Master of Science in Engineering

Effortless Bayesian Deep Learning

Tapping Into the Potential of Modern Optimizers

Silas Brack

February 2023

DTU Compute

Author
Silas Brack (s174433)

Supervisors
Søren Hauberg (sohau@dtu.dk)
Frederik Warburg (frwa@dtu.dk)
Marco Miani (mmia@dtu.dk)

Project period
12th September 2022 – 12th February 2023

ECTS points
30

Education
Master of Science (M.Sc.)

Class
Public

Edition
1st

Colophon
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class.
The source code of this book is available at:
https://github.com/fmarotta/kaobook

Publisher
DTU Compute, February 2023

Abstract

Bayesian methods promise to provide a principled way to quantify uncertainty in neural networks. This is important for many applications in machine learning, such as those involving safety-critical decisions that rely on risk assessment and interpretability. However, Bayesian inference is often computationally intractable, so approximate methods which compromise performance for practicality are used. The Laplace approximation (LA) is one such method, though it itself typically relies on crude approximations to the posterior precision matrix to make it computationally feasible. When we do not factorise the posterior precision into more practical approximations, existing methods require that the full posterior precision be instantiated. We call this the full Laplace approximation. For all but toy problems, however, since the full precision matrix scales quadratically with the number of parameters, this matrix is too large to be stored in memory.

In this work, we propose a method for computing the Laplace approximation using only Jacobian-vector products. This allows us to perform marginal training and posterior sampling using the full Laplace approximation without storing the entire posterior precision matrix. To accomplish this, we show that we can estimate the posterior precision's log-determinant and inverse square root using only Jacobian-vector products. To overcome the conditioning issues that arise during sampling, we offer several potential preconditioners which can be used to improve convergence of the inverse square root approximation. Next, we give a technique for evaluating the quality of the posterior samples without instantiating nor inverting the posterior precision based on results from traditional statistics. To implement these methods, we use JAX, a library for automatic differentiation which enables us to efficiently perform Jacobian-vector products without explicitly storing the entire posterior precision.

We perform the full Laplace approximation on both a sine function and MNIST using our method. This consists of two steps: (a) training a neural network by maximising either its posterior probability or marginal likelihood and (b) sampling from this posterior. Our approximate maximum marginal training procedure is able to learn a set of parameters which yields performance comparable to that of the maximum posterior and maximum likelihood training procedures. By analysing quantile–quantile plots of our posterior samples and visualising these samples, we find that our approximate sampling method produces samples which are correctly distributed. Our benchmarks of the performance of Jacobian-vector products estimate that our method yields a 10 000x memory reduction over the conventional full Laplace approximation, since it does not store the quadratically-scaling full posterior precision matrix. This is the first method for performing the Laplace approximation while only accessing the posterior precision implicitly.

Resumé

Bayesianske metoder lover at give en principiel måde at kvantificere usikkerheden i neurale netværk på. Dette er vigtigt for mange anvendelser inden for machine learning, f.eks. i forbindelse med sikkerhedskritiske beslutninger, der er afhængige af risikovurdering og fortolkningsmuligheder. Bayesiansk inferens er imidlertid ofte beregningsmæssigt uhåndterbar, så der anvendes tilnærmede metoder, som går på kompromis med ydeevnen for at opnå praktisk anvendelighed. Laplace-approksimationen (LA) er en sådan metode, selv om den typisk er afhængig af grove tilnærmelser af den efterfølgende præcisionsmatrix for at gøre den beregningsmæssigt gennemførlig. Når vi ikke faktoriserer den efterfølgende præcision i mere praktiske tilnærmelser, kræver de eksisterende metoder, at den fulde efterfølgende præcision skal instantieres. Vi kalder dette den fulde Laplace-approksimation. Da den fulde præcisionsmatrix imidlertid for alle problemer undtagen legetøjsproblemer skalerer kvadratisk med antallet af parametre, er denne matrix for stor til at blive lagret i hukommelsen.

