbb{E}_{p_\theta(\mathbf{x})}\left[\nabla_\theta \log p_\theta(\mathbf{x})\right] = \nabla_\theta \mathbb{E}_{p_\theta(\mathbf{x})}\left[1\right] = \nabla_\theta 1 = 0, \\
&\implies \nabla_\theta \mathbb{E}_{p_\theta(\mathbf{x})}\left[\nabla_\theta \log p_\theta(\mathbf{x})\right] = 0, \\
&\implies \mathbb{E}_{p_\theta(\mathbf{x})}\left[\nabla_\theta \log p_\theta(\mathbf{x}) \nabla_\theta \log p_\theta(\mathbf{x})\right] = -\mathbb{E}_{p_\theta(\mathbf{x})}\left[\nabla^2_\theta \log p_\theta(\mathbf{x})\right].
\end{aligned} \tag{3.34}$$

By construction the Fisher matrix is Positive Semi-Definite— a property that is desired for curvature matrices. Using it in a local quadratic approximation leads to the update $\mathbf{F}^{-1}\nabla_\theta \mathcal{L}$ which is known as Natural Gradient [2]. A different perspective comes from Differential Geometry where the Fisher matrix plays the role of the metric tensor for statistical manifolds — manifolds whose elements are probability distributions. In the Supervised Learning setting, where the model estimates a conditional distribution $p_\theta(\mathbf{y}|\mathbf{x})$ the *sample* Fisher is given by

$$\mathbf{F}_\theta = \mathbb{E}_{p_\theta(\mathbf{y}|\mathbf{x})}\left[\nabla_\theta \log p_\theta(\mathbf{y}|\mathbf{x}) \nabla_\theta \log p_\theta(\mathbf{y}|\mathbf{x})^\mathsf{T}\right], \tag{3.35}$$

and the full Fisher $\overline{\mathbf{F}}_\theta$ is the expectation of the *sample* Fisher over the empirical dataset distribution $p_d(\mathbf{x})$. Notably it does not depend on the true data distribution

of the target variable $\mathbf{y}$. For a Deep Learning model where the output distribution is defined based on the outputs $\mathbf{h}_L$ the Fisher matrix can be expressed as:

$$\begin{aligned}\mathbf{F}_\theta &= \mathbb{E}_{p_\theta(\mathbf{y}|\mathbf{x})}\left[\frac{\partial \mathbf{h}_L}{\partial \theta}^\intercal \frac{\partial \log p_\theta(\mathbf{y}|\mathbf{h}_L)}{\partial \mathbf{h}_L}^\intercal \frac{\partial \log p_\theta(\mathbf{y}|\mathbf{h}_L)}{\partial \mathbf{h}_L}\frac{\partial \mathbf{h}_L}{\partial \theta}\right] \\ &= \frac{\partial \mathbf{h}_L}{\partial \theta}^\intercal \mathbb{E}_{p_\theta(\mathbf{y}|\mathbf{x})}\left[\frac{\partial \log p_\theta(\mathbf{y}|\mathbf{h}_L)}{\partial \mathbf{h}_L}^\intercal \frac{\partial \log p_\theta(\mathbf{y}|\mathbf{h}_L)}{\partial \mathbf{h}_L}\right]\frac{\partial \mathbf{h}_L}{\partial \theta} \\ &= \frac{\partial \mathbf{h}_L}{\partial \theta}^\intercal \mathcal{F}_L \frac{\partial \mathbf{h}_L}{\partial \theta},\end{aligned} \tag{3.36}$$

where the term $\mathcal{F}_l$ is called the pre-activation Fisher matrix for layer l. By inspecting this equation and Definition 3.19 it is clear that the Fisher matrix is equal to the Generalised Gauss-Newton if $\mathcal{G}_L = \mathcal{F}_L$. Since in practice the unique choice $\mathcal{G}_L = \mathcal{H}_L$ is taken a sufficient condition is the objective function to be a negative log-likelihood and its Hessian matrix is independent of $\mathbf{y}$. Although this might appear restrictive, if $\mathbf{h}_L$ parametrises the natural parameters of an exponential family distribution this independence holds [82]. To show this, consider:

$$\log p(\mathbf{y}|\mathbf{x}) = \log h(\mathbf{y}) + T(\mathbf{y})^\intercal \boldsymbol{\eta} - \log Z(\boldsymbol{\eta}), \tag{3.37}$$

where h is the base measure, T is the sufficient statistic, Z is the partition function and η are the natural parameters. Differentiating the negative log-likelihood with respect to the natural parameters twice:

$$\begin{aligned}-\nabla_{\boldsymbol{\eta}} \log p(\mathbf{y}|\mathbf{x}) &= \nabla_{\boldsymbol{\eta}} \log Z(\boldsymbol{\eta}) - T(\mathbf{y}), \\ -\nabla^2_{\boldsymbol{\eta}} \log p(\mathbf{y}|\mathbf{x}) &= \nabla^2_{\boldsymbol{\eta}} \log Z(\boldsymbol{\eta}),\end{aligned} \tag{3.38}$$

which is indeed independent of $\mathbf{y}$. This result is very important as it demonstrates that for most Supervised Learning problems, where the objective is the negative log-likelihood or negative log-posterior, the Generalised Gauss-Newton matrix and the Fisher matrix are equivalent. In this case, all of the results proven in this section apply to the Fisher matrix as well, and this will be something that will link other approaches previously developed in the literature with our own methods.

3.6 On the rank of the empirical matrix

So far, we have mainly focused on analysing the *sample* Generalised Gauss-Newton matrix. However, our final goal is to use these result in order to estimate the expected, or simply referred to as *the* Generalised Gauss-Newton, which has an

expectation over the data distribution. In practice, only a finite data set is provided for training, which can have an important implication for the conditioning of the matrix. First note that the the curvature matrix $\mathcal{G}_L$ is of size $\mathbf{dim}(\mathbf{h}_L) \times \mathbf{dim}(\mathbf{h}_L)$. In any practical cases, however, a Deep Learning model would have thousands, if not million of parameters, and hence $\mathbf{dim}(\theta) \gg \mathbf{dim}(\mathbf{h}_L)$. In this case, after examining Definition 3.19 of the *sample* Generalised Gauss-Newton, it is clear that

$$\mathbf{rank}(\mathbf{G}_\theta) \leq \mathbf{rank}(\mathcal{G}_L) \leq \mathbf{dim}(\mathbf{h}_L). \tag{3.39}$$

Since the full empirical Generalised Gauss-Newton is nothing more than the average of the *sample* matrices over the whole data set, this implies that its rank is upper bounded by $N\mathbf{dim}(\mathbf{h}_L)$, where N is the number of data points. As modern Deep Learning models commonly have millions of parameters, it follows that even the full curvature matrix is usually severely under-determined. This phenomenon is particularly pronounced for the binary classifiers, where the dimensionality of the output of the network is one. This analogously applies to the Fisher matrix as well.

It is possible to draw a parallel between the curvature being zero and standard techniques where the maximum likelihood problem is under-determined for small datasets. This gives a further explanation of why the additional curvature term, coming from the regulariser (e.g. the prior in the case of MAP) is essential. Also, in optimisation it is often common to add a damping term, that is represented by an extra scaled identity matrix, which for the above reasons is crucial in order to make the Generalised Gauss-Newton well behaved.

Chapter 4

Tractable approximations of the Generalized Gauss-Newton matrix

In the previous chapter it was demonstrated that the *sample* Generalised Gauss-Newton matrix very closely resembles the Hessian. In the special case when the activation functions are piecewise linear its diagonal blocks are equal to the diagonal blocks of the Hessian. However, the special structure of the Generalised Gauss-Newton is valid only for a single sample, while when training a neural network the full matrix $\overline{\mathbf{G}_\theta}$ requires computation. In practical applications, where computation time is of importance, it is desirable to have a curvature matrix which is easy to compute, easy to invert, such that they can be used to precondition the gradient, and require memory comparable to the standard computation of a neural network. The first two section develop several approximations which achieve these criteria. The next section describes in details a full optimisation algorithm that makes use of the curvature approximations. The final sections present experimental results and comparison of these methods to other common optimisers in the literature.

The approximation methods presented in this chapter were developed in collaboration with my supervisor Prof. David Barber at University College London. The implementation of all of the curvature approximations in Theano and Lasagne [112, 27] and the experiments presented in the last section of the chapter were performed by me. The final write up of our publication and producing figures and other materials was accomplished in collaboration additionally with my college Hippolyt Ritter.

4.1 Motivating the block diagonal approximation

A significant issue of ever using the full Generalised Gauss-Newton matrix is that it is too large to store explicitly for even a moderate size network — its size is square in the number of parameters. In an attempt to overcome this, rather than trying to approximate the full matrix, the focus will be only on approximating its diagonal blocks. This would allow inversion and curvature-vector products to be computed independently for each layer. Although this discards any dependence across different layers, it still preserves important information about curvature dependencies within the same layer, as well as information about the scale of the objective. This approximation has been previously used in the literature only in the context of the Fisher matrix, and its motivation is purely computational [84, 44, 6]. Below, we present a novel, to our knowledge, result which motivates the block diagonal approximation by showing that in the case of linear networks, it is sufficient to know only the diagonal blocks of the Generalised Gauss-Newton if the goal is to precondition a gradient vector. The full statement is formalised in the following lemma:

Lemma 4.1. *Given Feed Forward Neural Network under the following assumptions*

- *The network is linear, meaning the activation functions ϕ are identity.*
- *The network does not have any biases, hence $\tilde{\mathbf{W}}_l = \mathbf{W}_l$.*
- *All layers have the same width and their weight matrices are invertible.*
- *The loss chosen curvature matrix $\mathcal{G}_L$ is constant.*
- *The matrix $\mathbb{E}_{p_d(\mathbf{x})}\left[\mathbf{x}\mathbf{x}^\top\right]$ is invertible.*

Denote with $\overline{\mathbf{G}}_{diag}$ the matrix constructed by taking only the diagonal blocks of the full Generalised Gauss-Newton matrix $\overline{\mathbf{G}}_\theta$. Given a vector $\mathbf{V}$, which is a gradient of some loss with respect to the network parameters:

$$\mathbf{V} = \mathbb{E}_{p_d(\mathbf{x},\mathbf{y})}\left[\frac{\partial \mathbf{h}_L}{\partial \theta}^\top v(\mathbf{x}, \mathbf{y}, \theta)\right].$$

The the following equality holds

$$\overline{\mathbf{G}}_\theta^{-1}\mathbf{V} = \frac{1}{L}\overline{\mathbf{G}}_{diag}^{-1}\mathbf{V}.$$

Proof. For simplicity of the derivation let:

$$\begin{aligned}
\boldsymbol{\Sigma}_x &= \mathbb{E}_{p_d(\mathbf{x},\mathbf{y})}\left[\mathbf{x}\mathbf{x}^\mathsf{T}\right], \\
\boldsymbol{\Sigma}_y &= \mathbb{E}_{p_d(\mathbf{x},\mathbf{y})}\left[v(\mathbf{x},\mathbf{y},\theta)\mathbf{x}^\mathsf{T}\right], \\
\mathbf{M}_l &= \prod_{1}^{l-1} \mathbf{W}_l, \\
\mathbf{N}_l &= \prod_{l+1}^{L} \mathbf{W}_l.
\end{aligned}$$

Since the network is linear and has no biases we can derive:

$$\begin{aligned}
\mathbf{a}_l &= \left(\prod_{1}^{l-1} \mathbf{W}_l\right)\mathbf{x} = \mathbf{M}_l\mathbf{x}, \\
\overline{\mathcal{A}}_{l,l} &= \mathbb{E}_{p_d(\mathbf{x},\mathbf{y})}\left[\mathbf{M}_l\mathbf{x}\mathbf{x}^\mathsf{T}\mathbf{M}_l^\mathsf{T}\right] = \mathbf{M}_l\boldsymbol{\Sigma}_x\mathbf{M}_l^\mathsf{T}, \\
\frac{\partial \mathbf{h}_L}{\partial \mathbf{w}_l} &= \left(\prod_{i=l+1}^{L} W_i\right)\left(\hat{\mathbf{a}}_l^\mathsf{T} \otimes \mathbf{I}\right) = \left(\mathbf{M}_l\mathbf{x}\right)^\mathsf{T} \otimes \mathbf{N}_l.
\end{aligned}$$

As a result the blocks of the Generalised Gauss-Newton can be expressed as:

$$\begin{aligned}
\overline{\mathbf{G}}_{l,k} &= \mathbb{E}_{p_d(\mathbf{x},\mathbf{y})}\left[\mathcal{A}_{l,k} \otimes \left(\frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_l}^\mathsf{T} \mathcal{G}_L \frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_k}\right)\right] \\
&= \mathbb{E}_{p_d(\mathbf{x},\mathbf{y})}\left[\left(\mathbf{M}_l\mathbf{x}\mathbf{x}^\mathsf{T}\mathbf{M}_k^\mathsf{T}\right) \otimes \left(\mathbf{N}_l^\mathsf{T}\mathcal{G}_L\mathbf{N}_k\right)\right] \\
&= \left(\mathbf{M}_l\boldsymbol{\Sigma}_x\mathbf{M}_k^\mathsf{T}\right) \otimes \left(\mathbf{N}_l^\mathsf{T}\mathcal{G}_L\mathbf{N}_k\right).
\end{aligned}$$

And the l^{th} block of the block vector $\mathbf{V}$:

$$\mathbf{V}_l = \mathbb{E}_{p_d(\mathbf{x},\mathbf{y})}\left[\frac{\partial \mathbf{h}_L}{\partial \mathbf{w}_l}^\mathsf{T} v(\mathbf{x},\mathbf{y},\theta)\right] = \mathbb{E}\left[\mathbf{vec}\left(\mathbf{N}_l^\mathsf{T} v(\mathbf{x},\mathbf{y},\theta)\mathbf{x}^\mathsf{T}\mathbf{M}_l^\mathsf{T}\right)\right] = \mathbf{vec}\left(\mathbf{N}_l^\mathsf{T}\boldsymbol{\Sigma}_y\mathbf{M}_l^\mathsf{T}\right).$$

Let $\mathbf{U} = \overline{\mathbf{G}}_{\text{diag}}^{-1}\mathbf{V}$. Combining the two previous results implies:

$$\begin{aligned}
\mathbf{U}_l = \overline{\mathbf{G}}_{l,l}^{-1}\mathbf{V}_l &= \mathbf{vec}\left(\left(\mathbf{N}_l^\mathsf{T}\mathcal{G}_L\mathbf{N}_l\right)^{-1}\mathbf{N}_l^\mathsf{T}\boldsymbol{\Sigma}_y\mathbf{M}_l^\mathsf{T}\left(\mathbf{M}_l\boldsymbol{\Sigma}_x\mathbf{M}_l^\mathsf{T}\right)^{-\mathsf{T}}\right) \\
&= \mathbf{vec}\left(\mathbf{N}_l^{-1}\mathcal{G}_L^{-1}\boldsymbol{\Sigma}_y\boldsymbol{\Sigma}_x^{-1}\mathbf{M}_l^{-1}\right).
\end{aligned}$$

Finally calculating the l^{th} block of the product of $\overline{\mathbf{G}}$ and $\mathbf{U}$:

$$\begin{aligned}
[\overline{\mathbf{G}}\mathbf{U}]_l &= \sum_k \overline{\mathbf{G}}_{l,k}\mathbf{U}_k = \sum_k \mathbf{vec}\left(\left(\mathbf{N}_l^\mathsf{T}\mathcal{G}_L\mathbf{N}_k\right)\mathbf{N}_k^{-1}\mathcal{G}_L^{-1}\boldsymbol{\Sigma}_y\boldsymbol{\Sigma}_x^{-1}\mathbf{M}_k^{-1}\left(\mathbf{M}_l\boldsymbol{\Sigma}_x\mathbf{M}_k^\mathsf{T}\right)^\mathsf{T}\right) \\
&= \sum_k \mathbf{vec}\left(\mathbf{N}_l^\mathsf{T}\boldsymbol{\Sigma}_y\mathbf{M}_l^\mathsf{T}\right) = \sum_k \mathbf{V}_l = L\mathbf{V}_l.
\end{aligned}$$

□

The next step to a practical approximation is to consider how to approximate every diagonal block $\overline{\mathbf{G}}_{l,l}$.

4.2 Approximating the Diagonal Blocks of $\overline{\mathbf{G}}$

Recall that from Lemma 3.21 the diagonal blocks of the *sample* Generalised Gauss-Newton matrix can be expressed as

$$\mathbf{G}_{l,l} = \mathcal{A}_{l,l} \otimes \mathcal{G}_l. \tag{4.1}$$

The diagonal blocks of the Generalised Gauss-Newton $\overline{\mathbf{G}}_{l,l}$ are then given as the expectation of the above equation over the data distribution $p_d(\mathbf{x}, \mathbf{y})$. To compute this, it would be required to carry out the computation for each single data point and then add all of the results. Since the expectation of a Kronecker product is not necessarily Kronecker factored, there is no compact representation for this matrix, and it will be required to be stored it explicitly. If the dimensionality of the inputs and outputs of a layer are both of order $O(d)$ then such matrix would have $O(d^4)$ elements. Even for $d = 1000$, which is not large by standard Deep Learning architectures, the memory required to store that many elements in a 32bit floating point precision is on the order of a few terabytes. As this is prohibitively large even on modern hardware, instead of estimating the matrix exactly, an approximation that is efficient both to store and compute is needed. The approach we propose is to do a factorised expectation approximation, also used in [84]:

$$\overline{\mathbf{G}}_{l,l} \approx \overline{\mathcal{A}}_{l,l} \otimes \overline{\mathcal{G}}_l. \tag{4.2}$$

In addition, this approximation is also useful when inverting the curvature matrix because of Property 3.8. Furthermore, under the conditions from Lemma 4.1 the above equation is exact, since $\mathcal{G}_l$ does not depend on the data and the expectation is taken only over $\mathcal{A}_{l,l}$.

The first factor $\overline{\mathcal{A}}_{l,l}$ is simply the uncentered covariance of the inputs to the layer:

$$\overline{\mathcal{A}}_{l,l} = \frac{1}{N} \sum_n \hat{\mathbf{a}}_l \hat{\mathbf{a}}_l^\intercal. \tag{4.3}$$

By stacking all of the vectors $\hat{\mathbf{a}}_l^n$ into a $d \times n$ matrix M this can be easily computed via the single product $MM^\intercal$. Such a matrix is usually already computed on the forward evaluation pass of the network.

For the second factor, however, there do not exist in general efficient ways of computing it. In the next section, several strategies for dealing with this are presented.

4.3 Practical calculations for $\overline{\mathcal{G}}_l$

Recall from Lemma 3.22 that the pre-activation *sample* Generalised Gauss-Newton matrix follows the recursive relationship:

$$\mathcal{G}_l = \mathbf{B}_l \mathbf{W}_l^\mathsf{T} \mathcal{G}_{l+1} \mathbf{W}_l \mathbf{B}_l. \tag{4.4}$$

However, for the curvature matrix of the objective, it is required to compute the expected value — $\mathbb{E}_{p_d(\mathbf{x},\mathbf{y})}[\mathcal{G}_l]$. In principle, the recursion can be applied independently for each single data point, but this would require storing the matrix $\mathcal{G}_l$ for every data point. This is impractical in terms of computation time and could be useful only in cases where time is not essential. In optimisation, however, computation time together with a decrease in the loss value are of the highest importance to practitioners. As a result, below, we show several strategies for performing this computation more efficiently.

4.3.1 Exact low rank calculation

Many problems in classification and regression deal with a relatively small number of outputs. This implies that the rank of the output layer *sample* Gauss-Newton matrix $\mathcal{G}_L$ is low. From the recursive relationship from Lemma 3.22 it follows that for any layer the pre-activation *sample* Gauss-Newton matrices have at most the rank of $\mathcal{G}_L$. Given that each one is Positive Semi-Definite it can be represented as the sum of its square root vectors:

$$\mathcal{G}_l = \sum_{k=1}^{K} \mathbf{c}_l^k \mathbf{c}_l^{k\mathsf{T}}. \tag{4.5}$$

From Lemma 3.22 the following recursion is obtained:

$$\mathbf{c}_l^k = \frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_l}^\mathsf{T} \mathbf{c}_L^k = \mathbf{B}^l \mathbf{W}_{l+1}^\mathsf{T} \mathbf{c}_{l+1}^k. \tag{4.6}$$

This is the same type of computation performed during back-propagation, but using $\mathbf{c}_L^k$ instead of the gradient vector $\frac{\partial \mathcal{L}}{\partial \mathbf{h}_L}$. Hence for a fixed index k computing the vectors $\mathbf{c}_l^k$ for all layers has the cost of a single backward pass. Similar to Equation 4.3 the vectors $\mathbf{c}_l^k$ can be computed efficiently for a whole batch of data points, after which they can be stacked into a matrix, and $\mathcal{G}_l$ can be computed via single matrix-matrix multiplication. For this approach, it is needed to store an array of size $N \times d \times K$ rather than $N \times d \times d$ in the naive case. For small K this is significantly more

efficient, both in terms of memory and computation. This method was originally named Kronecker Factored Low Rank (KFLR) [10] and is closely related to the curvature propagation in [86].

4.3.2 Recursive Mean Propagation

In practical cases where the dimensions of the outputs of the network are very high, for instance, autoencoders, using the exact computation from the previous section, becomes a computational bottleneck. Instead of exact computation, we propose to use the recursion from Lemma 3.22, but pass back only the expectation of the matrix. This yields the nested expectation approximation below:

$$\hat{\mathcal{G}}_l \approx \mathbb{E}_{p_d(\mathbf{x},\mathbf{y})}\left[\mathbf{B}_l \mathbf{W}_{l+1}^\mathsf{T} \hat{\mathcal{G}}_{l+1} \mathbf{W}_{l+1} \mathbf{B}_l\right]. \tag{4.7}$$

The recursion is initialised with the exact value of the last layer Gauss-Newton $\overline{\mathcal{G}}_L$, which can be easily computed. Hence, this approach is computationally feasible, even for deep neural networks. Moreover, notice that under the assumptions of Lemma 4.1 the matrices $\mathcal{G}_l$ do not depend on $\mathbf{x}$ and $\mathbf{y}$ and the computation is exact, not just an approximation. This method has been named initially Kronecker Factored Recursive Approximation (KFRA) [10].