I dette studie foreslår vi en metode til beregning af Laplace-approksimationen ved hjælp af Jacobian-vektorprodukter alene. Dette giver os mulighed for at udføre marginal træning og efterfølgende stikprøveudtagning ved hjælp af den fulde Laplace-approksimation uden at lagre hele den efterfølgende præcisionsmatrix. For at opnå dette viser vi, at vi kan estimere den posteriore præcisionens log-determinant og den inverse kvadratrod ved hjælp af kun jacobianiske vektorprodukter. For at overvinde de konditioneringsproblemer, der opstår under sampling, tilbyder vi flere potentielle prækonditioneringsværktøjer, som kan anvendes til at forbedre konvergensen af den inverse kvadratrodsapproksimation. Dernæst giver vi en teknik til at evaluere kvaliteten af de efterfølgende stikprøver uden at instantiere eller invertere den efterfølgende præcision baseret på resultater fra traditionel statistik. Til at gennemføre disse metoder anvender vi JAX, et bibliotek til automatisk differentiering, som gør det muligt at udføre jacobian-vektorprodukter effektivt uden eksplicit lagring af hele den efterfølgende præcision.

Vi udfører den fulde Laplace-approksimation på både en sinusfunktion og MNIST ved hjælp af vores metode. Denne består af to trin: (a) træning af et neuralt netværk ved at maksimere enten dets posterior sandsynlighed eller marginale sandsynlighed og (b) prøveudtagning fra denne posterior. Vores tilnærmede maksimale marginale træningsprocedure er i stand til at lære et sæt parametre, som giver en ydeevne, der er sammenlignelig med den maksimale posterior- og maksimale sandsynlighedstræningsprocedure. Ved at analysere quantile–quantile-plots af vores posteriorprøver og visualisere disse prøver finder vi, at vores tilnærmede prøveudtagningsmetode giver prøver, der er korrekt fordelt. Vores benchmarks af Jacobian-vektorprodukternes ydeevne anslår, at vores metode giver en 10 000x hukommelsesreduktion i forhold til den konventionelle fulde Laplace-approksimation, da den ikke lagrer den kvadratisk skalerende matrix med fuld præcision for den efterfølgende periode. Dette er den første metode til at udføre Laplace-approksimationen, hvor der kun er implicit adgang til den efterfølgende præcision.

Contents

List of Figures

List of Tables

List of Algorithms

Introduction 1

1.1 Bayesian Deep Learning

Bayesian methods allow for estimates of uncertainty which enable more efficient usage of data (e.g., via active learning) and avoid overfitting. They accomplish this by using Bayes' theorem to model a distribution on the model parameters, which is then used to make predictions. From here we can sample from this distribution to make predictions, thereby yielding uncertainty estimates, thus enabling the assessment of model predictive confidence. In doing so, they allow for uncertainty estimates, which is important for many applications in machine learning, such as those involving safety-critical decisions which rely on understanding risk, where traditional deterministic neural networks are overconfident or fail to provide uncertainty estimates at all. The estimation of model confidence is particularly important in domains where silent failures can have catastrophic consequences [1], such as autonomous driving [2], medical diagnosis [3], and financial markets [4]. These methods have been successfully applied to a wide range of classical problems in statistics, though attempts to apply them to deep learning have had limited success. For deep learning, the Bayesian approach is often limited by the computational cost of Bayesian inference. This is because Bayesian inference requires computing the posterior distribution of the model parameters given the data, which, due to the nonlinear complexity of the neural network architecture, is often intractable.

Typically, the Bayesian approach to deep learning is to use a prior distribution over the model parameters and then compute the posterior distribution of the model parameters given the data, where the posterior is maximised to obtain the optimal model parameters for the available data. This posterior distribution can then be sampled from to make predictions. The optimisation can, depending on the choices of likelihood and prior, often be done analytically or numerically. However, computing and sampling from the posterior is often difficult, and so approximate Bayesian methods are used instead. These approaches depend greatly on the choice of prior, which is non-trivial. Often, a prior is chosen which is simple and has a convenient analytical form, such as a Gaussian prior. However, even in this case, the choice of prior precision is still difficult, often being estimated by cross-validation. Instead, maximising the marginal likelihood allows for optimisation of hyperparameters in the likelihood and prior, which can be done using gradient-based optimisation algorithms. However, this is computationally infeasible, and so approximate Bayesian methods are used instead.

1.2 Current Methods

Currently, approximate Bayesian methods are either expensive to compute (Markov chain Monte Carlo) or are limited in their Bayesian inter-

pretation (e.g., Monte Carlo dropout [5], deep ensembles [6], SWAG [7]). Of these, MC dropout and Markov chain Monte Carlo are the most widely used, though both have significant drawbacks. As such, there is demand for a method which exhibits the same computational cost as the optimization algorithms for deterministic neural networks while providing accurate posterior approximations and working out-of-the-box for any given architecture.