4.3.3 Using The Fisher identity

In special cases where the Generalised Gauss-Newton matrix is equivalent to the Fisher matrix, as described in Section 3.5.1, it is possible to use this fact in order to approximate $\mathcal{G}_l$. Specifically, note that the last layer Gauss-Newton satisfies the Fisher identity:

$$\mathcal{G}_L = \mathcal{F}_L = \mathbb{E}_{p_\theta(\mathbf{y}|\mathbf{x})}\left[\nabla_{\mathbf{h}_L} \log p_\theta(\mathbf{y}|\mathbf{h}_L) \nabla_{\mathbf{h}_L} \log p_\theta(\mathbf{y}|\mathbf{h}_L)^\mathsf{T}\right]. \tag{4.8}$$

Instead of computing this exactly it can be approximated via Monte Carlo sampling:

$$\begin{aligned} \mathcal{G}_L &= \frac{1}{K}\sum_k \nabla_{\mathbf{h}_L} \log p_\theta(\mathbf{y}_k|\mathbf{h}_L) \nabla_{\mathbf{h}_L} \log p_\theta(\mathbf{y}_k|\mathbf{h}_L)^\mathsf{T}, \\ \mathbf{y}_k &\sim p_\theta(\mathbf{y}|\mathbf{h}_L). \end{aligned} \tag{4.9}$$

Computationally this now is equivalent to KFLR, by setting the square root vectors to $\mathbf{c}_L^k = \frac{1}{\sqrt{K}} \nabla_{\mathbf{h}_L} \log p_\theta(\mathbf{y}_k|\mathbf{h}_L)$. The main computational benefit, however, is that it is possible to estimate $\overline{\mathcal{G}_l}$ using a single sample for each data point, allowing this

method to have an overhead of just a single backward pass. The main drawback is that it can never be exact and potentially introduces additional noise in the curvature matrix estimate. Using the block diagonal Kronecker factored approximation together with the Fisher identity to obtain a stochastic approximation, in the context of the Fisher matrix, is exactly equivalent to a method called KFAC and has been developed independently in [84].

4.3.4 Using Random Projections

The previous section suggests an even more general approach for stochastically estimating $\mathcal{G}_l$. Consider the following identity for any Positive Semi-Definite matrix of rank K, assuming that $\mathbb{E}_{p(\mathbf{v})}\left[\mathbf{v}\mathbf{v}^\mathsf{T}\right] = \mathbf{I}$:

$$\mathbf{M} = \mathbf{L}\mathbf{L}^\mathsf{T} = \mathbb{E}_{p(\mathbf{v})}\left[\mathbf{L}\mathbf{v}\mathbf{v}^\mathsf{T}\mathbf{L}^\mathsf{T}\right] = \mathbb{E}_{p(\mathbf{v})}\left[(\mathbf{L}\mathbf{v})(\mathbf{L}\mathbf{v})^\mathsf{T}\right]. \tag{4.10}$$

Hence, by choosing any multivariate distribution $p(\mathbf{v})$ such that its second moment is the identity matrix, one can use Monte Carlo sampling to approximate $\mathcal{G}_L$ via:

$$\begin{aligned} \mathcal{G}_L &= \frac{1}{K}\sum_k (\mathbf{L}\mathbf{v}_k)(\mathbf{L}\mathbf{v}_k)^\mathsf{T}, \\ \mathbf{v}_k &\sim p(\mathbf{v}), \end{aligned} \tag{4.11}$$

where $\mathbf{L}$ is the Cholesky factorisation of $\mathcal{G}_L$. The computational benefits and drawbacks are analogous to those of the Fisher identity method and can similarly use only a single sample to calculate the approximation, making it also computationally viable.

4.4 The full optimization algorithm

So far, we have described how to approximate the Generalised Gauss-Newton matrix. However, the full second-order optimisation algorithm that will be used contains many additional details, which are described in this section. Most of the steps have been previously used in the context of optimisation in [81].

Recall that we will assume that the objective can be written in the following form:

$$\mathcal{E}(\theta, \mathcal{D}) = \frac{1}{N}\sum_n \mathcal{L}^n + \frac{\lambda}{2}||\theta||_2^2. \tag{4.12}$$

This implies that full curvature matrix is

$$\mathbf{C}(\theta, \mathcal{D}) = \overline{\mathbf{G}}_\theta + \lambda \mathbf{I}. \tag{4.13}$$

Hence, the quadratic approximation can be expressed as

$$\mathcal{E}(\theta_t + \delta, \mathcal{D}) \approx q(\delta) = \mathcal{E}(\theta_t, \mathcal{D}) + \delta^\mathsf{T} \nabla_{\theta_t} E(\theta_t, \mathcal{D}) + \frac{1}{2} \delta^\mathsf{T} \mathbf{C}(\theta_t, \mathcal{D}) \delta. \tag{4.14}$$

Dropping the dependence on θ_t and $\mathcal{D}$ for clarity, the solution to this problem is clearly

$$\delta_* = -\mathbf{C}^{-1} \nabla_{\theta_t} \mathcal{E}. \tag{4.15}$$

However, as already discussed, it is not feasible to store the curvature matrix and hence its exact inversion is not possible. Hessian free methods attempt to do this using a truncated Conjugate Gradient, but unfortunately, this is still too computationally heavy for a practical approach [81]. Instead, given one of the block-diagonal approximations introduced in this chapter — KFAC, KFLR, KFRA — denoted as $\widetilde{\mathbf{G}}_\theta$, the algorithm proposes a direction

$$\widetilde{\delta} = \left(\widetilde{\mathbf{G}}_\theta + \lambda \mathbf{I} \right)^{-1} \nabla_{\theta_t} \mathcal{E}. \tag{4.16}$$

This vector, however, is not the final update applied to the parameters θ_t. The reason for this is that due to potential approximation error it could potentially be suboptimal for $q(\delta)$. Instead, a line search is performed along the direction $\widetilde{\delta}$ for selecting the optimal step size. Let the step size be α, then

$$q\left(\alpha \widetilde{\delta}\right) = \mathcal{E} + \alpha \widetilde{\delta}^\mathsf{T} \nabla_{\theta_t} \mathcal{E} + \frac{\alpha^2}{2} \widetilde{\delta}^\mathsf{T} \mathbf{C} \widetilde{\delta}. \tag{4.17}$$

Since this is a quadratic function in α, minimising it has an analytical solution

$$\alpha_* = -\frac{\widetilde{\delta}^\mathsf{T} \nabla_{\theta_t} \mathcal{E}}{\widetilde{\delta}^\mathsf{T} \mathbf{C}. \widetilde{\delta}}. \tag{4.18}$$

Computing this value might seem a difficult task as the denominator term contains product with the full curvature matrix. However, using automatic differentiation and the R-operator [121] it is possible to calculate matrix-vector products of the full Generalised Gauss-Newton efficiently and in extension with $\mathbf{C}$ [81, 84], and here only one such product is required. The cost of this operation is on the order of one forward and one backward pass through the network. After computing the optimal step size, the algorithm finally applies the update:

$$\theta_{t+1} = \theta_t + \alpha_* \widetilde{\delta}. \tag{4.19}$$

4.4.1 The role of damping

The idea of using damping on the curvature matrix in second-order methods is not novel. One of the first methods to use it is was the Levenberg-Marquardt algorithm which applied it for the classical Gauss-Newton matrix for non-linear least squares. It can improve the numerical stability of the inversion of the matrix and can also be related to trust-region methods. The goal is to introduce an additional diagonal term to the curvature matrix with a magnitude such that local quadratic approximation $q(\delta)$ is accurately estimating the actual value of the function at the minimum of q. Hence in practice, the curvature matrix used in q is actually

$$\mathbf{C} = \overline{\mathbf{G}}_\theta + (\lambda + \tau)\mathbf{I}. \tag{4.20}$$

The value of τ is initialised to some reasonably large quantity, such that it can prevent any large initial steps. Thereafter it is adapted using the Levenberg-Marquardt heuristic based on the reduction ratio ρ defined as

$$\rho = \frac{\mathcal{E}(\theta + \alpha_* \widetilde{\delta}) - \mathcal{E}(\theta)}{q(\alpha_* \widetilde{\delta}) - q(0)}. \tag{4.21}$$

When $\rho < 1$ this means that the true function $\mathcal{E}$ has a lower value at $\theta + \alpha_* \widetilde{\delta}$ and thus the quadratic approximation underestimates the curvature. In the opposite case when $\rho > 1$ the current quadratic approximation overestimates the curvature. The Levenberg-Marquardt method introduces the damping decay constant $\omega_\tau < 1$ and at every iteration it does one of the following two options:

- If $\rho > 0.75$ then $\rho_{t+1} = \omega_\tau \rho_t$.
- If $\rho < 0.75$ then $\rho_{t+1} = \omega_\tau^{-1} \rho_t$.

In the case of a Deep Learning model, however, this requires the evaluation of $\mathcal{E}(\theta + \alpha_* \widetilde{\delta})$ on the same minibatch. In order to not introduce a significant computational overhead of one extra network evaluation, this adaptation is instead applied every T_τ iterations. In all of our experiments we used $\omega_\tau = 0.95$ and $T_\tau = 5$.

Similarly, to the parameter τ, another damping parameter ζ is introduced. Its primary purpose, however, is to regularise the approximate Gauss-Newton matrix that is used for computing $\widetilde{\delta}$, such that it approximates as best as possible the full Generalised Gauss-Newton. The main reason for having two parameters is because we have two levels of approximations, and each parameter independently affects them:

$$\mathcal{E}(\theta + \delta) \overset{\tau}{\approx} q(\delta; \overline{\mathbf{G}}_\theta) \overset{\zeta}{\approx} q(\delta; \widetilde{\mathbf{G}}_\theta). \tag{4.22}$$

The parameter ζ is updated greedily using a decay constant $\omega_\zeta < 1$. Every T_ζ iterations the algorithm computes three updates δ_* for three different values of ζ - $\{\omega_\zeta, \zeta, \omega_\zeta^{-1}\zeta\}$. From these three updates, the one that minimises $q(\alpha_*\widetilde{\delta})$ is selected, and ζ is updated accordingly. Similarly to τ in all experiments we set $\omega_\zeta = 0.95$ and $T_\zeta = 20$.

4.4.2 Inverting the approximate curvature matrix

The approximate curvature matrix used in the algorithm is

$$\widetilde{\mathbf{C}} = \widetilde{\mathbf{G}}_\theta + (\lambda + \zeta)\mathbf{I}. \tag{4.23}$$

The approximation that is used for $\widetilde{\mathbf{G}}_\theta$ in all methods considered is firstly block diagonal, and then each block is a Kronecker product. From the properties of the Kronecker product presented in the previous chapter, it is known that the inverse is just the Kronecker product of the inverses of the factors:

$$(\mathbf{A} \otimes \mathbf{B})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{B}^{-1}. \tag{4.24}$$

However, because of the diagonal term added, no such identity holds. The exact calculation for computing the inverse of a Kronecker product plus scaled identity requires to compute the eigenvalue decomposition of both matrices $\mathbf{A}$ and $\mathbf{B}$ [82]. Since the main goal is to find a fast to compute approximation we propose the following approximation:

$$(\mathbf{A} \otimes \mathbf{B} + \lambda\mathbf{I})^{-1} \approx \left(\mathbf{A} + \omega\sqrt{\lambda}\mathbf{I}\right)^{-1} \otimes \left(\mathbf{B} + \omega^{-1}\sqrt{\lambda}\mathbf{I}\right)^{-1}. \tag{4.25}$$

In order to choose the value of ω we can bound the norm of the residual:

$$\begin{aligned} R(\omega) &= \mathbf{A} \otimes \mathbf{B} + \lambda\mathbf{I} - \left(\mathbf{A} + \omega\sqrt{\lambda}\mathbf{I}\right) \otimes \left(\mathbf{B} + \omega^{-1}\sqrt{\lambda}\mathbf{I}\right) \\ &= \omega^{-1}\sqrt{\lambda}\mathbf{A} \otimes \mathbf{I} + \omega\sqrt{\lambda}\mathbf{I} \otimes \mathbf{B}, \\ ||R(\omega)|| &\leq \sqrt{\lambda}\left(\omega^{-1}||\mathbf{A} \otimes \mathbf{I}|| + \omega||\mathbf{I} \otimes \mathbf{B}||\right). \end{aligned} \tag{4.26}$$

Minimising the right hand side with respect to ω gives the solution

$$\omega_* = \sqrt{\frac{||\mathbf{A} \otimes \mathbf{I}||}{||\mathbf{I} \otimes \mathbf{B}||}}. \tag{4.27}$$

The choice of the norm is arbitrary, but for comparative purposes with previous work on KFAC, in all of our experiments, we use the trace norm. An alternative perspective on this approximation is that it is a particular form of damping, tailored explicitly to Kronecker factored matrices. The full algorithm, specifically for the KFRA approximation, is presented below:

Algorithm 1 Algorithm for KFRA updates excluding updates for τ and ζ

Input: minibatch X, weight matrices $\mathbf{W}_{1:L}$ and biases $\mathbf{b}_{1:L}$, activation functions $\phi_{1:L}$, true outputs Y, parameters η and γ

- Forward Pass -

$\mathbf{a}_0 = X$

for $l = 1$ **to** L **do**

$\mathbf{h}_l = \mathbf{W}_l \mathbf{a}_{l-1} + \mathbf{b}_l$

$\mathbf{a}_l = \phi_l(\mathbf{h}_l)$

end for

- Derivative and Hessian of the objective -

$d_L = \frac{\partial \mathcal{E}}{\partial \mathbf{h}_L}\Big|_{\mathbf{h}_L}$

$\widetilde{\mathcal{G}_L} = \mathbb{E}_{p_d(\mathbf{x},\mathbf{y})}\left[\mathcal{H}_L\right]\Big|_{\mathbf{h}_L}$

- Backward pass -

for $l = L$ **to** 1 **do**

- Update for $\mathbf{W}_l$ *-*

$g_l = \frac{1}{N} d_l \hat{\mathbf{a}}_{l-1}^\intercal + \lambda \hat{\mathbf{W}}_l$

$\widetilde{A}_l = \frac{1}{N} \hat{\mathbf{a}}_{l-1} \hat{\mathbf{a}}_{l-1}^\intercal$

$\omega = \sqrt{\frac{\mathbf{trace}\left(\widetilde{A}_l\right) \dim \widetilde{\mathcal{G}}_l}{\mathbf{trace}\left(\widetilde{\mathcal{G}}_l\right) \dim \widetilde{A}_l}}$

$k = \sqrt{\lambda + \zeta}$

$\widetilde{\delta}_l = (\widetilde{A}_l + \omega k)^{-1} g_l (\widetilde{\mathcal{G}}_l + \omega^{-1} k)^{-1}$

if $l > 1$ **then**

- Propagate gradient and approximate pre-activation Gauss-Newton -

$B_{l-1} = \phi'(\mathbf{h}_{l-1})$

$d_{l-1} = \mathbf{W}_l^\intercal d_l \odot B_{l-1}$

$\widetilde{\mathcal{G}}_{l-1} = (\mathbf{W}_l^\intercal \widetilde{\mathcal{G}}_l \mathbf{W}_l) \odot \left(\frac{1}{N} A_{l-1} A_{l-1}^\intercal\right)$

end if

end for

$v = \frac{\partial \mathbf{h}_L}{\partial \theta} \widetilde{\delta}$ (using the R-op from [121])

$\widetilde{\delta}^\intercal \overline{\mathbf{G}} \widetilde{\delta} = v^\intercal \mathcal{H}_L v$

$\widetilde{\delta}^\intercal \overline{C} \widetilde{\delta} = \widetilde{\delta}^\intercal \overline{\mathbf{G}} \widetilde{\delta} + (\lambda + \tau) ||\widetilde{\delta}||_2^2$

$\alpha_* = -\frac{\widetilde{\delta}^\intercal \nabla \mathcal{E}}{\widetilde{\delta}^\intercal \overline{C} \widetilde{\delta}}$

for $l = 1$ **to** L **do**

$\hat{W}_l = \hat{W}_l + \alpha_* \delta_l$

end for

4.5 Experiments

In order to evaluate the optimisation algorithm using the approximate Generalised Gauss-Newton matrix, we performed several experiments and analysed their results. In the first series of experiments, we trained a deep autoencoder on three standard grey-scale image datasets. In addition, we consider a small toy problem to classify hand-written digits as odd or even using a mixture classifier. The main goal of this experiment is to evaluate a model where the Gauss-Newton is not equivalent to the Fisher as the modelling distribution is not in the exponential family. The datasets being used are:

MNIST consists of $60,000$ 28×28 images of hand-written digits [69]. As common practice, only the first $50,000$ images are used for training, as the other $10,000$ are usually used for validation or testing.

CURVES contains $20,000$ training images of size 28×28 pixels of simulated hand-drawn curves, created by choosing three random points in the 28×28 pixel plane. For more details on this dataset, the reader is referred to the supplementary material of the original publication [54].

FACES is an augmented version of the Olivetti faces dataset with 10 different images of 40 people [119]. For comparative purposes with previous work, we create a training set of $103,500$ images by choosing 414 random pairs of rotation angles (-90 to 90 degrees) and scaling factors (1.4 to 1.8) for each of the 250 images for the first 25 people and then subsampling to 25×25 pixels as in [54].

All of the experiments were executed on a computer workstation with a Titan Xp GPU and an Intel Xeon CPU E5-2620 v4 @ 2.10GHz.

The performance of second-order methods compared against first-order methods was tested as well as the quality of the different approximations to the Generalised Gauss-Newton discussed earlier in the chapter. The main goal of an optimiser is to minimise the training loss; hence we do not try to compare any generalisation of the resulting models. To this end, in all experiments, we report only the training error. The second-order algorithm is following Section 4.4, where in addition to the proposed method KFRA, we also compare against KFAC [84], which always approximates the Fisher matrix as discussed in Section 4.3.3. For any stochastic approximation methods such as KFAC, only a single Monte Carlo sample is taken.

We emphasise that throughout all experiments, we used the default damping parameter settings, with no tweaking required to obtain acceptable performance. One can compare this hyperparameter to the exponential decay parameters β_1 and β_2 in Adam, which are typically left at their recommended default values.

Additionally, as a form of momentum for the curvature estimation, we compared the use of an exponential moving average with a factor of 0.9 on the curvature matrices $\mathcal{G}_l$ and $\mathcal{A}_{l,l}$ to only estimating them from the current minibatch. We did not find any benefit in using momentum on the updates themselves; on the contrary, this made the optimisation unstable and required clipping the updates. We, therefore, do not include momentum on the updates in our results.

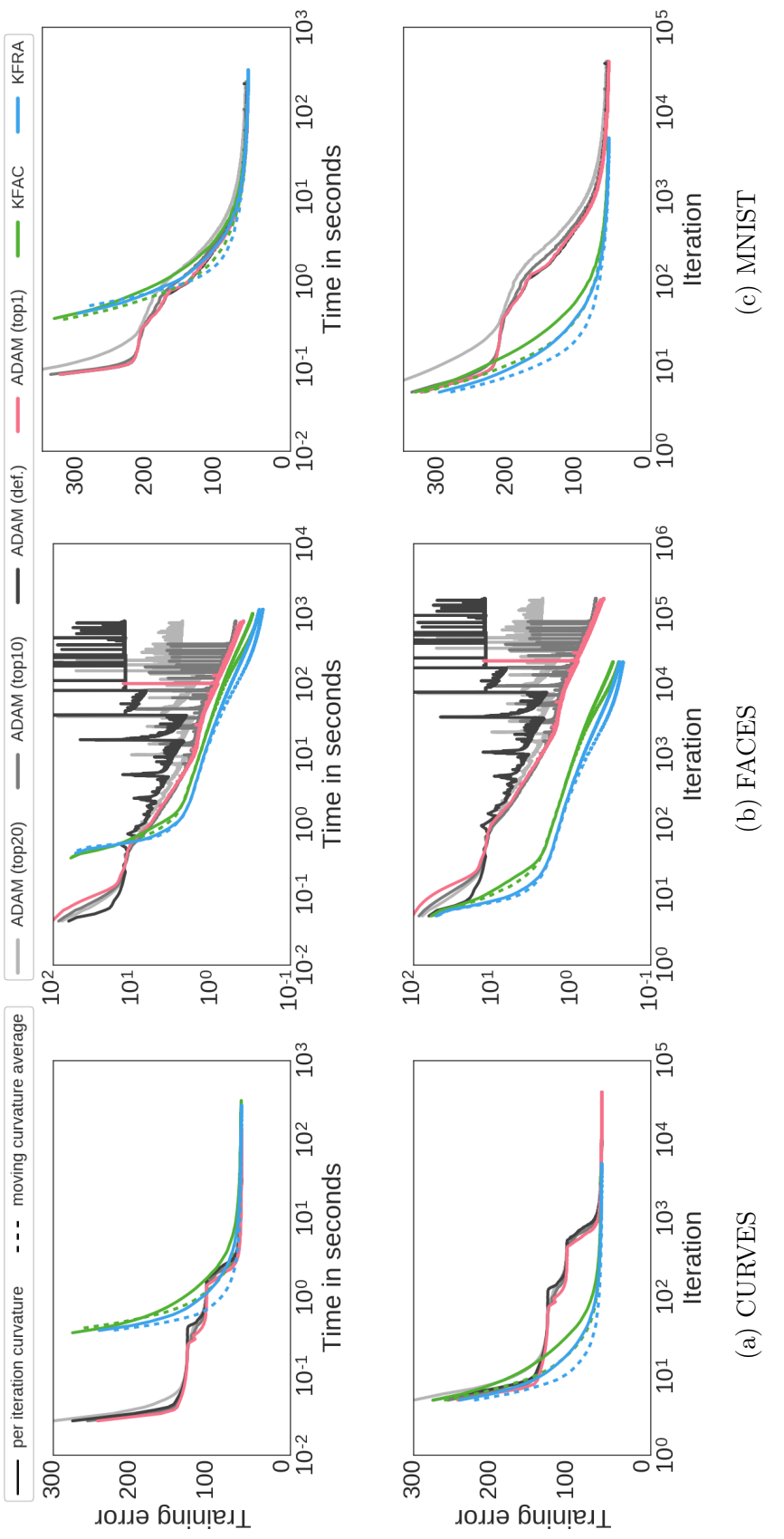
All of the autoencoder architectures are inspired by previous work in [54]. The layer sizes are D-1000-500-250-30-250-500-1000-D, where D is the dimensionality of the input. All models have been initialised identically across different optimisers using standard practices for sampling the weights [40]. The grey-scale values are interpreted as the mean parameter of a Bernoulli distribution, and the loss is the binary cross-entropy on CURVES and MNIST, and square error on FACES.

4.5.1 Comparison to First-Order Methods

We investigated the performance of both KFRA and KFAC compared to popular first-order methods. Four of the most prevalent gradient-based optimisers were considered — Stochastic Gradient Descent with and without Momentum, Nesterov Accelerated Gradient and ADAM [61]. A common practice when using first-order methods is to decrease the learning rate throughout the training procedure. With the goal to perform a fair comparison, we included an extra parameter T — the decay period — to each of the methods, halving the learning rate every T iterations. To find the best first-order method, we ran a grid search over these two hyperparameters, by varying the learning rate from 2^{-6} to 2^{-13} at every power of 2 and chose the decay period value from $\{100\%, 50\%, 25\%, 12.5\%, 6.25\%\}$.

Each first-order method was run for $40,000$ parameter updates for MNIST and CURVES and $160,000$ updates for FACES. This resulted in a total of 35 experiments and 1.4/5.6 million updates for each dataset per method. In contrast, the second-order methods did not require adjustment of any hyperparameters and were run for only $5,000/20,000$ updates, as they converged much faster. All of the methods were implemented using Theano and Lasagne [112, 27]. For the first-order methods, we found ADAM to outperform the others across the board, and we consequently

compared the second-order methods against ADAM only.



(a) CURVES (b) FACES (c) MNIST

Figure 4.1: Comparison of the objective function being optimised by KFRA, KFAC and ADAM on CURVES, FACES and MNIST. GPU benchmarks are in the first row, progress per update in the second. The dashed line indicates the use of momentum on the curvature matrix for the second-order methods. Errors are averaged using a sliding window of ten.

Figure 4.1 shows the performance of the different optimisers on all three datasets. We present progress both per parameter update, to demonstrate that the second-order optimisers effectively use the available curvature information, and per GPU wall clock time, as this is relevant when training a network in practice. For ADAM, we display the performance using the default learning rate 10^{-3} as well as the top-performing combination of learning rate and decay period. To illustrate the sensitivity of ADAM to these hyperparameter settings in these challenging tasks, and how much can, therefore, be gained by parameter tuning, we also plot the average performance resulting from using the top 10 and top 20 settings.

Even after significantly tuning the ADAM learning rate and decay period, the second-order optimisers outperformed ADAM out-of-the-box across all datasets. In particular, on the challenging FACES dataset, the optimisation was not only much faster when using second-order methods, but also more stable. On this dataset, ADAM appears to be highly sensitive to the learning rate and diverged when running with the default learning rate of 10^{-3}. In contrast to ADAM, the second-order optimisers did not get trapped in plateaus in which the error does not change significantly.

In comparison to KFAC, KFRA showed a noticeable speed-up in the optimisation both per-iteration and when measuring the wall clock time. To investigate the potential advantages of KFRA over KFAC and whether it stems from the quality of its updates, in the next section, we compared the alignment of the updates of each method with the exact Gauss-Newton update.

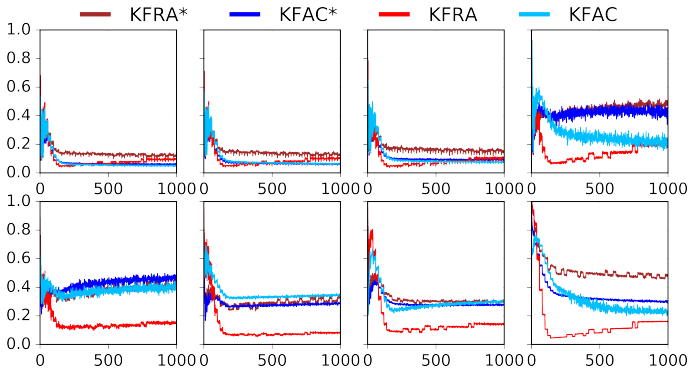
4.5.2 Alignment of the Approximate Updates

To gain insight into the quality of the approximations that are made in the second-order methods under consideration, we compare how well the KFAC and KFRA parameter updates $\widetilde{\delta}$ are aligned with updates obtained from using the exact block diagonal Gauss-Newton and the full Gauss-Newton matrix. Additionally, we check how using the approximate inversion of the Kronecker factored curvature matrices discussed in Section 4.4.2 impacts the alignment.

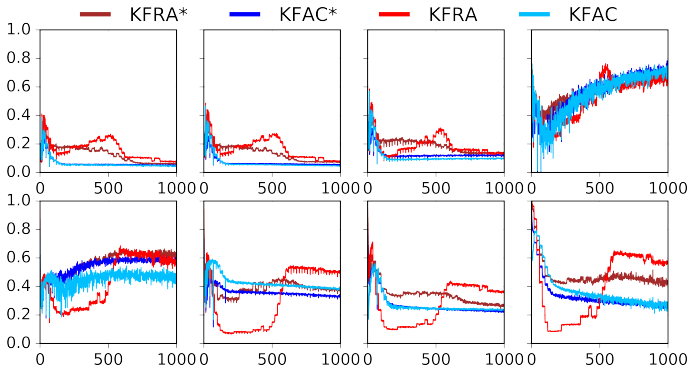
In order to find the updates for the full Gauss-Newton method we use the Conjugate Gradient and the R-operator to solve the linear system $\left(\overline{\mathbf{G}}_\theta + \lambda \mathbf{I}\right) \delta = \nabla_\theta \mathcal{L}$ as in [81]. For the block diagonal Gauss-Newton method, we use the same strategy, but the method is applied independently for each separate layer of the network. We compared the different approaches for batch sizes of 250, 500 and 1000. Since the

results did not differ significantly, we, therefore, show results only for a batch size of 1000.

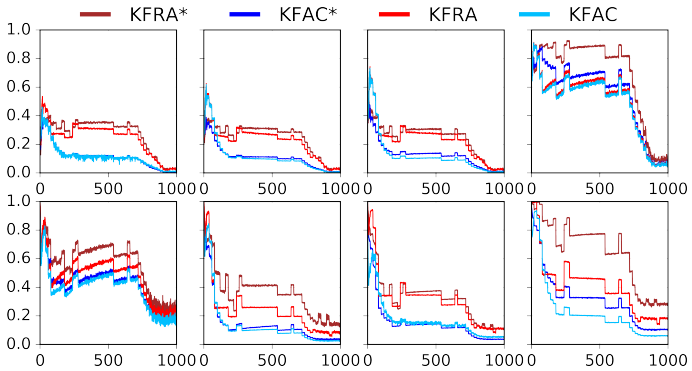
Figure 4.2 shows the cosine similarity between the update vectors $\widetilde{\delta}_l$ for each of the eight layers of the autoencoder model. As expected, the further the layer is from the output, the lower the similarity is as more information is lost due to the approximation error. We can also observe that in general KFRA produces better alignments than KFAC, however, in the case of the CURVES dataset, the approximate inverse seems to be significantly affecting this and making the updates the worst of the four variants. Additionally, one can notice that layers four and five, which are around the bottleneck of the network always have relatively high cosine similarity. We conjecture that the similarity could be affected by the dimensionality of the layers that the blocks are approximating. Figure 4.3 shows the cosine similarity between the full update vectors $\widetilde{\delta}$ for each approximation, including when using the exact block diagonal Generalised Gauss-Newton matrix. Surprisingly, on the MNIST dataset, the exact block diagonal matrix has significantly lower similarity than expected, and it gets worse over the training of the model. This could indicate that sometimes the approximations can condition better problems. Note that also it does not include the extra ζ parameter, which is non-zero and potentially plays a role in this. Comparing KFRA to KFAC, we see similar trends to the per layer cosine similarities, with KFRA outperforming KFAC on average. Notably, on both MNIST and FACES the approximate inversions seem to perform better. This could be due to better numerical stability of the operations, or just that the factorised diagonal damping improves the updates. In general it is difficult to analyse and understand curvature approximations in high dimensions and both figures confirm this. The main purpose of these was more to illustrate how nuanced such analysis can be and show that under different conditions different approximations "look" better. Nevertheless, in practice there is only one important metric - whatever the downstream task is (for instance optimisation) and we can not assert with great certainty that being closer in cosine similarity would always translate to better performance.



(a) CURVES

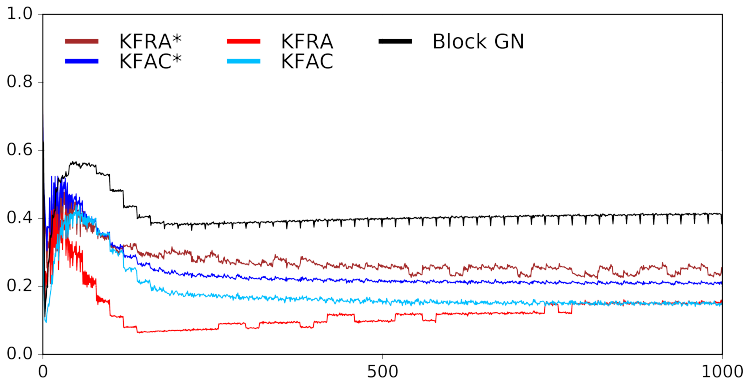


(b) MNIST

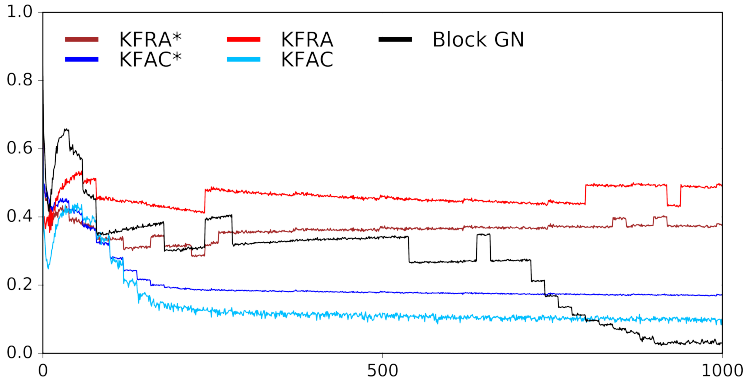


(c) FACES

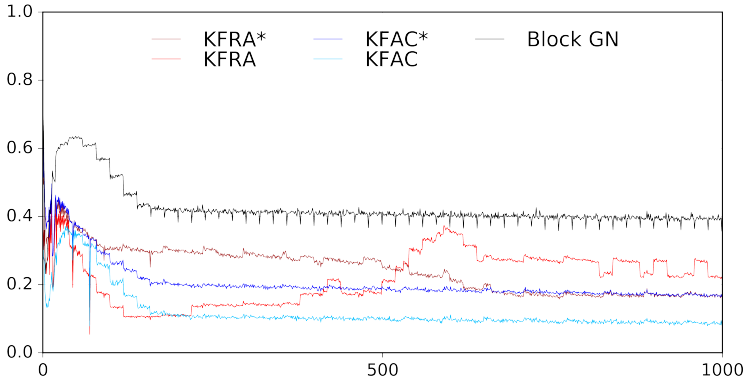
Figure 4.2: Alignment of the diagonal approximate updates. Each plot shows the cosine similarity between the updates generated from Kronecker factored approximations KFAC and KFRA, and the update generated when using the exact Gauss-Newton block for each of the eight layers of the network, ordered from left to right and top to bottom, through out the optimisation. The x axis denotes the iteration number. The algorithms denoted by $*$ exactly invert the Kronecker product of the factors plus $\lambda\mathbf{I}$ using eigenvalue decomposition, while the other variants use the approximate scheme described in Section 4.4.2.



(a) CURVES



(b) MNIST



(c) FACES

Figure 4.3: Alignment of the full approximate updates. Each plot shows the cosine similarity between the full update generated from the approximations KFAC and KFRA, the block diagonal Gauss-Newton, and the update generated when using the exact Generalised Gauss-Newton matrix. The x axis denotes the iteration number. The algorithms denoted by $*$ exactly invert the Kronecker product of the factors plus $\lambda\mathbf{I}$ using eigenvalue decomposition, while the other variants use the approximate scheme described in Section 4.4.2.

4.5.3 Non-Exponential Family Model

All of the experiments conducted so far are of Supervised Learning problems with log-likelihoods and exponential family predictive distributions, hence the Fisher and the Generalised Gauss-Newton matrix are equivalent. In order to experiment with a problem where the two are not equivalent we design a toy experiment with a Supervised Learning problem, where the predictive distribution is not in the exponential family. The problem is to do binary classification, but rather than using a Bernoulli distribution, instead mixture of two classifiers is chosen:

$$p(\mathbf{y}|\mathbf{h}_L) = \sigma(\mathbf{h}_1^L)\sigma(\mathbf{h}_2^L)^{\mathbf{y}}(1-\sigma(\mathbf{h}_2^L))^{1-\mathbf{y}} + (1-\sigma(\mathbf{h}_1^L))\sigma(\mathbf{h}_3^L)^{\mathbf{y}}(1-\sigma(\mathbf{h}_3^L))^{1-\mathbf{y}}. \tag{4.28}$$

where $\sigma(x) = \frac{1}{1+e^{-x}}$. We use the same architecture as for the encoding layers of the autoencoders — D-1000-500-250-30-3, where D is the size of the input. The dataset on which we test is MNIST, where the labels $\mathbf{y}$ represented whether the image is an even or an odd digit. Our choice was motivated by recent interest in neural network mixture models [29, 140, 99, 123]. Training was run for $40,000$ updates for ADAM with a grid search as in Section 4.5.1, and for $5,000$ updates for the second-order methods. The resulting training curves are shown in Figure 4.4.

For the CPU, both per iteration and wall clock time the second-order methods were faster than ADAM; on the GPU, however, ADAM was faster per wall clock time. The value of the objective function at the final parameter values was higher for second-order methods than for ADAM. However, it is important to keep in mind that all methods achieved a nearly perfect cross-entropy loss of around 10^{-8}. When so close to the minimum, we expect the gradients and curvature to be very small and potentially dominated by noise introduced from the mini-batch sampling. Additionally, since the second-order methods invert the curvature, they are more prone to accumulating numerical errors than first-order methods, which may explain this behaviour close to a minimum.

Interestingly, KFAC performed almost identically to KFLR, despite the fact that KFLR computes the exact pre-activation Gauss-Newton matrix. This suggests that in the low-dimensional output setting, the benefits from using the exact low-rank calculation are diminished by the noise and the rather coarse factorised Kronecker approximation.

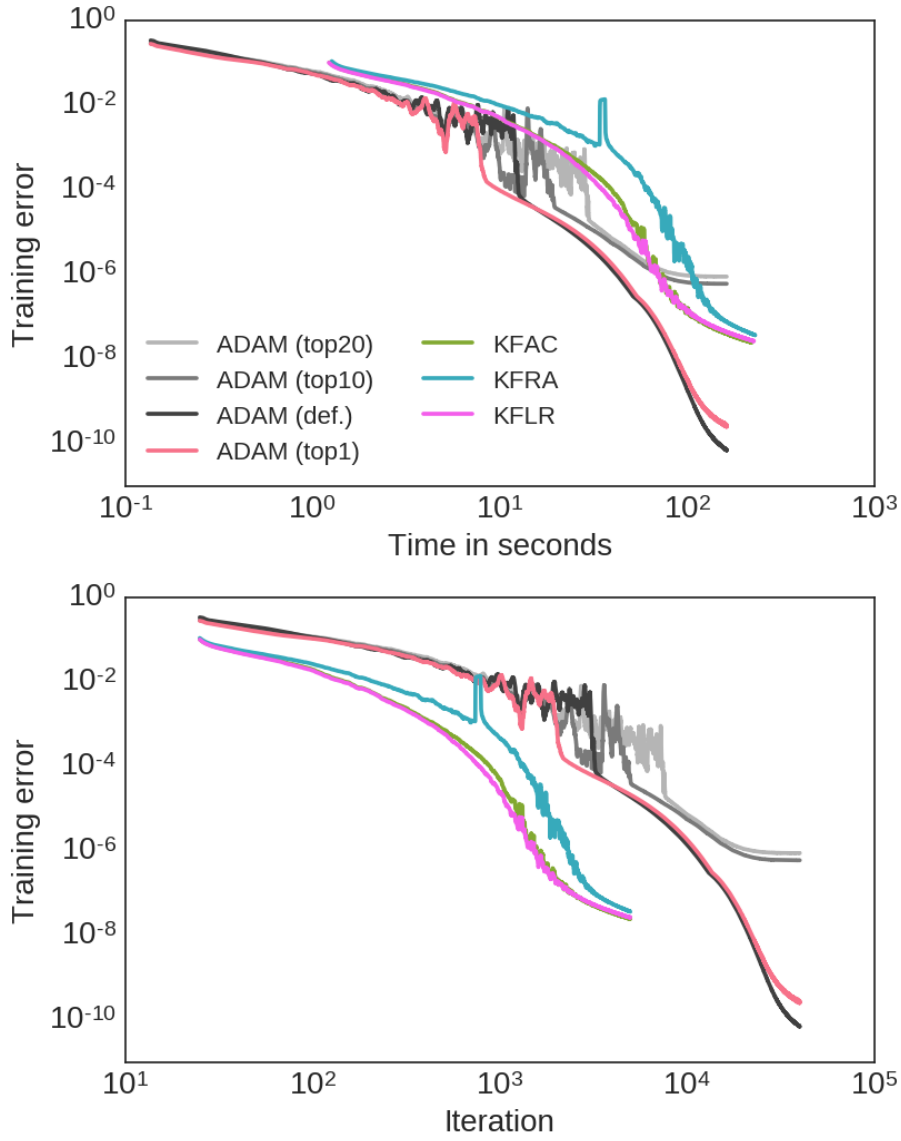


Figure 4.4: Fisher vs Gauss-Newton for non-exponential family models. KFAC corresponds to the Fisher matrix, while both KFRA and KFLR are using the Gauss-Newton matrix. The plots shows the optimisation performance on the MNIST binary mixture-classification model.

Chapter 5

Uncertainty estimation for Deep Learning models

In the previous chapters, we introduced some theoretical background on the structure of the Generalised Gauss-Newton matrix and demonstrated a scalable approximation that can successfully be used in an optimisation algorithm. Experimentally the algorithm achieved comparable results to state-of-the-art optimisers used to train Deep Learning models. In this chapter is presented the application of our scalable approximation to the curvature matrix for uncertainty estimation of neural network parameters. In practice, Deep Learning models are usually trained using MAP estimation, which provides no notion of such uncertainty as the result is a single parameter value. Some recent attempts to approximating the posterior distribution of neural network parameters are based on optimising a variational lower bound, treating the network parameters as latent variables:

$$\begin{aligned}
\log p(\mathcal{D}) &= \log \int p(\mathcal{D}|\theta)p(\theta) = \log \int \frac{q(\theta)}{q(\theta)} p(\mathcal{D}|\theta)p(\theta) \\
&= \log \mathbb{E}_{q(\theta)}\left[\frac{p(\mathcal{D}|\theta)p(\theta)}{q(\theta)}\right] \\
&\geq \mathbb{E}_{q(\theta)}\left[\log \frac{p(\mathcal{D}|\theta)p(\theta)}{q(\theta)}\right] \\
&= \mathbb{E}_{q(\theta)}\left[\log p(\mathcal{D}|\theta)\right] - \mathcal{D}_{\mathrm{KL}}(q(\theta)||p(\theta)).
\end{aligned} \tag{5.1}$$

In practice the form of the approximate distribution $q(\theta)$ is very simple. The approaches proposed in [43, 8, 63] as well as the expectation propagation based methods of [52] and [39] assume independence between all individual weights. Since optimising the lower bound is equivalent to optimising the reverse KL divergence

$\mathcal{D}_{\text{KL}}(q(\theta)||p(\theta|\mathcal{D}))$, a procedure that is known to be mode seeking, it often leads to significantly underestimating the uncertainty over the parameters.

The most common practice up to date for getting uncertainty estimates is probably Dropout [126, 34]. The method was initially introduced as randomly dropping units in the network along with their connection to prevent overfitting. This was achieved by sampling a binary mask $\mathbf{m}_l$ for every layer and multiplying the preactivations in the network with the mask [1]:

$$\mathbf{h}_l = \mathbf{h}_l \circ \mathbf{m}_l. \tag{5.2}$$

Usually, each binary value of $\mathbf{m}_l$ is sampled from a Bernoulli distribution with a probability p across the whole network. Since the masks can be "moved" to be part of the weight matrix, [35] reinterpret this procedure as an approximate variational procedure, where the distribution q is a mixture of all possible combination of masks. This, however, was shown not to be mathematically rigorous as the approximate distribution is degenerate and the KL divergence does not exist [57]. Nevertheless, this method has been widely used in the literature, achieving great success and being developed further [74, 34, 63, 33].