The Laplace approximation [8, 9] is a simple yet theoretically well-supported posterior approximation suitable for Bayesian modeling. The Laplace approximation has been applied in, for example, the prediction of earthquake hypocenters [10]. The primary advantages of the Laplace approximation are that it is simple to implement, effective at out-of-distribution detection [11] and can be performed post-hoc on a pre-trained model. This last point is important, since it also means that the Laplace approximation can be as cheap to train as the optimization algorithms for deterministic neural networks, while only requiring a small amount of additional computation to compute the posterior samples for inference.

The Laplace approximation corresponds to performing a second-order Taylor expansion of the log-posterior around the *maximum a posteriori* (MAP) estimate of the model parameters as

$$\log p(\boldsymbol{\theta} \mid \boldsymbol{y}) \approx \log p(\boldsymbol{\theta}_{\text{MAP}} \mid \boldsymbol{y}) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{MAP}})^{\mathsf{T}} \nabla^2_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta} \mid \boldsymbol{y})(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{MAP}}).$$

This is equivalent to approximating the posterior as a Gaussian distribution with mean and precision given by the MAP estimate and the Hessian matrix of the log-posterior respectively, i.e.,

$$p(\boldsymbol{\theta} \mid \boldsymbol{y}) \approx \mathcal{N}\left(\boldsymbol{\theta} \,\middle|\, \boldsymbol{\theta}_{\text{MAP}}, \left(-\nabla^2_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta} \mid \boldsymbol{y})\right)^{-1}\right).$$

To compute the posterior distribution of the model parameters, we thus need to compute the Hessian of the negative log-posterior loss with respect to the model parameters. Typically, this is done by storing the posterior precision matrix explicitly and computing the inverse of this matrix. Since the full Hessian matrix is a $D \times D$ square matrix with D^2 elements, where D is the number of parameters, this is intractable for large models, which can contain millions or billions of parameters, thus limiting the applicability of full Laplace outside of toy problems of a few thousand parameters. To overcome this, we can use a crude approximation, such as only storing the posterior precision matrix's diagonal [12, 13], its Kronecker factorisation [14–16], and other low-rank approximations. Many of these approximations to the posterior precision have successfully been applied to the Laplace approximation [17, 18], though are limited by the quality of the approximation. To overcome this, we propose a method which performs the full Laplace approximation without requiring the precision to be stored explicitly.

1.3 Large-Scale Laplace

We propose an implementation of the Laplace approximation which does not require instantiation of the full posterior precision matrix. In general, the Laplace approximation has two steps. First, we would like to learn a

posterior distribution over the model parameters by maximising either the posterior or the marginal likelihood. Maximising the posterior during training is trivial, since it has a cheap closed-form gradient. However, maximising the marginal likelihood is more difficult.

As we will prove later, the Laplace log-marginal likelihood is given by, ignoring constant terms,

$$\log p(\boldsymbol{y}) \overset{\text{LA}}{\approx} -\frac{1}{2}\rho \sum_{i=1}^{N} \|\boldsymbol{y}_i - f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)\|^2 - \frac{1}{2}\alpha \|\boldsymbol{\theta}\|^2 - \frac{1}{2}\log\det \boldsymbol{\Lambda},$$

where $\boldsymbol{\Lambda}$ is the posterior precision. The first two terms are easy to compute (and differentiate), but computing the log-determinant of very large matrices is generally infeasible, since it typically has $O(D^3)$ complexity or requires storing the matrix in memory. To overcome this, we thus need to estimate the log-determinant of the posterior precision matrix without instantiating it.

The second step of the Laplace approximation is to perform inference and make predictions from the posterior predictive distribution. We estimate the posterior predictive via Monte Carlo by sampling from the posterior distribution and averaging the model predictions over the posterior samples. Since the posterior distribution is approximated by a Gaussian, we sample from the posterior as

$$\boldsymbol{\varepsilon} = \boldsymbol{\theta}_{\text{MAP}} + \boldsymbol{\Lambda}^{-1/2}\boldsymbol{\varepsilon}_0,$$

which effectively only requires computing the inverse square root of the posterior precision matrix. This computation is also intractable for large models, since it usually involves computing the Cholesky decomposition of the precision, which suffers from the same performance issues as the computation of the log-determinant.