A somewhat orthogonal direction to previous methods is the usage of model ensembles [67, 102]. Each ensemble member is typically obtained by training the same network from a different parameter initialisation. The main reason why these are more of a complementary rather than a competing approach is that in the same spirit one can create an ensemble of Dropout networks, Laplace approximations or any other method that is applicable to a single model. Hence in this work, we will not be evaluating any of the methods via ensembles.

The approach that we develop in this chapter for uncertainty estimation is based on the work of [79]. Using the scalable approximation to the Generalised Gauss-Newton matrix from earlier chapters, we construct and approximate Laplace distribution. We validate the usefulness of the distribution by comparing its uncertainty estimates to Dropout on out of distribution data empirically. Thereafter, the Laplace approximation is applied to the problem of online learning, where we demonstrate that it outperforms other methods in the literature.

The theoretical development of using the curvature approximations from previous chapters, was done in collaboration with my supervisor Prof. David Barber and my college Hippolyt Ritter at University College London. Me and my college

[1] Where $\circ$ stands for elementwise product, also called the Hadamard product.

were responsible for implementing the code required to run the sampling procedures required for the Laplace approximation. The implementation and execution of the experiments was done by Hippolyt Ritter. Similarly, in the work on online learning, the experimental work was done by him, while me and Prof. David Barber participated in developing the theoretical framework. All three of us contributed for writing up and presenting our published work in [113, 114].

5.1 A scalable Laplace approximation

The Laplace method was originally developed for approximating integrals of functions of the following form:

$$f(\mathbf{x}) = e^{g(\mathbf{x})}. \tag{5.3}$$

Assume that the exponent $g(\mathbf{x})$ is a smooth function, which has a relatively high peak around its maximum value. In order to approximate the integral of f Laplace suggested to approximate f using a Gaussian density function which has known analytical expressions for its integrals [68]. To do so he proposed to Taylor expand the exponent function around its maximum value:

$$g(\mathbf{x}) \approx g(\mathbf{x}_*) + (\mathbf{x} - \mathbf{x}_*)^\mathsf{T} \nabla_{\mathbf{x}_*} g(\mathbf{x}) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_*)^\mathsf{T} \nabla^2_{\mathbf{x}_*} g(\mathbf{x})(\mathbf{x} - \mathbf{x}_*), \tag{5.4}$$

where $\mathbf{x}^*$ is the argument for which g attains its maximal value. This is equivalent to the quadratic approximation used for for constructing second-order optimisers applied to $g(\mathbf{x})$. Since by assumption $\mathbf{x}_*$ is a strict maximiser of g, the second-order term $\nabla^2_{\mathbf{x}_*} g(\mathbf{x})$ is guaranteed to be Negative Semi-Definite. Additionally, this also implies that ${}^\mathsf{T}\nabla_{\mathbf{x}_*} g(\mathbf{x}) = 0$ and hence the first-order term vanishes. Examining the logarithm of the Gaussian probability density function in terms of its dependencies on x:

$$\log \mathcal{N}(\boldsymbol{\mu}, \Sigma) = \text{const} - \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\mathsf{T} \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu}), \tag{5.5}$$

if we set $\Sigma = -\left(\nabla^2_{\mathbf{x}_*} g(\mathbf{x})\right)^{-1}$ and $\boldsymbol{\mu} = \mathbf{x}_*$ the two expressions have the same functional form. This Gaussian approximation is exactly what the Laplace method does.

This method is particularly well suited when trying to approximate a distribution, since by definition any distribution with full support can be written in this form. Consider the problem of estimating the posterior distribution of a parametric model using the Bayesian framework:

$$p(\theta|\mathcal{D}) = \frac{p(\mathcal{D}|\theta)p(\theta)}{p(\mathcal{D})}. \tag{5.6}$$

In complicated models, such as neural networks, the posterior is intractable. Standard training under MAP estimation provides a value θ^* that is a local maximiser of the posterior distribution. Assuming as before that the prior is an isotropic Gaussian with precision λ would imply that the Hessian of the log-posterior is nothing more than $-\overline{\mathbf{H}}_\theta - \frac{\lambda}{N}\mathbf{I}$. As discussed in Section 3.5 the Hessian of neural network is not guaranteed to be Positive Semi-Definite, and hence this expression can not be used as the covariance matrix of a Gaussian distribution, unless it is a strict maximum. However, any practical training algorithms achieve a maximum of the log-posterior only approximately due to the fact that they rely on stochastic gradients. As a result, instead of using the Hessian it is possible to use the Generalised Gauss-Newton or the Fisher matrix for a practical Gaussian approximation. [2] Using the notations from Section 4.4 for the curvature matrix $\mathbf{C} = \overline{\mathbf{G}}_\theta + \lambda\mathbf{I}$ and representing with $\mathbf{C}^*$ the matrix evaluated at the parameter value θ^* the approximate posterior is

$$p(\theta|\mathcal{D}) \approx \mathcal{N}(\theta^*, \frac{1}{N}\mathbf{C}^{*-1}). \tag{5.7}$$

For many applications the goal is to improve predictions on new data, which constitutes calculating the predictive distribution:

$$p(\mathbf{y}|\mathbf{x}, \mathcal{D}) = \int_\theta p(\mathbf{y}|\mathbf{x}, \mathcal{D}, \theta)p(\theta) = \int_\theta p_\theta(\mathbf{y}|\mathbf{x})p(\theta|\mathcal{D}). \tag{5.8}$$

Using the Gaussian approximation from the Laplace method for $p(\theta|\mathcal{D})$, which will be denoted as $q(\theta|\mathcal{D})$, the above integral can be approximated via Monte Carlo:

$$p(\mathbf{y}|\mathbf{x}, \mathcal{D}) \approx \int_\theta p_\theta(\mathbf{y}|\mathbf{x})q(\theta|\mathcal{D}) \approx \frac{1}{K}\sum_k p_{\theta_k}(\mathbf{y}|\mathbf{x}) \qquad \theta_k \sim q(\theta|\mathcal{D}). \tag{5.9}$$

5.1.1 Practical approximations

Unfortunately, as discussed in the Chapter 4, it is not feasible to compute or invert the full Generalised Gauss-Newton matrix for a Deep Learning model. As a result the following two approximations will be investigated. The first one aims to approximate only the diagonal of the curvature matrix. The benefits of this approach is that it has a very low memory cost and is easy to invert. Due to the equivalence of the Generalised Gauss-Newton and Fisher in order to compute this approximation, in practice we use the Fisher identity from Section 3.5.1:

$$\begin{aligned} \mathbf{diag}\left(\mathbf{F}\right) &= \mathbf{diag}\left(\mathbb{E}_{p_\theta(\mathbf{x})}\left[\nabla_\theta \log p_\theta(\mathbf{x}) \nabla_\theta \log p_\theta(\mathbf{x})^\mathsf{T}\right]\right) \\ &= \mathbb{E}_{p_\theta(\mathbf{x})}\left[\nabla_\theta \log p_\theta(\mathbf{x})^2\right], \end{aligned} \tag{5.10}$$

[2] In this setting the two matrices are equivalent.

where the square operation is taken elementwise. This diagonal approximation to the curvature has been used successfully for pruning the weights [71] and, more recently, for transfer learning in [64]. This corresponds to modelling each layer weights as the following independent Gaussian distributions:

$$\mathbf{w}_l \sim \mathcal{N}\left(w_l^*, N\mathbf{diag}\left(\overline{\mathbf{G}}_{l,l}^* + \lambda\mathbf{I}\right)^{-1}\right). \tag{5.11}$$

The second approach we propose is to use the block diagonal approximations developed earlier, in which each block is further approximated via Kronecker product:

$$\overline{\mathbf{G}}_{l,l} \approx \overline{\mathcal{A}}_{l,l} \otimes \overline{\mathcal{G}}_l. \tag{5.12}$$

Using this representation as a covariance of a Gaussian distribution corresponds to a Matrix-Normal distribution [45]:

$$\mathbf{w}_l \sim \mathcal{MN}\left(w_l^*, \overline{\mathcal{A}}_{l,l}^{*-1}, \overline{\mathcal{G}}_l^{*-1}\right). \tag{5.13}$$

However, this approximates only $\overline{\mathbf{G}}$ and in practice there is also a diagonal identity term coming from the prior distribution $p(\theta)$. Since adding an identity to a Kronecker product destroys this structure, we use the tactic from Section 4.4.2 to come up with the final layer-wise approximation:

$$\mathbf{w}_l \sim \mathcal{MN}\left(w_l^*, \left(\sqrt{N}\overline{\mathcal{A}}_{l,l}^* + \sqrt{\lambda}\mathbf{I}\right)^{-1}, \left(\sqrt{N}\overline{\mathcal{G}}_l^* + \sqrt{\lambda}\mathbf{I}\right)^{-1}\right). \tag{5.14}$$

This approximate posterior is reminiscent of [78] and [127], who optimise the parameters of a matrix normal distribution as their weights. However, that work is optimising a variational lower bound, which usually leads to much worse performing models.

Although N is the dataset size and λ corresponds to the precision of the Gaussian prior, they can instead be treated as hyperparameters, in order to improve even further the approximation. Their values can be selected by looking at the performance of the approximate predictive distribution on a validation set. This does not require any retraining of the model and adds a computational overhead only once when computing the posterior after training. Setting N to a larger value than the size of the dataset can be interpreted as including duplicates of the data points as pseudo-observations. Manipulating λ from its initial value could modify the uncertainty about each layer's parameters. This has a regularising effect both on the block diagonal approximation to the true Laplace, which may be overestimating the

variance in certain directions due to ignoring the covariances between the layers, as well as the Laplace approximation itself, which may be placing probability mass in low probability areas of the true posterior.

In contrast to optimisation methods, computing the curvature matrix has to be done only once after training is complete. This means that it is not time-critical and hence it is possible to use methods like KFLR or running KFRA with a batch size of one to compute the exact value of $\overline{\mathcal{G}_l}$ and $\overline{\mathbf{G}}_{l,l}$ in turn. This is technically not possible with methods relying on the Fisher identity like KFAC, which always uses a stochastic approximation, however with enough samples, their errors would become sufficiently small. On the other hand, for very large datasets, such as ImageNet, and huge models, it could still be impractically slow to perform the exact computation. Additionally, it is also common in many image classification tasks to use data augmentation, like random cropping or reflections of images, in order to boost the effective dataset size. This practice can increase the number of possible inputs several order of magnitude, and in these cases, it would be required to use an approximation. Hence, in our practical implementation, we make use of the minibatch approximations, since we also use data augmentation in several of our experiments in Section 5.2, in order to demonstrate the wider applicability of the method.

5.2 Experiments on uncertainty estimation

Since the Laplace approximation is a method for *predicting* in a Bayesian manner and not for training, we focus on comparing to uncertainty estimates obtained from Dropout [35]. The trained networks will be identical, but the prediction methods will differ. In addition, a diagonal Laplace approximation is included in the comparisons to highlight the benefit of modelling the covariances between the weights. The datasets being used are:

Toy 1D is a small synthetic dataset similar to [52]. To generate it 20 uniformly distributed points from the interval $[-4, 4]$ are sampled and for each one a regression value y is sampled from $\mathcal{N}(x^3, 3^2)$.

Binarised MNIST is a modified version of the MNIST dataset described in Section 4.5. Each pixel of the inputs provided to the models is sampled randomly from a Bernoulli distribution according to the pixel intensity of the grey-scale image, which makes the effective dataset size significantly larger than the original grey-scale dataset.

notMNIST contains 28×28 grey-scale images of the letters 'A' to 'J' from various computer fonts, i.e. not digits [13]. The main goal of this dataset is mimic as close as possible the MNIST dataset, but have other ten object classes.

CIFAR100 consists of $60,000$ 32×32 colour images in 100 classes with exactly 600 images per class. [65]. As common practice, only the first $50,000$ images are used for training, as the other $10,000$ are usually used for validation or testing.

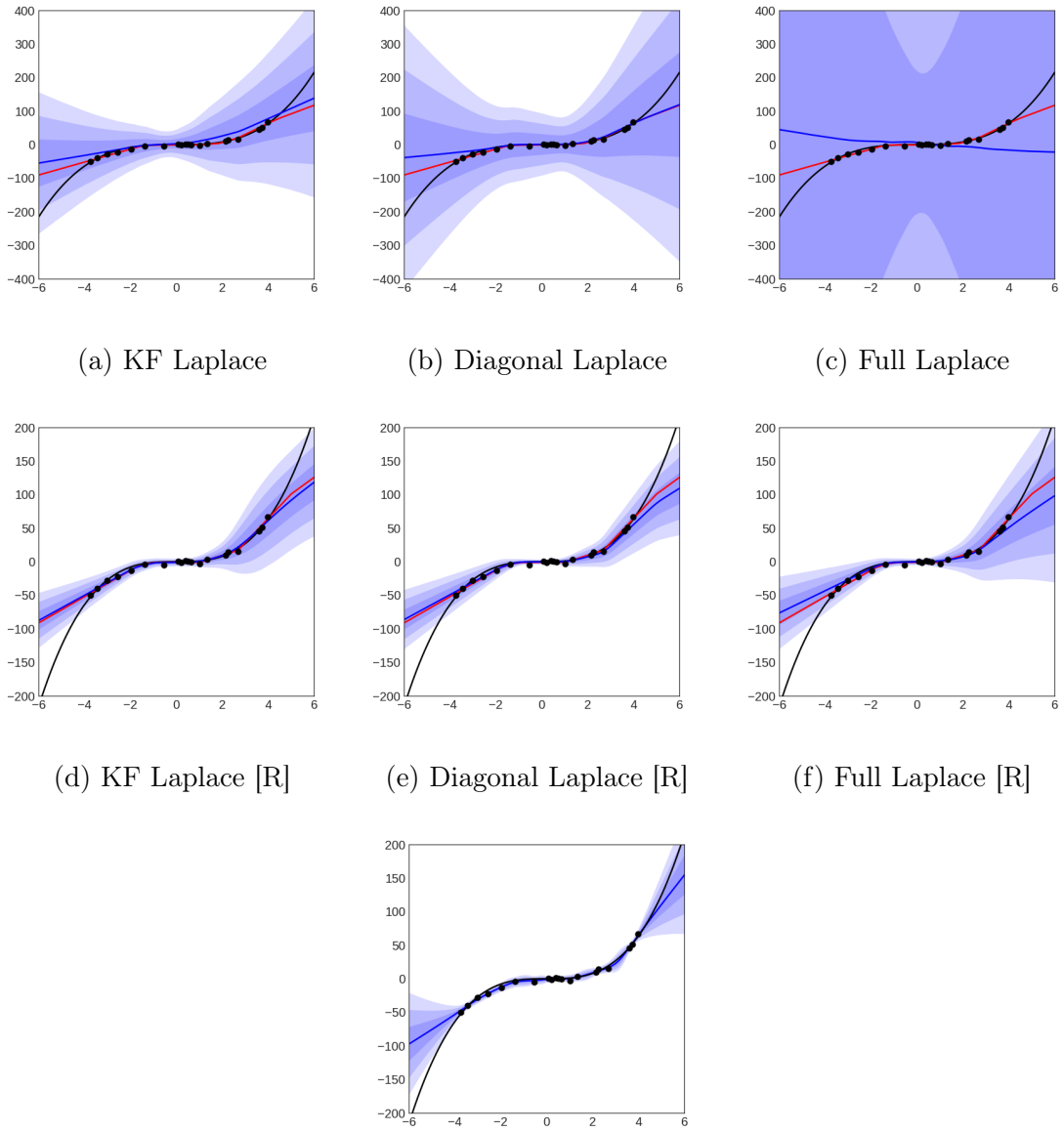
All experiments are implemented using Theano and Lasagne [112, 27].

5.2.1 Toy Regression Dataset

As an initial experiment, we investigate the uncertainty obtained from the Laplace approximations on the Toy 1D dataset. In contrast to previous work on toy data [52], we use a two-layer network with seven units per layer rather than one layer with 100 units. The main reason for this is because the inputs and the outputs are one-dimensional, hence for a single layer network, the weights become vectors and the matrix normal reduces to just a multivariate Gaussian. Furthermore, we are interested in investigating how sensitive is the Laplace approximation to the ratio of the number of data points to the number of parameters and how hyperparameter tuning affects it.

For comparative purposes, except the diagonal and Kronecker factored Laplace approximations, we also compute the full Laplace. For the diagonal and full Laplace approximations, we use the Fisher identity and draw one sample per data point. When the values N and λ in Equation 5.14 are treated as hyperparameters, we will refer to the method as being "regularised". In these cases, their values are set using a grid search over the likelihood of 20 validation points, sampled separately from the same distribution as the training dataset. In an attempt to approximate the full posterior distribution, we run Hamiltonian Monte Carlo (HMC) and obtain $50,000$ samples of parameters as in [94]. This and the full Laplace approximation are very computationally intensive methods and are feasible only for such a small dataset and model.

The predictive distributions from all of the methods described on the toy dataset are depicted in Figure 5.1. All of the Laplace approximations are increasingly uncertain away from the data, as the true posterior estimated using HMC samples. Without the regularisation, we can see that all of them significantly overestimate



(a) KF Laplace (b) Diagonal Laplace (c) Full Laplace

(d) KF Laplace [R] (e) Diagonal Laplace [R] (f) Full Laplace [R]

(g) HMC

Figure 5.1: Toy regression uncertainty. Black dots are data points, the black line shows the noiseless function. The red line shows the deterministic prediction of the network, the blue line the mean output. Each shade of blue visualises one additional standard deviation. The [R] indicates that the method has been regularised by treating N and λ as hyperparameters as discussed in the text. Best viewed in colour.

the uncertainty in the vicinity of the training data. This is significantly mitigated by using the hyperparameter search discussed earlier for modifying N and λ. With the best hyperparameter setting, all approximations give an overall good fit to the HMC predictive posterior. Recall from our discussion earlier in Section 3.6 that the Generalised Gauss-Newton of a neural network is usually significantly underdetermined as the number of data points is much smaller than the number of parameters — in our case we have 20 data points to estimate a 78×78 precision matrix. This potentially explains why the full Laplace approximation overestimates the uncertainty significantly more and has a much worse predictive mean. As a result, it requires much stronger regularisation parameters N and λ compared to the other two methods. Similarly, the diagonal approximation's hyperparameters are larger than those of the Kronecker factored distribution. Consistently with the experiments presented next, we find the diagonal Laplace approximation to place more mass in low probability areas of the posterior than the Kronecker factored approximation, resulting in higher variance on the regression problem. This indicates that restricting the structure of the covariance is not only a computational necessity for most architectures but also mitigates some of the issues of the full Laplace approximation.

5.2.2 Out-of-Distribution Uncertainty

For a more realistic test, similar to previous work, we assess the uncertainty of the predictions when classifying data from a different distribution than the training data [77]. The dataset on which we train the models is the Binarised MNIST. After the model being trained, and any posterior distribution parameters have been estimated, we test the network predictive distribution when applied to images from the notMNIST. An ideal classifier would make uniform predictions over its classes.

The neural network used in this experiment consists of two layers of 1024 hidden units and ReLU activation functions. During training, the precision of the Gaussian prior is set to 10^{-2}. In addition to the diagonal and Kronecker factored Laplace approximation, we train a model using Dropout for which we set the dropout probability to 0.5 on each of the hidden layers [126]. As an additional baseline similar to [8, 43], we compare to training an approximate diagonal Gaussian, optimised by maximising the standard variational lower bound via the reparametrisation trick [62]. The hyperparameters of the Laplace approximations are set based on the cross-entropy loss on the validation set of MNIST. All algorithms use a learning rate of 10^{-2} and momentum of 0.9 during training for 250 epochs. For estimating the cur-

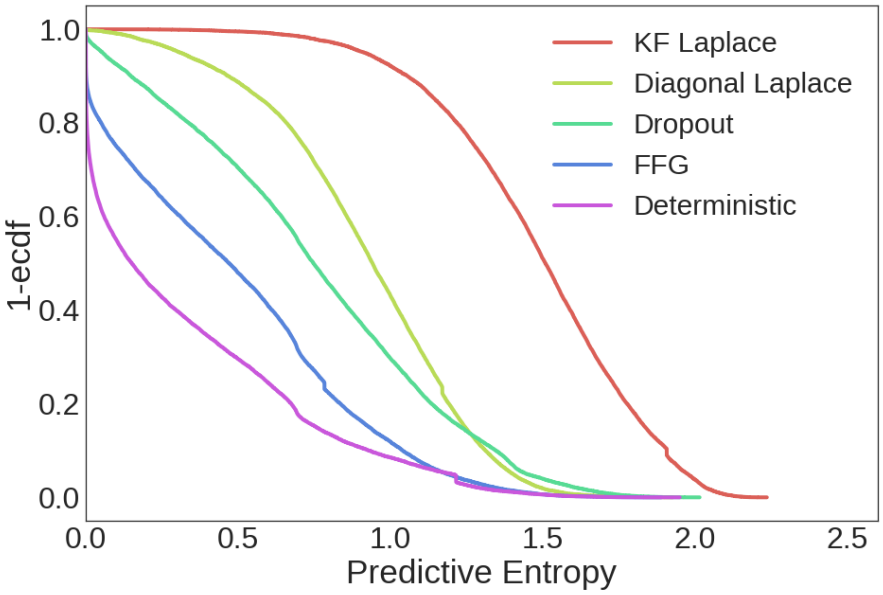


Figure 5.2: Predictive entropy of different posterior approximation when test on out-of-distribution data. The training data is a dynamically binarised MNIST dataset and the test data is the notMNIST dataset.

vature matrices, we use $1,000$ binary inputs sampled for each image in the dataset in the same way as they are provided during training. To summarise, we compare the uncertainty obtained by predicting the digit class of the notMNIST images using

1. a deterministic forward pass through the Dropout trained network,
2. by sampling different Dropout masks and averaging the predictions, and by sampling different weight matrices from
3. the Kronecker factored approximate Laplace distribution
4. the diagonal approximate Laplace distribution
5. the fully factorised Gaussian (FFG) trained using a variational lower bound.

Each method uses 100 Monte Carlo samples from their approximate posterior distributions.

The uncertainty of each method is measured by the entropy of the predictive distribution. For this specific data, it has a minimum value of 0 when a single class is predicted with certainty and a maximum of about 2.3 for uniform predictions. For

Prediction Method	Accuracy
FFG	98.88%
Deterministic	98.86%
MC Dropout	98.85%
Diagonal Laplace	98.85%
KF Laplace	98.80%

Table 5.1: Test accuracy of the Feed Forward Neural Network trained on MNIST

each data point in notMNIST we calculate the entropy and then compute an empirical cumulative distribution over this range. Figure 5.2 shows the inverse empirical cumulative distribution for each approach (this is one minus the cumulative distribution value). If a method does not have any predictions with entropy value below c, the curve in the image would be equal to 1 for any value on the x axis below c. Consistent with the results in [35] for Dropout, averaging the probabilities of multiple passes through the network yields predictions with higher uncertainty than a deterministic pass that approximates the geometric average [126]. However, there still are some images that are predicted to be a digit with certainty. Our Kronecker factored Laplace approximation makes hardly any predictions with absolute certainty and assigns high uncertainty to most of the letters as desired. The diagonal Laplace approximation required stronger regularisation towards predicting deterministically, yet it performs similarly to Dropout. The variational factorised Gaussian posterior has low uncertainty as expected. Just predicting with high uncertainty, although desired, can be achieved trivially by just always outputting a uniform distribution and is not sufficient for a model to be useful. To measure whether the approximate posterior distributions also have captured the original training data in Table 5.1, we show their predictive accuracy on the test set of MNIST. In all cases, neither MC Dropout nor the Laplace approximation significantly changes the classification accuracy of the network in comparison to a deterministic forward pass.

5.2.3 Adversarial Examples

To further test the robustness of our prediction method close to the data distribution, we perform an adversarial attack on the Deep Learning model. It has been experimentally demonstrated that neural networks are prone to being fooled by gradient-based changes to their inputs [129]. However, Bayesian models may be

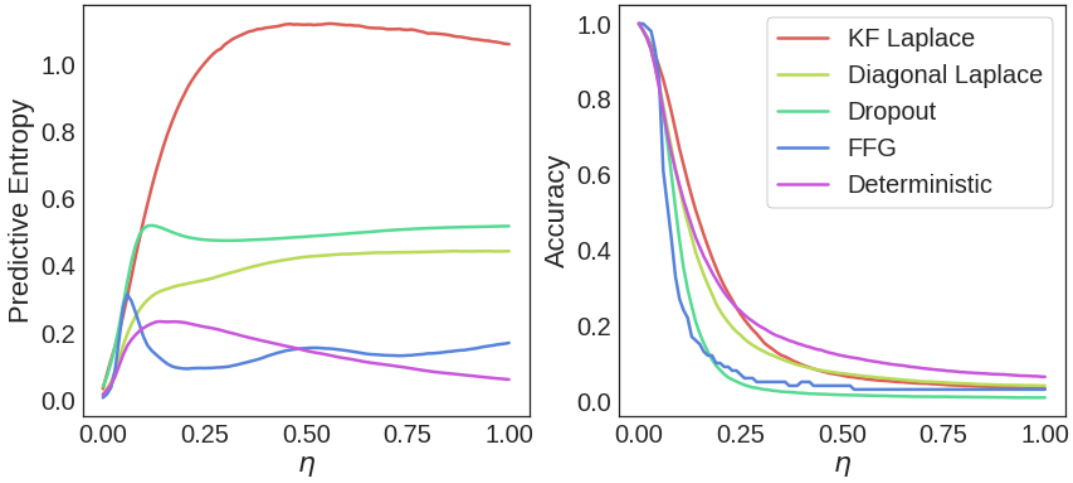


Figure 5.3: Uncertainty in untargeted adversarial attacks. The left plot shows the predictive entropy, while the right plot shows the predictive accuracy of different methods as a function of the step size of the attack. The adversarial image is generated using the Fast Gradient method.

more robust to such attacks since they implicitly form an infinitely large ensemble by integrating over the model parameters and empirical support of this claim has been demonstrated in [74]. For our experiments, we use the fully connected net trained on MNIST from the previous section and compare the sensitivity of the different prediction methods for two kinds of adversarial attacks.

First, we test for untargeted adversarial attacks using the untargeted Fast Gradient Sign method [42]. To construct an adversarial example, it computes the gradient of the class predicted with maximal probability from method m with respect to the input x, takes only the sign of each element and moves the input in the opposite direction with a varying step size η:

$$\mathbf{x_{adv}} = \mathbf{x} - \eta \mathbf{sign}(\nabla_{\mathbf{x}} \max_y \log p^{(m)}(y|\mathbf{x})). \tag{5.15}$$

This step size is rescaled by the difference between the maximal and minimal value per dimension in the dataset. It is to be expected that this method generates examples away from the data manifold, as there is no clear subset of the data that the results correspond to, e.g. there are no "not ones".

Figure 5.3 shows the average predictive uncertainty and the accuracy of the original class on the MNIST test set as the step size η increases. The Kronecker factored method achieves significantly higher uncertainty than any other prediction method as the images move away from the data. Both Laplace approximations maintain

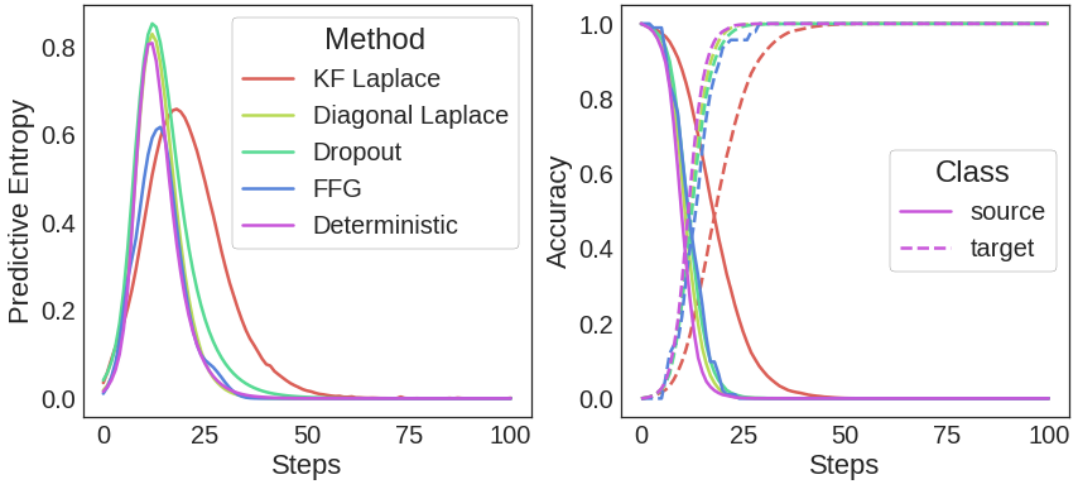


Figure 5.4: Uncertainty in targeted adversarial attacks. The left plot shows the predictive entropy, while the right plot shows the predictive accuracy of different methods as a function of the step size of the attack. The adversarial image is generated with the goal of making models to predict the class digit 0 from MNIST.

higher accuracy than the Monte Carlo Dropout on their original predictions. The FFG method seems to have the worst performance. It does not seem to capture well any uncertainty as well its accuracy drops pretty quickly away from the original example. Interestingly, the deterministic forward pass appears to be most robust in terms of accuracy. However, it has much smaller uncertainty on the predictions it makes and will confidently predict a false class for most images.

Additionally, we perform a targeted attack that attempts to force the network to predict a specific class, in our case 0 following [74]. Hence, for each method, we exclude all data points in the test set that are already predicted as 0. The updates are of similar form to the untargeted attack, however they increase the probability of the pre-specified class y rather than decreasing the current maximum:

$$\mathbf{x}_y^{t+1} = \mathbf{x}_y^t + \eta \mathbf{sign}(\nabla_{\mathbf{x}} \log p^{(m)}(y|\mathbf{x}_y^{(t)}), \tag{5.16}$$

and initialising with $\mathbf{x}_y^{(0)} = \mathbf{x}$. We use a step size of $\eta{=}10^{-2}$ for the targeted attack. The uncertainty and accuracy on the original and target class are shown in Figure 5.4. Here, the Kronecker factored Laplace approximation has slightly smaller uncertainty at its peak in comparison to the other methods; however, it appears to be much more robust as the number steps increases. It only misclassifies a little over 50% of the images after about 20 steps, whereas for the other methods this is the case after roughly 10 steps and reaches 100% accuracy on the target class after

almost 50 updates, whereas the other methods are fooled on all images after about 25 steps.

In conjunction with the experiment on notMNIST, it appears that the Laplace approximation achieves higher uncertainty than Dropout away from the data, as in the untargeted attack. In the targeted attack, it exhibits smaller uncertainty than Dropout, yet it is more robust to having its prediction changed. The diagonal Laplace approximation again performs similarly to Dropout.

5.2.4 Uncertainty on Misclassifications

To highlight the scalability of our method, we apply it to a state-of-the-art convolutional network architecture. Recently, deep residual networks have been the most successful ones among those [49, 51]. As demonstrated in the literature, Kronecker factored curvature methods are applicable to convolutional layers by interpreting them as matrix-matrix multiplications [44].

We compare our uncertainty estimates on wide residual networks [137], a recent variation that achieved competitive performance on CIFAR100 while, in contrast to most other residual architectures, including Dropout at specific points. While this does not correspond to using Dropout in the Bayesian sense, it allows us to at least compare our method to the uncertainty estimates obtained from Dropout.

Our wide residual network has 3 block repetitions and a width factor of 8 with and without Dropout using the hyperparameters taken from [137]. The network parameters are trained on a cross-entropy loss using Nesterov momentum with an initial learning rate of 0.1 and momentum of 0.9 for 200 epochs with a minibatch size of 128. We decay the learning rate every 50 epochs by a factor of 0.2, which is slightly different to the schedule used in the original wide residual network experiments (they decay after 60, 120 and 160 epochs). As the original authors, we use L_2-regularisation with a factor of $5{\times}10^{-4}$. We make one small modification to the architecture: instead of downsampling with $1{\times}1$ convolutions with stride 2, we use $2{\times}2$ convolutions. This is due to Theano not supporting the transformation of images into the patches extracted by a convolution for $1{\times}1$ convolutions with stride greater than 1, which we require for our curvature backpropagation through convolutions.

We apply a standard Laplace approximation to the batch normalisation parameters — a Kronecker factorisation is not needed since the parameters are one-dimensional. When calculating the curvature factors, we use the moving averages

	Accuracy	
Prediction Method	Dropout	Deterministic
Deterministic	79.12%	79.18%
MC Dropout	79.20%	-
KF Laplace	79.10%	79.36%

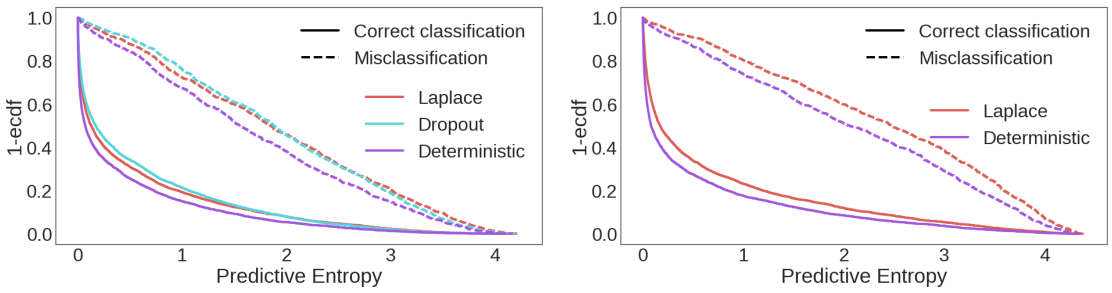
Table 5.2: Accuracy on the final 5,000 CIFAR100 test images for a wide residual network trained with and without Dropout.

for the per-layer means and standard deviations obtained after training, in order to maintain independence between the data points in a minibatch. We are not aware of any interpretation of Dropout as performing Bayesian inference on the parameters of batch normalisation at the time of writing.

The accuracy of predictions from different methods is displayed in Table 5.2. All models achieve comparable results as expected. For calculating the curvature factors, we draw 5,000 samples per image using the same data augmentation as during training, effectively increasing the dataset size to 2.5×10^8. We use the first 5,000 images as a validation set to tune the hyperparameters of our Laplace approximation and the final 5,000 ones for evaluating the predictive uncertainty on all methods. The diagonal approximation had to be regularised to the extent of becoming deterministic, so we omit it from the results.

Figure 5.5 depicts the predictive uncertainty of different approximations on the test set. We distinguish between the uncertainty on correct and incorrect classifications, as the mistakes of a system used in practice may be less severe if the network can at least indicate that it is uncertain. Thus, high uncertainty on misclassifications and low uncertainty on correct ones would be desirable, such that a system could return control to a human expert when it can not make a confident decision. In general, the network tends to be more uncertain on its misclassifications than its correct ones regardless of whether it was trained with or without Dropout and of the method used for prediction. Both Dropout and the Laplace approximation similarly increase the uncertainty in the predictions irrespectively of the correctness of the classification. Yet, our experiments show that the Kronecker factored Laplace approximation can be scaled to modern convolutional networks and maintain good classification accuracy while having similar uncertainty about the predictions as Dropout.

We had to use much stronger regularisation for the Laplace approximation on the



(a) Dropout Wide Resnet

(b) Deterministic Wide Resnet

Figure 5.5: Both plots show the inverse cumulative density function of the predictive entropy from Wide Residual Networks trained with and without Dropout on CIFAR100. The left figure demonstrates the result for a network with Dropout, while the right one for a deterministic network. Dashed lines indicate the entropy for missclassified images and straight for correctly classified.

wide residual network, possibly because the block-diagonal approximation becomes more inaccurate on deep networks and the number of parameters is much higher relative to the number of data. It would be interesting to see how the Laplace approximations behave on a much larger dataset like ImageNet for similarly sized networks, where we have a better ratio of data to parameters and curvature directions. However, even on a relatively small dataset like CIFAR we did not have to regularise the Laplace approximation to the degree of the posterior becoming deterministic.

5.3 Online learning

Creating an agent that performs well across multiple tasks and continuously incorporates new knowledge has been a longstanding goal of research on artificial intelligence. A stepping stone in this direction is to have a model that has the capabilities to train on a sequence of tasks and successfully learn all of them. In practice, however, many machine learning algorithms in this setting suffer from what has been termed "catastrophic forgetting" [31, 88, 110] — as they go through more tasks in the sequence they stop performing well on earlier ones. This has recently has received more attention in the context of Deep Learning [41, 64]. Unfortunately, the naive approach of setting the initial parameters for a new task with the optimal ones from the previous tasks does not work [41]. As stochastic gradient descent

does not necessarily remain sufficiently close to the original values, as new tasks are observed, performance on earlier ones degrades further.

Bayesian learning provides an elegant solution to this problem. In this section, we combine the framework of Bayesian online learning [101] with the Kronecker factored Laplace approximation developed in Section 5.1 to continuously update a posterior distribution over the weights of the model. The experiments in the next section demonstrate the effectiveness of this approach with significant gains over previously published methods, particularly on a long sequence of tasks.

5.3.1 Bayesian online learning for neural networks

The goal under consideration is to optimise the parameters θ of a single neural network to perform well across multiple tasks $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_T$. In the case that we are interested, under a probabilistic framework, this corresponds to finding the MAP estimate of the full posterior distribution:

$$\theta^* = \arg\max_{\theta} p(\theta|\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_T). \tag{5.17}$$

In the context of online learning, it is further assumed that tasks arrive sequentially, and it is possible to train on only one of them at a time. To address this, we will use the Bayesian online learning framework [101]. Applying Bayes' rule to the full posterior distribution in Equation 5.17 following sequential relationship can be derived:

$$p(\theta|\mathcal{D}_{1:t+1}) = \frac{p(\mathcal{D}_{t+1}|\theta)p(\theta|\mathcal{D}_{1:t})}{p(\mathcal{D}_{t+1}|\mathcal{D}_{1:t})}. \tag{5.18}$$

This clearly reveals a principled way to online learning. Given the posterior distribution from all of the previous tasks $p(\theta|\mathcal{D}_{1:t})$ one has to multiply it with the likelihood of the new task and renormalise accordingly. This can be alternatively be formulated as carrying over the posterior from previous tasks as a prior distribution over the parameters and learning the new task. As it has been discussed earlier, in Deep Learning models the posterior distribution is intractable, and hence this procedure can not be computed exactly. Assumed Density Filtering [87] formulates a parametric approximate posterior q with parameters ψ_t which is iteratively updated in two steps:

Update step The approximate posterior q_t from the previous tasks is used instead of the true posterior $p(\theta|\mathcal{D}_{1:t})$ in Equation 5.18 to find a new posterior given the most

recent data:

$$p(\theta|\mathcal{D}_{t+1}, \psi_t) = \frac{p(\mathcal{D}_{t+1}|\theta)q(\theta|\psi_t)}{p(\mathcal{D}_{t+1}|\psi_t)}. \tag{5.19}$$

Projection step The projection step finds the distribution within the parametric class of the approximation that most closely resembles the posterior computed in the update step:

$$q(\theta|\psi_{t+1}) \approx p(\theta|\mathcal{D}_{t+1}, \psi_t). \tag{5.20}$$

One criteria for accomplishing this is to find the parameters ψ_{t+1} by minimising the KL divergence between the two distributions [101]. This, however, is mostly appropriate for models where both distributions are available in closed form. Since the posterior distribution is generally not tractable for models such as neural networks, the Laplace approximation from the previous section will be used instead. To incorporate this the two iterative steps of Assumed Density Filtering are modified using a Gaussian approximate posterior $q(\theta)$ parameterised by a mean vector μ_t and precision matrix $\mathbf{\Lambda}_t$, collectively denoted as $\psi_t = \{\mu_t, \mathbf{\Lambda}_t\}$:

Update step In the projection step, the normaliser of the posterior in Equation 5.18 will never be needed; hence this term will be ignored for any practical purposes. The Gaussian approximate distribution q obtain from the previous step is equivalent to adding a quadratic penalty in the log domain, centred around its mean:

$$\begin{aligned} \log p(\theta|\mathcal{D}_{t+1}, \psi_t) &= \log p(\mathcal{D}_{t+1}|\theta) + \log q(\theta|\psi_t) + \textbf{const} \\ &= \log p(\mathcal{D}_{t+1}|\theta) - \frac{1}{2}(\theta - \mu_t)^\intercal \mathbf{\Lambda}_t (\theta - \mu_t) + \textbf{const}. \end{aligned} \tag{5.21}$$

Projection step In this step, we approximate the posterior using the Laplace approximation from Section 5.1. Firstly the mean of the approximate distribution is set to be equal to the mode of the new posterior:

$$\mu_{t+1} = \arg\max_\theta \log p(\mathcal{D}_{t+1}|\theta) + \log q(\theta|\psi_t). \tag{5.22}$$

The precision matrix of the new distribution is then computed as the Generalised Gauss-Newton matrix of the likelihood, evaluated at the mode μ_{t+1}, plus the precision of the previous approximate $q(\theta|\psi_t)$. This leads to the following recursive update:

$$\mathbf{\Lambda}_{t+1} = \overline{\mathbf{G}}_{t+1}(\mu_{t+1}) + \mathbf{\Lambda}_t. \tag{5.23}$$

The recursion is initialised with the Generalised Gauss-Newton of the log prior. Typically this is an isotropic Gaussian, in which case the curvature matrix is just a scaled identity. A similar recursive Laplace approximation for online learning has been recently discussed, however with limited experimental results and in the context of using a diagonal approximation to the Hessian [58]. It is worth emphasising, that sums of Kronecker products do not in general factorise, i.e. $\mathbf{A} \otimes \mathbf{B} + \mathbf{C} \otimes \mathbf{D} \neq (\mathbf{A} + \mathbf{C}) \otimes (\mathbf{B} + \mathbf{D})$ so it is not possible to simply add all factors together. In our implementation, an approximate curvature matrix is kept in memory for every task, similar to how EWC keeps the MAP parameters for each task [64]. This approximation will be termed "Online Laplace". If constant scaling in the number of tasks is required, one can make a further approximation by adding up the Kronecker factors separately. This is comparable to the independence assumption between the factors within the same task, and in general, will make the approximation less accurate.

A desirable property of the Laplace approximation is that the approximate posterior becomes peaked around its current mode as more data is observed. This becomes particularly clear if one considers the precision matrix as the product of the number of data points and the average precision. By becoming increasingly peaked, the approximate posterior will naturally allow the parameters to change less for later tasks, retaining the information about previous ones. At the same time, even though the Laplace method is a local approximation, it should leave sufficient flexibility for the parameters to adapt to new tasks, as the curvature of neural networks has been observed to be flat in many directions [118].

For comparative purposes we will also compare to fitting the true posterior with a new Gaussian on every new task, where the Generalised Gauss-Newton matrix of all tasks is computed at the most recent MAP estimate:

$$\mathbf{\Lambda}_{t+1} = \overline{\mathbf{G}}_{\text{prior}} + \sum_{i=1}^{t+1} \overline{\mathbf{G}}_i(\mu_{t+1}). \tag{5.24}$$

Technically, this is not a valid Laplace approximation, as the parameter μ_{t+1} is not necessarily a mode of the true posterior, but only a mode of the filtering distribution, which uses $q_(\theta, \psi_t)$. Moreover, since this requires computing the Generalised Gauss-Newton on all datasets, this procedure violates the sequential learning setting as it requires access to previous tasks' data and requires significantly more computation. However, the main goal of doing this is to gain insights into how much curvature information our iterative method looses, compared to the "oracle" which can com-

pute the curvature matrices on all of the data. In general, this is always expected to perform better, but the hope is that the drop in performance is not significant. This approximation will be termed "Non-Online Laplace".

Regularising the approximate posterior

A similar online method developed in [64] suggests using a multiplier λ on the quadratic penalty in Equation 5.21. This provides a way of trading off between retaining information from previous tasks and having sufficient flexibility in learning new ones. As modifying the objective function directly will propagate through the recursive relationship for the precision matrix, we propose to place the multiplier on the new task's Generalised Gauss-Newton matrix when updating the curvature matrix. This corresponds to modifying Equation 5.23 to:

$$\mathbf{\Lambda}_{t+1} = \lambda \overline{\mathbf{G}}_{t+1}(\mu_{t+1}) + \mathbf{\Lambda}_t. \tag{5.25}$$

Notably, since the multiplier has a direct effect on the magnitude of the approximate posterior, used as a prior in the next task, it will affect all future MAP estimates μ. The optimal value of λ can potentially inform us about the quality of our Gaussian approximation. If it strongly deviates from its "natural" value of 1 this would indicate a poor approximation to the posterior and most likely either over or underestimates the uncertainty about the parameters. A toy example of this is visualised in Figure 5.6 where both the likelihood and the prior are Gaussian. Values less than 1 shift the joint maximum towards that of the likelihood, i.e. the new task, while values greater than 1 it moves towards the prior, i.e. previous tasks. In principle, it is possible to use a different value of λ_t for every new observed task. This, however, would make the number of hyperparameters to grow linearly in the number of tasks, which would make tuning very costly. For this reason and to keep its interpretation as a way of measuring the goodness of our approximation, its value is kept the same across all tasks.

Computational complexity

Excluding the cost of finding the MAP estimate μ_t for every task, our method further requires computing the two Kronecker factors of the approximate Laplace distribution. Computing the factors can be done efficiently via minibatch sampling and requires the same calculation as a forward and a backward pass through the network plus two additional matrix-matrix products per layer. Thus the overhead

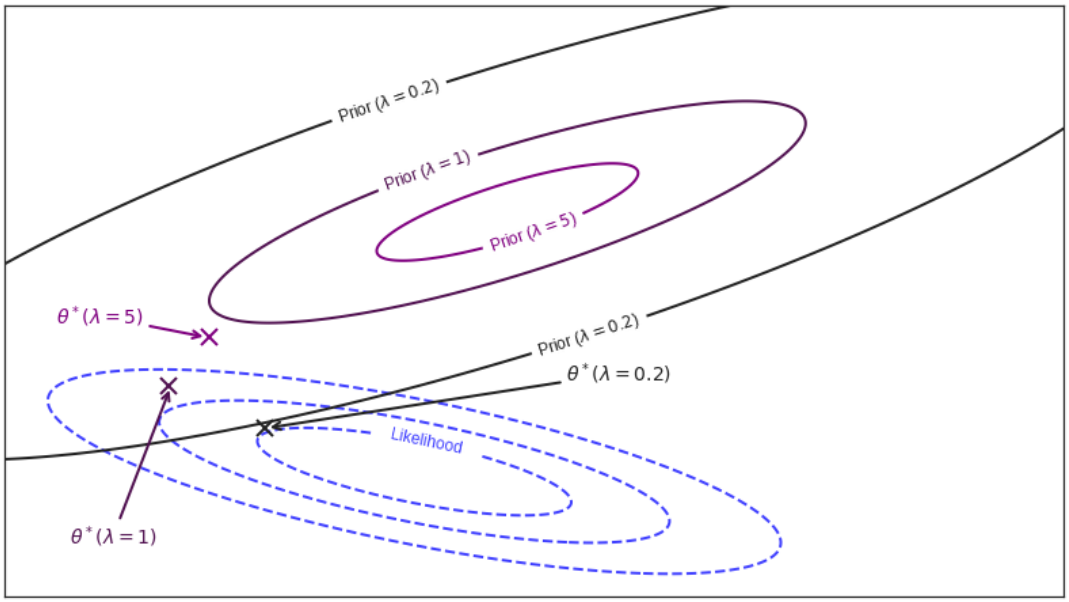


Figure 5.6: Effect of the value of λ on the MAP estimate. The prior distribution contour lines are marked with straight lines, while the likelihood with dashed lines. The MAP estimate for different values λ is marked by $\times$.

is roughly equivalent to an extra training epoch. Additionally, since we are using the Kronecker factored Gaussian as a prior, there is some additional cost during training in calculating its gradient with respect to the parameters. For every layer, this corresponds to two matrix-matrix products. Hence, if we assume that each layer has a weight matrix of size $d \times d$ for L layers, the additional computation is $\mathcal{O}(Ld^3)$.

5.3.2 Alternative methods

So far we have described the Bayesian approach for solving online learning, together with a practical proposal for approximating it. However, there are other methods in the literature that have been developed for this, which will be used in the next section for comparison. The first method that will be presented is Elastic Weight Consolidation (EWC) [64]. Originally this approach was also motivated by the work of [79] for constructing an iterative Laplace approximation, as described already in this chapter. However, instead of forming an approximate posterior that is Gaussian around the last discovered optimal parameters, EWC builds an approximate posterior that is a mixture of Gaussians, around the parameters that were found optimal at the end of each previous task. In the notation of Equation 5.21 the objective that

is being maximised when finding the next mean parameter μ_{t+1} is

$$\log p(\mathcal{D}_{t+1}|\theta) - \sum_{i=1}^{t} \frac{1}{2}(\theta - \mu_i)^\intercal \overline{\mathbf{F}}_i(\mu_i)(\theta - \mu_i). \tag{5.26}$$

When dealing dealing with more than two tasks, this is inconsistent with the Bayesian online learning framework and leads to over counting data — specifically more weight would be given to earlier tasks [58]. In addition, the authors make only a diagonal approximation to the Fisher matrix. Although this method is not compatible with the Bayesian framework, note that it is still possible to apply EWC with a richer curvature approximation, such as the Kronecker factored one that we have proposed.

The second method that will be presented is Synaptic Intelligence [141]. The method is similar in spirit to EWC, in the sense that on every new task it introduces a quadratic penalty to all of the previous tasks final parameters:

$$\log p(\mathcal{D}_{t+1}|\theta) - \sum_{i=1}^{t} \frac{1}{2}(\theta - \mu_i)^\intercal \Omega_i(\mu_i)(\theta - \mu_i). \tag{5.27}$$

However, rather than using the Fisher, or any other curvature approximation, the authors propose to weight each component of the quadratic losses, by regularisation value that depends on two quantities — the "importance" ω of that parameter for the given task, and the amount of change it exhibit during training on that task:

$$\begin{aligned} \omega_t &= \sum_{k=0}^{K} \nabla_\theta \mathcal{L}_k(\theta_t^k) \circ \triangle\theta_t^k, \\ d_t &= \theta_t^K - \theta_t^0, \\ [\Omega]_{i,j} &= \lambda \delta_j^i \frac{[\omega_t]_i}{[d_t]_i^2 + \epsilon}. \end{aligned} \tag{5.28}$$

The index k is over the number of optimization steps that are performed during training on task t and ϵ is a small constant added for avoiding any numerical issues. The parameter λ is a hyper-parameter that similar to the Laplace approximation trades-off the weight of previous tasks to that of the current one. The main, argument for squaring d_t in the denominator is to make it scale the same as that of ω_t. This approach seems to have little theoretical justification and is much more based on a human heuristic of determining which parameters are important. In addition, the formulation by construction makes Ω (what can be thought of as the precision of a Gaussian approximation) diagonal and it is not possible to apply this method with any non-diagonal matrix.

5.4 Experiments on online learning

In the following series of experiments our online Laplace approximation is compared to Elastic Weight Consolidation (EWC) and Synaptic Intelligence (SI) (see Section 5.3.2). In addition, a diagonal Laplace approximation is also evaluated in order to demonstrate the benefits of using richer curvature estimates. The "Non-Online Laplace" Laplace from the previous section is included as well, with the goal to measure the loss of curvature information due to the online setting of the problem. In order to investigate whether the updates from EWC are indeed worse compared to the Bayesian online learning framework, as originally raised in [58], EWC is also run with a Kronecker factored approximation. This would be termed "EWC Laplace", which corresponds to the original method when it is diagonal. As a reminder, the standard Laplace approximation from Section 5.3.1 is denoted as "Online Laplace", the one which allowed to look at all previous data as "Non-Online Laplace". The dataset being used in the experiments are:

MNIST the same as described earlier in Section 4.5.

notMNIST the same as described earlier in Section 5.2.

Fashion MNIST consists of $60,000$ training and $10,000$ test examples [133]. Each image is a 28×28 grayscale image, associated with a label from 10 classes. Its main intention is to mimic the image format of MNIST, but contains a collection of totally different objects, in this case several fashion item categories.

Permuted MNIST consists of 50 independent tasks, each one being represented as correctly classifying the MNIST dataset images under a fixed permutation of the image pixels. This makes the individual data distributions mostly independent of each other, testing the ability of each method to fully utilise the model's capacity.

Disjoint MNIST splits the original MNIST dataset in to two tasks, one containing the digits 0 to 4 and one containing the digits 5 to 9. Both tasks are intended to be trained as a ten-way classifier, with the goal of investigating the capabilities of models to retain information when trained on two fully disjoint set of objects.

CIFAR10 is a similar dataset to CIFAR100, described in Section 5.2, but with 10 more coarser object classes.

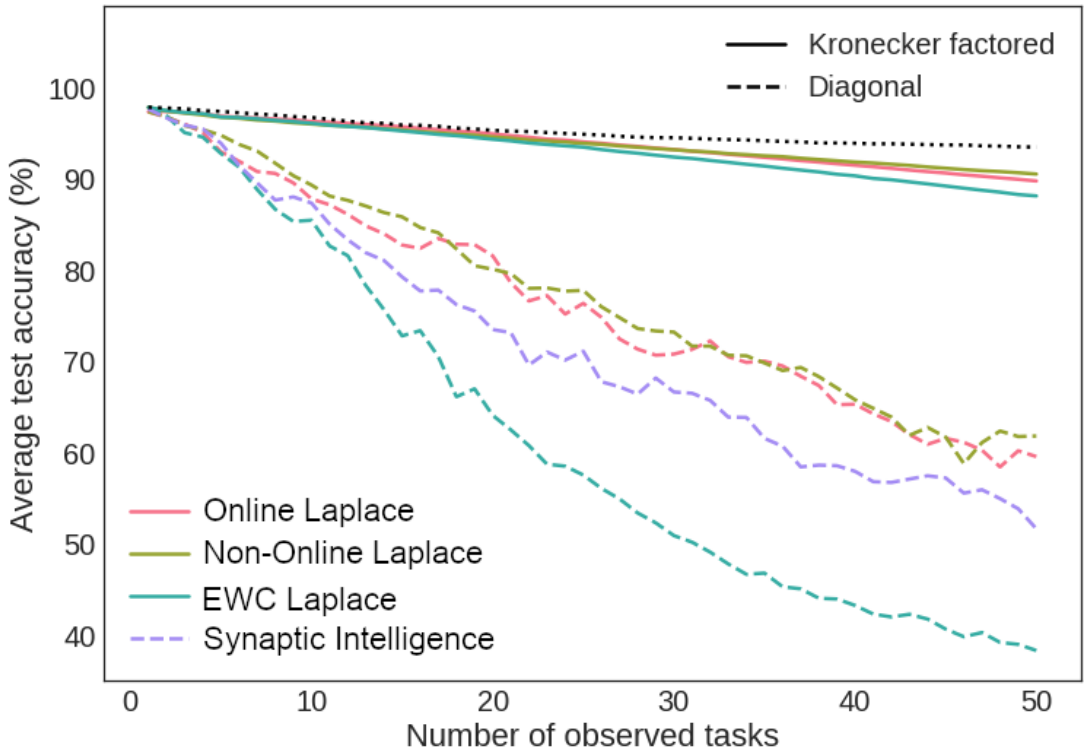


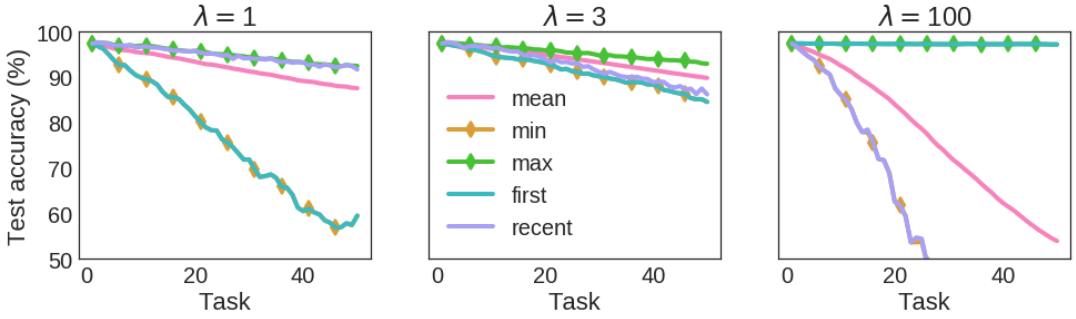
Figure 5.7: Online learning on permuted MNIST. The plot shows the mean test accuracy on a sequence of 50 permuted MNIST tasks. The dotted black line shows the performance of a single network trained on all observed data up to task k and can be considered as the best possible performance.

SVHN consists of over $600,000$ examples of 32×32 colour images of street view house numbers [96]. Each image represents a single digit from '0' to '9' , hence there are 10 labels analogously to MNIST.

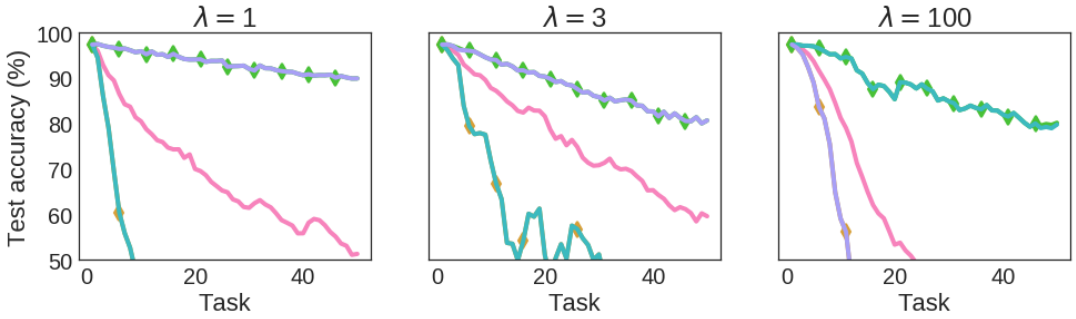
All experiments are implemented using Theano and Lasagne [112, 27].

5.4.1 Online learning on Permuted MNIST

The first experiment we conduct compares all different methods on the Permuted MNIST dataset. The model that is being trained is a Feed Forward Neural Network with two hidden layers of 100 units and a Rectifier Linear Unit activation functions. The network is chosen to be smaller than in previous work with the goal to make each task more challenging. In practice, we found that the performance of different methods was mildly dependent on the choice of the optimiser in use. Therefore, we optimize all techniques with Adam [61] for 20 epochs per dataset and a learning



(a) Kronecker factored



(b) Diagonal

Figure 5.8: Effect of λ for different approximations. Each plot shows the mean, minimum and maximum classification accuracy across the tasks observed so far, as well as the accuracy on the first and most recent task.

rate of 10^{-3} as in [141], SGD with Momentum [104] with an initial learning rate of 10^{-2} and 0.95 momentum, and Nesterov's Accelerated Gradient [95] with an initial learning rate of 0.1, which we divide by 10 every 5 epochs, and 0.9 momentum. For the momentum-based methods, we train for at least 10 epochs and early-stop once the validation error does not improve for 5 epochs. Furthermore, we decay the initial learning rate with a factor of $\frac{1}{1+kt}$ for the momentum-based optimisers, where t is the index of the task and k a decay constant. We set k using a coarse grid search for each value of the hyperparameter λ in order to prevent the objective from diverging towards the end of training, in particular with the Kronecker factored curvature approximation. For the Laplace approximation based methods, we consider $\lambda \in \{1, 3, 10, 30, 100\}$; for SI we try $c \in \{0.01, 0.03, 0.1, 0.3, 1\}$. We ultimately pick the combination of optimiser, hyperparameter and decay rate that gives the best validation error across all tasks at the end of training. For the Laplace-based methods, we found momentum-based optimisers to lead to better performance, whereas Adam gave better results for SI.

Figure 5.7 shows the mean test accuracy as new tasks are observed for the optimal hyperparameters of each method. We find our "Online Laplace" approximation to maintain higher test accuracy throughout training than placing a quadratic penalty around the MAP parameters of every task, in particular when using a simple diagonal approximation to the Generalised Gauss-Newton. However, the main difference between the methods lies in using a Kronecker factored approximation of the curvature over a diagonal one. Using this approximation, we achieve over 90% average test accuracy across 50 tasks, almost matching the performance of a network trained jointly on all observed data. The "Non-Online Laplace" method which recalculates the curvature for each task instead of retaining previous estimates does not significantly improve performance.

We also investigate the effects of different values of the hyperparameter λ on the performance of the "Online Laplace" approximation. The main goal of this experiment is to visualise and understand better the effective trade-off between remembering previous tasks and being able to learn new ones. The comparison is performed for the Kronecker factored as well as the diagonal approximations for completeness. Figure 5.8 shows different statistics of the accuracy on the test set for three different values of the hyperparameter. The exact values have been selected as the smallest, the largest and the one that achieves the best performance on the validation set. The performance on the first and the most recent task is of particular interest as they measure most accurately the trade-off between memorisation and flexibility. For all displayed values of the hyperparameter, the Kronecker factored approximation has significantly higher test accuracy than the diagonal approximation on both the most recent and the first task, as well as on average. For the natural choice of $\lambda = 1$ the network performance on the first task decays substantially. Using $\lambda = 3$ the Kronecker factored Laplace approximation the network has a much smaller gap between the performance on the first and more recent task, which is not possible with the diagonal approximation. Finally, when $\lambda = 100$ the order of different performance curves reverses, and the network almost fully memorises the first task.

In conclusion, the results show that the Kronecker factored curvature approximation brings significant benefits across all methods. In addition, we validate that the principled Bayesian framework updates achieve better performance than the one suggested in [64]. Finally, even with a better curvature approximation the Laplace approximation seems to overestimate the uncertainty in the parameters,

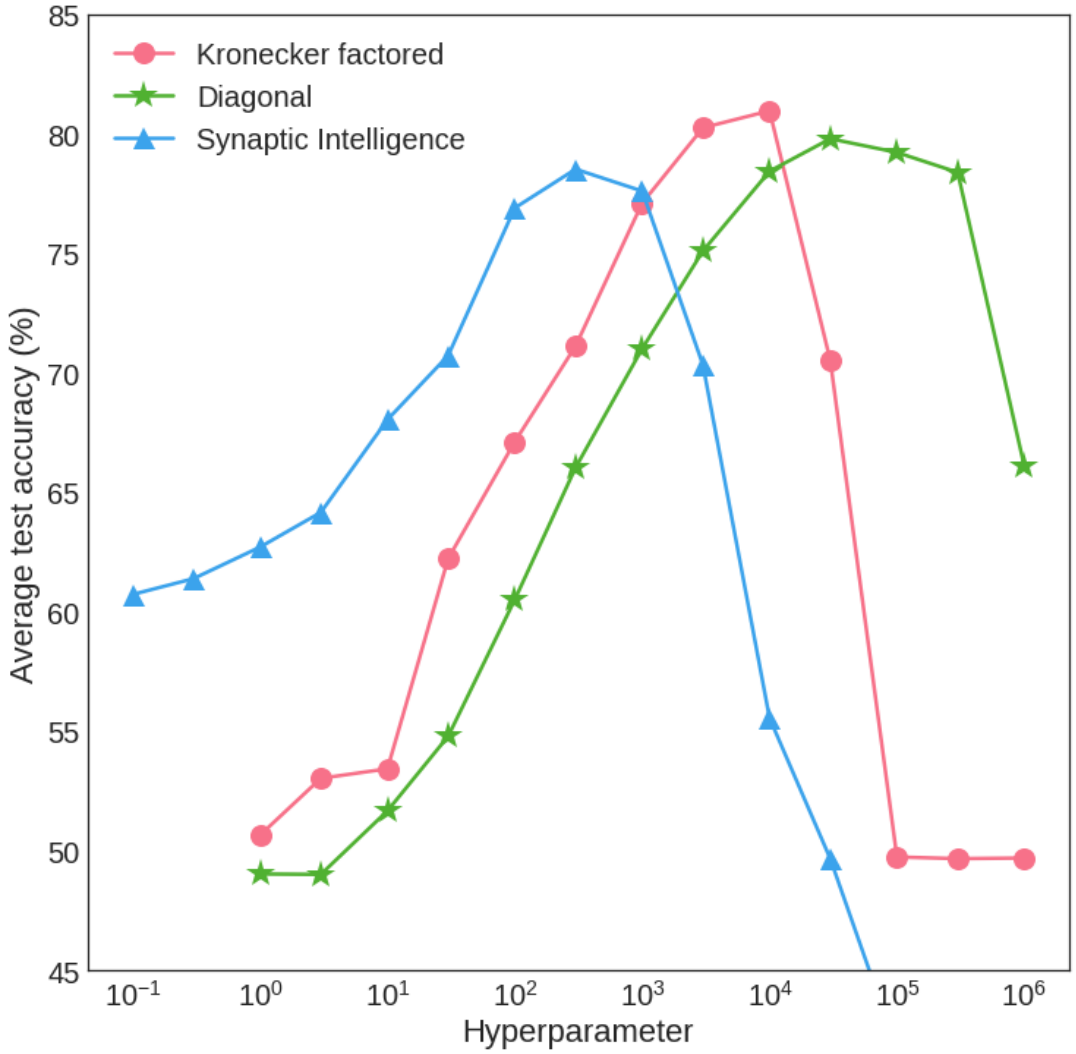


Figure 5.9: Online learning on Disjoint MNIST. The plot shows the test accuracy of each method for different values of the hyperparameter λ for the Laplace approximations and c for Synaptic Intelligence.

which makes the optimal choice of the hyperparameter λ deviate from its natural value of 1. Instead for this particular sequence of tasks, the optimal value, based on optimising validation set performance, turns out to be $\lambda = 3$.

5.4.2 Online learning on Disjoint MNIST

The second experiment we compare our Kronecker factored approximation, its diagonal variant and Synaptic Intelligence on the more realistic Disjoint MNIST dataset. Previous work has found this problem to be challenging for EWC, as during the first

half of training the network is encouraged to set the bias terms for the second set of labels to highly negative values [72]. This setup makes it difficult to balance out the biases for the two sets of classes when training on the second task without overcompensating by making the biases for the first set of classes to highly negative values. Previous work reports just over 50% test accuracy for EWC, which corresponds to either completely forgetting the first task or being unable to learn the second one, as each task individually can be solved with around 99% accuracy [72]. Since there only two tasks, all of the three variations of the three methods "Online Laplace", "Non-Online Laplace" and "EWC Laplace" are identical. Consequently, we focus only on comparing the different curvature approximations — Kronecker factored and diagonal.

The neural network model is identical to the network from the previous section. However, we found that in this problem, both the Laplace approximation and Synaptic Intelligence required significantly higher values of their corresponding regularisation hyperparameter. For the Laplace method, we tested values of $\lambda \in \{1, 3, 10, \ldots, 3{\times}10^5, 10^6\}$, and for Synaptic Intelligence values of $c \in \{0.1, 0.3, 1, \ldots, 3{\times}10^4, 10^5\}$. The training was done using Nesterov Accelerated Gradient with a learning rate of 0.1 and momentum of 0.9 using a minibatch size of 250. The learning rate was decayed by a factor of 10 every 1000 parameter updates. The initial learning rate for the second task is further decayed depending on the corresponding hyperparameter value to prevent the objective from diverging. We experimented with various decay factors but found the simple rule of using $\frac{\lambda}{10}$ for the Kronecker factored approximation and $\frac{\lambda}{1000}$ for the diagonal approximation to work well. Each run was repeated ten times, and the results are averaged across ten independent runs.

Figure 5.9 shows the test accuracy for different values of the hyperparameters of each method. The results have been averaged over ten independent runs. We did not manage to match the performance of the method as reported in [72] and found the Laplace approximation to work significantly better. The Kronecker factored approximation gives a small improvement over the diagonal one and requires weaker regularisation, which further suggests that it better fits the posterior distribution. Both of the Laplace methods outperform Synaptic Intelligence for the optimal value of their corresponding hyperparameter.

	Test Error (%)					
Method	MNIST	nMNIST	fMNIST	SVHN	C10	Avg.
SI	87.27	79.12	84.61	77.44	57.61	77.21
EWC Laplace(D)	97.83	94.73	89.13	79.80	53.29	82.96
EWC Laplace(KF)	97.85	94.92	89.31	85.75	58.78	85.32
Online Laplace(D)	96.48	93.41	88.09	81.79	53.80	82.71
Online Laplace(KF)	97.17	94.78	90.36	85.59	59.11	85.40
Non-Online Laplace(D)	96.56	92.33	89.27	78.00	56.57	82.55
Non-Online Laplace(KF)	97.90	94.88	90.08	85.24	58.63	85.35

Table 5.3: Final test accuracy for sequential vision tasks. The bracket (D) indicates a diagonal approximation, while (KF) a Kronecker factored.

5.4.3 Online learning on multiple datasets

The final experiments test our method on a suite of related vision datasets. The goal of this online learning problem is to train a classifier on the following sequence of datasets — MNIST, notMNIST, Fashion MNIST, SVHN and CIFAR10. All five datasets contain around $50,000$ training images from 10 different classes, however each of the 10 classes are distinct. Any dataset whose images are smaller that 32×32 are zero padded, and any grey-scale images have their intensities replicated over the three colour channels, such that all datasets have the same image format.

The neural network used for training is a LeNet-like architecture with two convolutional layers [69]. The first layer has a bank of 20 5×5 convolutional filters while the second layer has 50 convolutional filters with the same spatial size. After each convolution a 2×2 max-pooling with stride of 2 is applied. Thereafter, the network has a fully connected hidden layer with 500 before the final layer. As the meaning of the classes in each dataset is different, we keep the weights of the final layer separate for each task. The activation function of all layers is a Rectifier Linear unit. For the convolutional layers we use the extension of the Kronecker factored curvature approximations to convolutions from [44]. The network is optimised in the same was as in Section 5.4.1. For measuring the optimal possible classification performance we train five baseline networks, with the same architecture, each trained separately on each task.

Figure 5.10 shows the test accuracy of the "Online Laplace" approximation, both diagonal and Kronecker factored, compared to Synaptic Intelligence on every

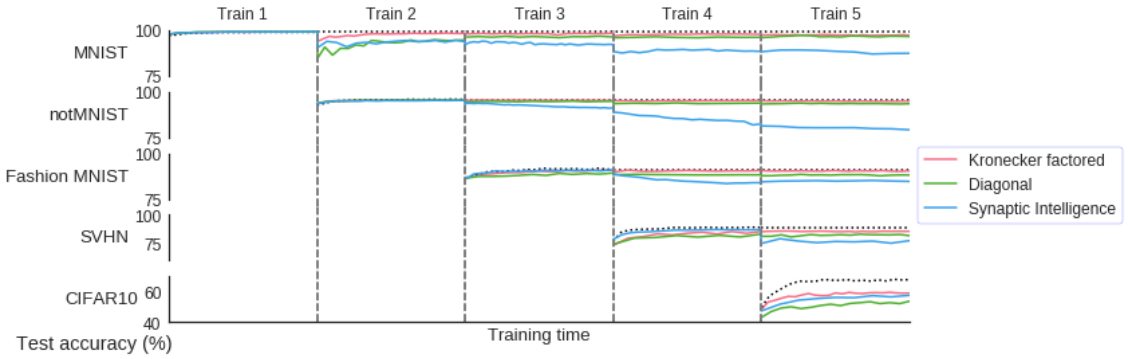


Figure 5.10: Online learning experiments using "Online Laplace". The plot shows the test accuracy of a convolutional network on every dataset as training progress sequentially through the five datasets MNIST, notMNIST, Fashion MNIST, SVHN and CIFAR10. The dotted black lines indicates the performance of the baseline networks with the same architecture trained on each task separately.

dataset as they progress sequentially through the five tasks. Similarly Figure 5.11 and Figure 5.12 show the "Non-Online Laplace" and "EWC Laplace" results respectively. In addition to help the reader the final test accuracy on each task are shown in Table 5.3 for all methods. Overall, the "Online Laplace" approximation in conjunction with a Kronecker factored approximation of the curvature achieves the highest test accuracy across all five tasks. However, the difference between the three Laplace-based methods is small in comparison to the improvement stemming from the better approximation to the Generalised Gauss-Newton matrix. Using a diagonal approximation for the Laplace approximation, the network mostly remembers the first three tasks, but has difficulties learning the fifth one. SI, in contrast, shows decaying performance on the initial tasks, but learns the fifth task almost as well as our method with a Kronecker factored approximation. However, using the Kronecker factored approximation, the network achieves good performance relative to the individual networks across all five tasks. In particular, it remembers the easier early tasks almost perfectly while being sufficiently flexible to learn the more difficult later tasks better than the diagonal methods, which suffer from forgetting. In conclusion, these experiments suggest that in larger models, the main difference is driven by the more richer curvature approximation, rather then the exact method applied.

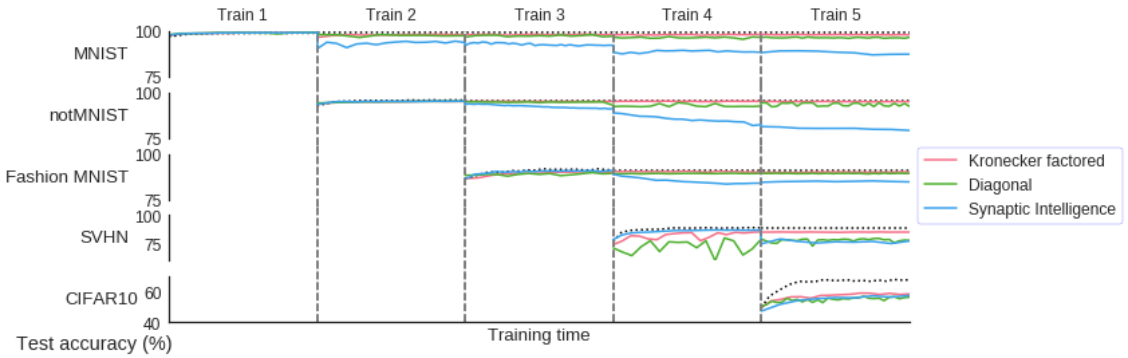


Figure 5.11: Online learning experiments using "Non-Online Laplace". The plot shows the test accuracy of a convolutional network on every dataset as training progress sequentially through the five datasets MNIST, notMNIST, Fashion MNIST, SVHN and CIFAR10. The dotted black lines indicates the performance of the baseline networks with the same architecture trained on each task separately.

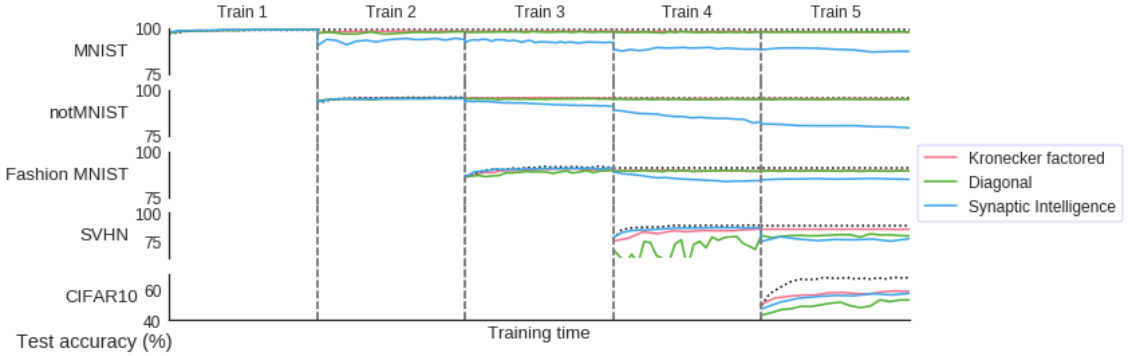


Figure 5.12: Online learning experiments using "EWC Laplace". The plot shows the test accuracy of a convolutional network on every dataset as training progress sequentially through the five datasets MNIST, notMNIST, Fashion MNIST, SVHN and CIFAR10. The dotted black lines indicates the performance of the baseline networks with the same architecture trained on each task separately.

Chapter 6

Conclusion and future research directions

This work presents an insightful theoretical analysis of the structure of the Generalised Gauss-Newton matrix for Feed Forward Neural Networks. Using the results from the analysis as a motivating basis, an efficient block-diagonal approximation is proposed. The resulting curvature approximation achieves competitive performance against state-of-the-art first-order methods when applied to optimisation. Unlike standard optimisers, which require significant hyperparameter tunining, our approach provides good performance with the default hyperparameter values. After that, using the curvature approximation, we propose to construct a Laplace approximation to the posterior distribution over the weights of a trained model. Experimentally we demonstrate that the resulting method leads to better uncertainty estimates on out-of-distribution data and is more robust to simple adversarial attacks on state-of-the-art convolutional network architecture. Finally, we adapt the Laplace approximation into a Bayesian online learning framework, where we recursively approximate the posterior after every task with a Gaussian. The algorithm substantially outperforms related methods on the task of overcoming catastrophic forgetting.

Most of the modern architectures and their initialisation have been optimised to perform well under training specifically using first-order optimisation. Nevertheless, as shown in [85], a second-order method is capable of training models to be successful in tasks, that first-order methods struggle to succeed. A natural question arises — is it possible that the community has not discovered useful models, just because they are trainable only with a better optimiser? The work presented here together

with previously published results on scalable curvature approximations for neural networks [84, 44, 6, 83] make a step towards answering these questions. Additionally, the optimisation method used in Chapter 4 does not need any hyperparameter tunning, a benefit similar to LBFGS. One drawback is that in practice, the Gauss-Newton matrix for the output of the network has to be computed explicitly. For standard losses, as demonstrated, this is not an issue, but as more interesting models are presented in the literature, this could be problematic for wider adoption. Below are listed several avenues for future research on the topic of curvature approximations for Deep Learning models:

Scalable approximations for more layers This work presented only an approximation to the diagonal blocks of the Generalised Gauss-Newton matrix for fully connected layers. However, in practice, there are many different parametric layers, such as convolutional layers and self-attention layers. Also, for recurrent models, due to the repeated applications of the same weights, this further complicates the curvature approximations [83]. Recent work extends the KFAC approximation to convolutions and can be applied to the Gauss-Newton, but for the method to be more widely applicable all layers used in the literature should have such approximations [44].

More flexible block approximations Although the Kronecker factored approximations perform significantly better than diagonal, there is no reason not construct even richer approximations. The original Kronecker factor method can be modified by correcting its eigenvalues to be fully independent [37]. It could also be possible to construct a combination of rank-1 plus a Kronecker factored matrix, such that it captures better the Gauss-Newton of each block. This can have further implications in providing more insights to the curvature of the loss surfaces of Deep Learning models.

Automatic batch size scheduler One of the major advantages of first-order optimisers is their simplicity and in the straightforward way that they deal with stochastic gradients. Although our method worked well with reasonably sized batch sizes, in very small batch sizes the benefit of the curvature diminishes as its estimates will become too stochastic as well. In light of this, one of the hyperparameters of the method is what batch size to use? Too large and the method will become too computationally expensive to be practical, too small, and it might lose any benefits

of estimating the curvature. Figuring out what is the optimal batch size in an online fashion, such that it adapts to the objective at the current parameters, would make the method truly hyperparameter free. This would potentially give a good incentive to the community for its wider adoption.

Further research into applications beyond function minimisation The thesis presented two avenues where the curvature approximations provided significant benefits compared to previous methods. There are many other applications, where further research is required to understand better how these approximations can be used adequately. For instance, in competitive games, which include the dynamics of GANs, are an instance of min-max optimisation. In the literature so far, there have only been analysed methods that use first-order steps for both players [89, 53, 93, 7]. Second-order could potentially be useful in these models, but this would require more than just a straightforward plugging in of their updates. Another interesting direction is to try to apply these curvature matrix approximations for Reinforcement Learning. This can be done either for better policy improvements or using the Laplace approximations for better exploration. In addition, meta-learning methods like MAML could make use of better optimisation steps as well [30].

Bibliography

[1] Akkaya, I. et al. “Solving Rubik’s Cube with a Robot Hand”. In: *ArXiv* (2019) (cit. on p. 6).

[2] Amari, S.-i. “Natural Gradient Works Efficiently in Learning”. In: *Neural Computation* 10.2 (1998), pp. 251–276 (cit. on pp. 2, 38).

[3] Anthony, T., Tian, Z., and Barber, D. “Thinking Fast and Slow with Deep Learning and Tree Search”. In: *Advances in Neural Information Processing Systems 30*. 2017, pp. 5360–5370 (cit. on p. 6).

[4] Armijo, L. “Minimization of functions having Lipschitz continuous first partial derivatives”. In: *Pacific Journal of mathematics* 16.1 (1966), pp. 1–3 (cit. on p. 17).

[5] Atanov, A., Ashukha, A., Molchanov, D., Neklyudov, K., and Vetrov, D. “Uncertainty Estimation via Stochastic Batch Normalization”. In: *International Symposium on Neural Networks*. 2019, pp. 261–269 (cit. on p. 18).

[6] Ba, J., Grosse, R., and Martens, J. “Distributed Second-Order Optimization using Kronecker-Factored Approximations”. In: *5th International Conference on Learning Representations*. 2017 (cit. on pp. 42, 94).

[7] Balduzzi, D., Racaniere, S., Martens, J., Foerster, J., Tuyls, K., and Graepel, T. “The Mechanics of n-Player Differentiable Games”. In: *Proceedings of the 35th International Conference on Machine Learning*. Vol. 80. Proceedings of Machine Learning Research. Stockholmsmässan, Stockholm Sweden, 2018, pp. 354–363 (cit. on p. 95).

[8] Blundell, C., Cornebise, J., Kavukcuoglu, K., and Wierstra, D. “Weight Uncertainty in Neural Network”. In: *Proceedings of the 32nd International Conference on Machine Learning*. Vol. 37. Proceedings of Machine Learning Research. Lille, France, 2015, pp. 1613–1622 (cit. on pp. 62, 70).

[9] Botev, A., Lever, G., and Barber, D. "Nesterov's accelerated gradient and momentum as approximations to regularised update descent". In: *2017 International Joint Conference on Neural Networks (IJCNN)*. 2017, pp. 1899–1903 (cit. on p. 4).

[10] Botev, A., Ritter, H., and Barber, D. "Practical Gauss-Newton Optimisation for Deep Learning". In: *Proceedings of the 34th International Conference on Machine Learning*. Vol. 70. Proceedings of Machine Learning Research. International Convention Centre, Sydney, Australia, 2017, pp. 557–565 (cit. on pp. 4, 20, 46).

[11] Botev, A., Zheng, B., and Barber, D. "Complementary Sum Sampling for Likelihood Approximation in Large Scale Classification". In: *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*. Vol. 54. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA, 2017, pp. 1030–1038 (cit. on p. 4).

[12] Boyd, S. and Vandenberghe, L. *Convex Optimization*. 2004 (cit. on p. 17).

[13] Bulatov, Y. *notMNIST dataset*. http://yaroslavvb.blogspot.com/2011/09/notmnist-dataset.html. 2011 (cit. on p. 68).

[14] Byrd, R. H., Hansen, S. L., Nocedal, J., and Singer, Y. "A Stochastic Quasi-Newton Method for Large-Scale Optimization". In: *SIAM Journal on Optimization* 26.2 (2016), pp. 1008–1031 (cit. on p. 17).

[15] Cauchy, A. "Méthode générale pour la résolution des systemes d'équations simultanées". In: *Comp. Rend. Sci. Paris* 25.1847 (1847), pp. 536–538 (cit. on p. 14).

[16] Chapelle, O. and Li, L. "An Empirical Evaluation of Thompson Sampling". In: *Advances in Neural Information Processing Systems 24*. 2011, pp. 2249–2257 (cit. on p. 19).

[17] Chen, H., Liu, X., Yin, D., and Tang, J. "A Survey on Dialogue Systems: Recent Advances and New Frontiers". In: *SIGKDD Explor. Newsl.* 19.2 (Nov. 2017), pp. 25–35 (cit. on p. 6).

[18] Chen, P. "Hessian Matrix vs. Gauss—Newton Hessian Matrix". In: *SIAM Journal on Numerical Analysis* 49.4 (2011), pp. 1417–1435 (cit. on p. 34).

[19] Cho, K., Van Merriënboer, B., Bahdanau, D., and Bengio, Y. "On the Properties of Neural Machine Translation: Encoder-Decoder Approaches". In: *Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation (SSST-8), 2014*. 2014 (cit. on p. 12).

[20] Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., and LeCun, Y. "The Loss Surfaces of Multilayer Networks". In: ed. by G. Lebanon and S. V. N. Vishwanathan. Vol. 38. Proceedings of Machine Learning Research. San Diego, California, USA: PMLR, May 2015, pp. 192–204 (cit. on p. 1).

[21] Clevert, D.-A., Unterthiner, T., and Hochreiter, S. "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)". In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. 2016 (cit. on p. 8).

[22] Cramér, H. "A contribution to the theory of statistical estimation". In: *Scandinavian Actuarial Journal* 1946.1 (1946), pp. 85–94 (cit. on p. 2).

[23] Csáji, B. C. et al. "Approximation with Artificial Neural Networks". In: *Faculty of Sciences, Etvs Lornd University, Hungary* 24.48 (2001), p. 7 (cit. on pp. 8, 9).

[24] Daniels, H. "The Asymptotic Efficiency of a Maximum Likelihood Estimator". In: *Fourth Berkeley Symposium on Mathematical Statistics and Probability*. Vol. 1. 1961, pp. 151–163 (cit. on p. 12).

[25] Deng, L. and Platt, J. "Ensemble Deep Learning for Speech Recognition". In: *Fifteenth Annual Conference of the International Speech Communication Association*. Sept. 2014 (cit. on p. 19).

[26] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *CoRR* abs/1810.04805 (2018). arXiv: 1810.04805 (cit. on pp. 6, 12).

[27] Dieleman, S. et al. *Lasagne: First release.* 2015 (cit. on pp. 41, 53, 68, 85).

[28] Dietterich, T. G. "Ensemble Methods in Machine Learning". In: *International Workshop on Multiple Classifier Systems*. 2000, pp. 1–15 (cit. on p. 19).

[29] Eigen, D., Ranzato, M., and Sutskever, I. "Learning Factored Representations in a Deep Mixture of Experts". In: *arXiv preprint arXiv:1312.4314* (2013) (cit. on p. 60).

[30] Finn, C., Abbeel, P., and Levine, S. "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks". In: *Proceedings of the 34th International Conference on Machine Learning*. Vol. 70. Proceedings of Machine Learning Research. International Convention Centre, Sydney, Australia, 2017, pp. 1126–1135 (cit. on p. 95).

[31] French, R. M. "Catastrophic forgetting in connectionist networks". In: *Trends in Cognitive Sciences* 3 (1999), pp. 128–135 (cit. on p. 77).

[32] Fukushima, K. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological Cybernetics* 36.4 (Apr. 1980), pp. 193–202 (cit. on p. 10).

[33] Gal, Y. and Ghahramani, Z. "A Theoretically Grounded Application of Dropout in Recurrent Neural Networks". In: *Advances in Neural Information Processing Systems 29*. 2016, pp. 1019–1027 (cit. on p. 63).

[34] Gal, Y. and Ghahramani, Z. "Bayesian Convolutional Neural Networks with Bernoulli Approximate Variational Inference". In: *arXiv:1506.02158* (2015) (cit. on p. 63).

[35] Gal, Y. and Ghahramani, Z. "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning". In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*. ICML'16. New York, NY, USA, 2016, pp. 1050–1059 (cit. on pp. 18, 63, 67, 72).

[36] Gal, Y., Islam, R., and Ghahramani, Z. "Deep Bayesian Active Learning with Image Data". In: vol. 70. Proceedings of Machine Learning Research. International Convention Centre, Sydney, Australia, 2017, pp. 1183–1192 (cit. on p. 19).

[37] George, T., Laurent, C., Bouthillier, X., Ballas, N., and Vincent, P. "Fast Approximate Natural Gradient Descent in a Kronecker-Factored Eigenbasis". In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS'18. Montréal, Canada, 2018, pp. 9573–9583 (cit. on p. 94).

[38] Ghorbani, B., Krishnan, S., and Xiao, Y. "An Investigation into Neural Net Optimization via Hessian Eigenvalue Density". In: ed. by K. Chaudhuri and R. Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. Long Beach, California, USA: PMLR, June 2019, pp. 2232–2241 (cit. on p. 1).

[39] Ghosh, S., Fave, F. M. D., and Yedidia, J. "Assumed Density Filtering Methods for Learning Bayesian Neural Networks". In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI'16. Phoenix, Arizona, 2016, pp. 1589–1595 (cit. on p. 62).

[40] Glorot, X. and Bengio, Y. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy, 2010, pp. 249–256 (cit. on p. 53).

[41] Goodfellow, I. J., Mirza, M., Xiao, D., Courville, A., and Bengio, Y. "An Empirical Investigation of Catastrophic Forgetting in Gradient-Based Neural Networks". In: *arXiv e-prints*, arXiv:1312.6211 (2013), arXiv:1312.6211. arXiv: 1312.6211 [stat.ML] (cit. on p. 77).

[42] Goodfellow, I. J., Shlens, J., and Szegedy, C. "Explaining and Harnessing Adversarial Examples". In: *CoRR* abs/1412.6572 (2015) (cit. on p. 73).

[43] Graves, A. "Practical Variational Inference for Neural Networks". In: *Advances in Neural Information Processing Systems 24*. 2011, pp. 2348–2356 (cit. on pp. 62, 70).

[44] Grosse, R. and Martens, J. "A Kronecker-Factored Approximate Fisher Matrix for Convolution Layers". In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*. ICML'16. New York, NY, USA, 2016, pp. 573–582 (cit. on pp. 42, 75, 90, 94).

[45] Gupta, A. and Nagar, D. *Matrix Variate Distributions*. Monographs and Surveys in Pure and Applied Mathematics. 1999 (cit. on p. 66).

[46] Hahnloser, R. H., Sarpeshkar, R., Mahowald, M. A., Douglas, R. J., and Seung, H. S. "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit". In: *Nature* 405.6789 (2000), pp. 947–951 (cit. on p. 8).

[47] Hannun, A. et al. “Deep Speech: Scaling up end-to-end speech recognition”. In: *CoRR* abs/1412.5567 (2014). arXiv: 1412.5567 (cit. on p. 6).

[48] He, K., Zhang, X., Ren, S., and Sun, J. “Deep Residual Learning for Image Recognition”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778 (cit. on p. 1).

[49] He, K., Zhang, X., Ren, S., and Sun, J. “Deep Residual Learning for Image Recognition”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778 (cit. on pp. 10, 75).

[50] He, K., Zhang, X., Ren, S., and Sun, J. “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”. In: *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*. ICCV ’15. USA, 2015, pp. 1026–1034 (cit. on pp. 1, 32).

[51] He, K., Zhang, X., Ren, S., and Sun, J. “Identity Mappings in Deep Residual Networks”. In: *14th European Conference on Computer Vision*. Vol. 9908. 2016, pp. 630–645 (cit. on p. 75).

[52] Hernández-Lobato, J. M. and Adams, R. P. “Probabilistic Backpropagation for Scalable Learning of Bayesian Neural Networks”. In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. ICML’15. Lille, France, 2015, pp. 1861–1869 (cit. on pp. 62, 67, 68).

[53] Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., and Hochreiter, S. “GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium”. In: *Advances in Neural Information Processing Systems 30*. 2017, pp. 6626–6637 (cit. on p. 95).

[54] Hinton, G. E. and Salakhutdinov, R. R. “Reducing the Dimensionality of Data with Neural Networks”. In: *Science* 313.5786 (2006), pp. 504–507 (cit. on pp. 52, 53).

[55] Hochreiter, S. and Schmidhuber, J. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780 (cit. on p. 12).

[56] Houthooft, R. et al. “VIME: Variational Information Maximizing Exploration”. In: *Advances in Neural Information Processing Systems 29*. 2016, pp. 1109–1117 (cit. on p. 19).

[57] Hron, J., Matthews, A. G. d. G., and Ghahramani, Z. "Variational Gaussian dropout is not Bayesian". In: *ArXiv* (2017) (cit. on pp. 19, 63).

[58] Huszár, F. "Note on the quadratic penalties in elastic weight consolidation". In: *Proceedings of the National Academy of Sciences* 115.11 (2018), E2496–E2497 (cit. on pp. 80, 83, 84).

[59] Huzurbazar, V. "The likelihood equation, consistency and the maxima of the likelihood function". In: *Annals of Eugenics* 14.1 (1947), pp. 185–200 (cit. on p. 12).

[60] Kaufmann, E., Korda, N., and Munos, R. "Thompson Sampling: An Asymptotically Optimal Finite-Time Analysis". In: *International Conference on Algorithmic Learning Theory*. 2012, pp. 199–213 (cit. on p. 19).

[61] Kingma, D. P. and Ba, J. "Adam: A Method for Stochastic Optimization". In: *arXiv e-prints*, arXiv:1412.6980 (2014), arXiv:1412.6980. arXiv: 1412.6980 (cit. on pp. 1, 15, 53, 85).

[62] Kingma, D. P. and Welling, M. "Auto-Encoding Variational Bayes". In: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*. 2014 (cit. on pp. 18, 70).

[63] Kingma, D. P., Salimans, T., and Welling, M. "Variational Dropout and the Local Reparameterization Trick". In: *Advances in Neural Information Processing Systems 28*. 2015, pp. 2575–2583 (cit. on pp. 62, 63).

[64] Kirkpatrick, J. et al. "Overcoming catastrophic forgetting in neural networks". In: *Proceedings of the National Academy of Sciences* 114.13 (2017), pp. 3521–3526 (cit. on pp. 66, 77, 80–82, 87).

[65] Krizhevsky, A. "Learning Multiple Layers of Features from Tiny Images". In: *University of Toronto* (2009) (cit. on p. 68).

[66] Krizhevsky, A., Sutskever, I., and Hinton, G. E. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. 2012, pp. 1097–1105 (cit. on pp. 6, 10).

[67] Lakshminarayanan, B., Pritzel, A., and Blundell, C. "Simple and Scalable Predictive Uncertainty Estimation using Deep Ensembles". In: *Advances in Neural Information Processing Systems 30*. 2017, pp. 6402–6413 (cit. on pp. 19, 63).

[68] Laplace, P. S. de. "Memoir on the Probability of the Causes of Events". In: 1986 (cit. on p. 64).

[69] Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324 (cit. on pp. 52, 90).

[70] LeCun, Y., Bengio, Y., and Hinton, G. "Deep learning". In: *Nature* 521.7553 (2015), pp. 436–444 (cit. on p. 6).

[71] LeCun, Y., Denker, J. S., and Solla, S. A. "Optimal Brain Damage". In: *Advances in Neural Information Processing Systems 2*. 1990, pp. 598–605 (cit. on p. 66).

[72] Lee, S.-W., Kim, J.-H., Jun, J., Ha, J.-W., and Zhang, B.-T. "Overcoming Catastrophic Forgetting by Incremental Moment Matching". In: *Advances in Neural Information Processing Systems 30*. 2017, pp. 4652–4662 (cit. on p. 89).

[73] Lee, S., Purushwalkam, S., Cogswell, M., Crandall, D., and Batra, D. "Why M Heads are Better than One: Training a Diverse Ensemble of Deep Networks". In: *CoRR* abs/1511.06314 (2015). arXiv: 1511.06314 (cit. on p. 19).

[74] Li, Y. and Gal, Y. "Dropout Inference in Bayesian Neural Networks with Alpha-Divergences". In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70*. ICML'17. Sydney, NSW, Australia, 2017, pp. 2052–2061 (cit. on pp. 63, 73, 74).

[75] Liu, C., Zhu, J., and Song, Y. "Stochastic Gradient Geodesic MCMC Methods". In: *Advances in Neural Information Processing Systems*. Ed. by D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett. Vol. 29. Curran Associates, Inc., 2016, pp. 3009–3017 (cit. on p. 2).

[76] Loizou, N. and Richtárik, P. "Momentum and Stochastic Momentum for Stochastic Gradient, Newton, Proximal Point and Subspace Descent Methods". In: *ArXiv* (2017) (cit. on p. 15).

[77] Louizos, C. and Welling, M. "Multiplicative Normalizing Flows for Variational Bayesian Neural Networks". In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70*. ICML'17. Sydney, NSW, Australia, 2017, pp. 2218–2227 (cit. on p. 70).

[78] Louizos, C. and Welling, M. "Structured and Efficient Variational Deep Learning with Matrix Gaussian Posteriors". In: *Proceedings of The 33rd International Conference on Machine Learning*. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA, 2016, pp. 1708–1716 (cit. on p. 66).

[79] MacKay, D. J. C. "A Practical Bayesian Framework for Backpropagation Networks". In: *Neural Comput.* 4.3 (1992), pp. 448–472 (cit. on pp. 2, 63, 82).

[80] Mandt, S., Hoffman, M. D., and Blei, D. M. "A Variational Analysis of Stochastic Gradient Algorithms". In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*. ICML'16. New York, NY, USA, 2016, pp. 354–363 (cit. on p. 14).

[81] Martens, J. "Deep Learning via Hessian-Free Optimization". In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML'10. Haifa, Israel, 2010, pp. 735–742 (cit. on pp. 17, 47, 48, 56).

[82] Martens, J. "New perspectives on the Natural Gradient method". In: *CoRR* abs/1412.1193 (2014) (cit. on pp. 34, 38, 39, 50).

[83] Martens, J., Ba, J., and Johnson, M. "Kronecker-factored Curvature Approximations for Recurrent Neural Networks". In: *International Conference on Learning Representations*. 2018 (cit. on p. 94).

[84] Martens, J. and Grosse, R. "Optimizing Neural Networks with Kronecker-Factored Approximate Curvature". In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France, 2015, pp. 2408–2417 (cit. on pp. 42, 44, 47, 48, 52, 94).

[85] Martens, J. and Sutskever, I. "Learning Recurrent Neural Networks with Hessian-Free Optimization". In: *Proceedings of the 28th International Conference on International Conference on Machine Learning*. ICML'11. Bellevue, Washington, USA, 2011, pp. 1033–1040 (cit. on p. 93).

[86] Martens, J., Sutskever, I., and Swersky, K. "Estimating the Hessian by Back-Propagating Curvature". In: *Proceedings of the 29th International Coference on International Conference on Machine Learning*. ICML'12. 2012, pp. 963–970 (cit. on p. 46).

[87] Maybeck, P. "Stochastic Models, Estimation and Control". In: 1982. Chap. 12.7 (cit. on pp. 3, 78).

[88] McCloskey, M. and Cohen, N. J. "Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem". In: *Psychology of Learning and Motivation - Advances in Research and Theory* 24 (1989), pp. 109–165 (cit. on p. 77).

[89] Mescheder, L., Nowozin, S., and Geiger, A. "The Numerics of GANs". In: *Advances in Neural Information Processing Systems 30*. 2017, pp. 1825–1835 (cit. on p. 95).

[90] Mnih, V. et al. "Human-level control through deep reinforcement learning". In: *Nature* 518 (2015), pp. 529–533 (cit. on p. 6).

[91] Moritz, P., Nishihara, R., and Jordan, M. "A Linearly-Convergent Stochastic L-BFGS Algorithm". In: *Artificial Intelligence and Statistics*. 2016, pp. 249–258 (cit. on p. 17).

[92] Moulines, E. and Bach, F. R. "Non-Asymptotic Analysis of Stochastic Approximation Algorithms for Machine Learning". In: *Advances in Neural Information Processing Systems 24*. 2011, pp. 451–459 (cit. on p. 14).

[93] Nagarajan, V. and Kolter, J. Z. "Gradient descent GAN optimization is locally stable". In: *Advances in Neural Information Processing Systems 30*. 2017, pp. 5585–5595 (cit. on p. 95).

[94] Neal, R. M. "Bayesian Learning via Stochastic Dynamics". In: *Advances in Neural Information Processing Systems 5*. 1993, pp. 475–482 (cit. on p. 68).

[95] Nesterov, Y. "A Method of Solving a Convex Programming Problem with Convergence Rate O (1/k2)". In: *Soviet Mathematics Doklady* 27 (1983), pp. 372–376 (cit. on pp. 15, 86).

[96] Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. "Reading Digits in Natural Images with Unsupervised Feature Learning". In: *NIPS* (2011) (cit. on p. 85).

[97] Nocedal, J. "Updating Quasi-Newton Matrices with Limited Storage". In: *Mathematics of Computation* 35.151 (1980), pp. 773–782 (cit. on p. 17).

[98] Oord, A. van den, Kalchbrenner, N., Espeholt, L., k. kavukcuoglu koray, Vinyals, O., and Graves, A. "Conditional Image Generation with PixelCNN Decoders". In: *Advances in Neural Information Processing Systems 29*. 2016, pp. 4790–4798 (cit. on p. 6).

[99] Oord, A. van den and Schrauwen, B. "Factoring Variations in Natural Images with Deep Gaussian Mixture Models". In: *Advances in Neural Information Processing Systems 27*. 2014, pp. 3518–3526 (cit. on p. 60).

[100] Oord, A. v. d. et al. "WaveNet: A Generative Model for Raw Audio". In: *CoRR* abs/1609.03499 (2016). arXiv: 1609.03499 (cit. on p. 6).

[101] Opper, M. "A Bayesian Approach to On-Line Learning". In: *On-Line Learning in Neural Networks*. USA, 1999, pp. 363–378 (cit. on pp. 78, 79).

[102] Pearce, T., Leibfried, F., and Brintrup, A. "Uncertainty in Neural Networks: Approximately Bayesian Ensembling". In: *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*. Vol. 108. Proceedings of Machine Learning Research. Online, 2020, pp. 234–244 (cit. on p. 63).

[103] "Perceptrons Cambridge". In: *MA: MIT Press. zbMATH* (1969) (cit. on p. 8).

[104] Polyak, B. "Some methods of speeding up the convergence of iteration methods". In: *Ussr Computational Mathematics and Mathematical Physics* 4 (1964), pp. 1–17 (cit. on pp. 1, 15, 86).

[105] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. "Language Models are Unsupervised Multitask Learners". In: (2019) (cit. on pp. 6, 12).

[106] Rajpurkar, P., Zhang, J., Lopyrev, K., and Liang, P. "SQuAD: 100, 000+ Questions for Machine Comprehension of Text". In: *CoRR* abs/1606.05250 (2016). arXiv: 1606.05250 (cit. on p. 6).

[107] Ramachandran, P., Zoph, B., and Le, Q. V. "Searching for Activation Functions". In: *CoRR* abs/1710.05941 (2017). arXiv: 1710.05941 (cit. on p. 8).

[108] Ramezani-Kebrya, A., Khisti, A., and Liang, B. "On the Stability and Convergence of Stochastic Gradient Descent with Momentum". In: *CoRR* abs/1809.04564 (2018). arXiv: 1809.04564 (cit. on p. 15).

[109] Rao, C. R. *Information and accuracy attainable in the estimation of statistical parameters.* 1945 (cit. on p. 2).

[110] Ratcliff, R. "Connectionist Models of Recognition Memory: Constraints Imposed by Learning and Forgetting Functions". In: *Psychological Review* 97 (1990), pp. 285–308 (cit. on p. 77).

[111] Reddi, S. J., Kale, S., and Kumar, S. "On the Convergence of Adam and Beyond". In: *International Conference on Learning Representations*. 2018 (cit. on p. 16).

[112] Al-Rfou, R. et al. "Theano: A Python framework for fast computation of mathematical expressions". In: *arXiv e-prints* abs/1605.02688 (2016) (cit. on pp. 41, 53, 68, 85).

[113] Ritter, H., Botev, A., and Barber, D. "A Scalable Laplace Approximation for Neural Networks". In: *International Conference on Learning Representations*. 2018 (cit. on pp. 4, 64).

[114] Ritter, H., Botev, A., and Barber, D. "Online Structured Laplace Approximations for Overcoming Catastrophic Forgetting". In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS'18. 2018, pp. 3742–3752 (cit. on pp. 4, 64).

[115] Robbins, H. and Monro, S. "A Stochastic Approximation Method". In: *The Annals of Mathematical Statistics* 22 (1951), pp. 400–407 (cit. on p. 14).

[116] Ronneberger, O., Fischer, P., and Brox, T. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*. 2015, pp. 234–241 (cit. on p. 11).

[117] Rosenblatt, F. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386 (cit. on p. 7).

[118] Sagun, L., Evci, U., Ugur Guney, V., Dauphin, Y., and Bottou, L. "Empirical Analysis of the Hessian of Over-Parametrized Neural Networks". In: *arXiv e-prints*, arXiv:1706.04454 (2017), arXiv:1706.04454. arXiv: 1706.04454 [cs.LG] (cit. on p. 80).

[119] Samaria, F. S. and Harter, A. C. "Parameterisation of a Stochastic Model for Human Face Identification". In: *Proceedings of the Second IEEE Workshop on Applications of Computer Vision*. 1994, pp. 138–142 (cit. on p. 52).

[120] Schaul, T., Zhang, S., and LeCun, Y. "No More Pesky Learning Rates". In: *Proceedings of the 30th International Conference on Machine Learning*. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA, 2013, pp. 343–351 (cit. on p. 31).

[121] Schraudolph, N. N. "Fast Curvature Matrix-Vector Products for Second-Order Gradient Descent". In: *Neural Computation* 14.7 (2002), pp. 1723–1738 (cit. on pp. 34, 48, 51).

[122] Settles, B. *Active learning literature survey*. Tech. rep. University of Wisconsin-Madison Department of Computer Sciences, 2009 (cit. on p. 19).

[123] Shazeer, N. et al. "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer". In: *CoRR* abs/1701.06538 (2017). arXiv: 1701.06538 (cit. on p. 60).

[124] Silver, D. et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419 (2018), pp. 1140–1144 (cit. on p. 6).

[125] Simonyan, K. and Zisserman, A. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. 2015 (cit. on p. 6).

[126] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958 (cit. on pp. 18, 63, 70, 72).

[127] Sun, S., Chen, C., and Carin, L. "Learning Structured Weight Uncertainty in Bayesian Neural Networks". In: *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*. Vol. 54. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA, 2017, pp. 1283–1292 (cit. on p. 66).

[128] Szegedy, C. et al. "Going Deeper with Convolutions". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 1–9 (cit. on p. 6).

[129] Szegedy, C. et al. "Intriguing properties of Neural Networks". In: *CoRR* abs/1312.6199 (2014) (cit. on p. 72).

[130] Vaswani, A. et al. "Attention is All you Need". In: *Advances in Neural Information Processing Systems 30*. 2017, pp. 5998–6008 (cit. on p. 12).

[131] Wang, X., Ma, S., Goldfarb, D., and Liu, W. "Stochastic Quasi-Newton Methods for Nonconvex Stochastic Optimization". In: *SIAM Journal on Optimization* 27.2 (2017), pp. 927–956 (cit. on p. 17).

[132] Wolfe, P. "Convergence conditions for ascent methods. II: Some corrections". In: *SIAM review* 13.2 (1971), pp. 185–188 (cit. on p. 17).

[133] Xiao, H., Rasul, K., and Vollgraf, R. "Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms". In: *arXiv preprint arXiv:1708.07747* (2017) (cit. on p. 84).

[134] Yan, X., Yang, J., Sohn, K., and Lee, H. "Attribute2Image: Conditional Image Generation from Visual Attributes". In: *Computer Vision – ECCV 2016*. 2016, pp. 776–791 (cit. on p. 6).

[135] Yang, T., Lin, Q., and Li, Z. "Unified convergence analysis of stochastic momentum methods for convex and non-convex optimization". In: *arXiv preprint arXiv:1604.03257* (2016) (cit. on p. 15).

[136] Yu, L., Hermann, K. M., Blunsom, P., and Pulman, S. "Deep Learning for Answer Sentence Selection". In: *CoRR* abs/1412.1632 (2014). arXiv: 1412.1632 (cit. on p. 6).

[137] Zagoruyko, S. and Komodakis, N. "Wide Residual Networks". In: *Proceedings of the British Machine Vision Conference (BMVC)*. 2016, pp. 87.1–87.12 (cit. on p. 75).

[138] Zeiler, M. D. "ADADELTA: An Adaptive Learning Rate Method". In: *CoRR* abs/1212.5701 (2012). arXiv: 1212.5701 (cit. on p. 15).

[139] Zeiler, M. D. and Fergus, R. "Visualizing and Understanding Convolutional Networks". In: *Computer Vision – ECCV 2014*. Cham, 2014, 818–833" (cit. on p. 6).

[140] Zen, H. and Senior, A. "Deep Mixture Density Networks for Acoustic Modeling in Statistical Parametric Speech Synthesis". In: 2014, pp. 3844–3848 (cit. on p. 60).

[141] Zenke, F., Poole, B., and Ganguli, S. "Continual Learning through Synaptic Intelligence". In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70*. ICML'17. Sydney, NSW, Australia, 2017, pp. 3987–3995 (cit. on pp. 83, 86).