As stated before, we will present a method which performs the full Laplace approximation without explicitly storing the posterior precision matrix. Our method approximates the posterior precision's inverse square root and log-determinant using only implicit Jacobian-vector products (JVPs). These JVPs compute products of the posterior precision matrix with a vector without storing the matrix itself in memory. Since this product outputs a vector, the memory requirements are linear in the number of parameters (rather than quadratic), allowing us to perform the full Laplace approximation with significantly larger models. We use JAX [19], a Python library for GPU-accelerated automatic differentiation and computational graph compilation, to efficiently compute Jacobian-vector products while only storing the implicit computational graph. JAX uses XLA [20], a compiler for linear algebra, to compile the JVP computation into efficient machine code. The computation of the log-determinant and inverse square root of the precision will thus enable us to perform training and inference using the Laplace approximation. An overview of the Laplace approximation and our contributions is shown in Figure 1.1.

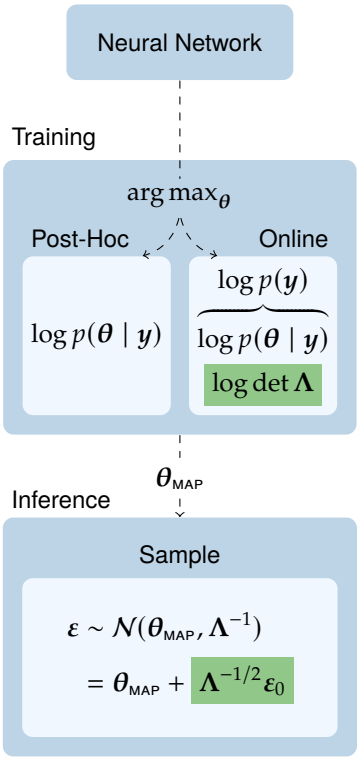


Figure 1.1: Overview of the Laplace approximation (LA) for Bayesian neural networks. We highlight in green the computations which we approximate in this project using only Jacobian-vector products, namely the log-determinant (see Chapter 3) and inverse square root (see Chapter 4) of the posterior precision. Our contributions allow us to forgo the typical crude approximations of the precision by performing the LA using the full posterior precision without instantiating it explicitly.

The Laplace Approximation 2

In this chapter, we discuss training and inference in deep learning from a Bayesian perspective. We introduce the Laplace approximation as a method for approximating the posterior distribution of a model's parameters given the data. We discuss the construction of the Laplace posterior, its properties, and how it can be computed in practice.

2.1 Optimisation in Deep Learning

Suppose a neural network is a real-valued function $f : \mathbb{R}^N \times \mathbb{R}^D \to \mathbb{R}^O$ parametrised in $\boldsymbol{\theta} \in \mathbb{R}^D$ which maps an input $\boldsymbol{x} \in \mathbb{R}^N$ to an output $f(\boldsymbol{x}, \boldsymbol{\theta}) \equiv f_{\boldsymbol{\theta}}(\boldsymbol{x})$. Our goal is to find the parameters $\boldsymbol{\theta}^*$ which best model the observed data. From a frequentist perspective, we can define a loss function on the model $\mathcal{L}(\boldsymbol{\theta}) : \mathbb{R}^D \to \mathbb{R}$ such that the optimal parameters $\boldsymbol{\theta}^*$ minimise this loss function for the given model. Often, we formulate the loss function as the negative log-likelihood of the data under the model, $\mathcal{L}(\boldsymbol{\theta}) = -\log p(\boldsymbol{y} \mid \boldsymbol{\theta})$. In this case, the goal of optimisation is to find the parameters $\boldsymbol{\theta}^*$ which maximise the likelihood of the data.[1]

1: Maximising the likelihood is equivalent to minimising the negative log-likelihood, since the log function is strictly monotonically increasing.

The likelihood function is problem-specific, and is defined as the probability density of the data given the parameters $\boldsymbol{\theta}$. Typically, the likelihood is chosen to be a Gaussian distribution with mean $f_{\boldsymbol{\theta}}(\boldsymbol{x})$ and precision ρ (which is frequently chosen to be equal to one) for regression problems, and a categorical distribution with logits $f_{\boldsymbol{\theta}}(\boldsymbol{x})$ for classification problems. These two cases correspond to the mean-squared error (MSE) and cross-entropy (CE) negative log-likelihood loss functions, respectively. Due to the assumption of independence between observations, we can factorise the likelihood and the negative log-likelihood loss as

$$p(\boldsymbol{y} \mid \boldsymbol{\theta}) = \prod_i^N p(\boldsymbol{y}_i \mid \boldsymbol{\theta}), \tag{2.1}$$

$$\mathcal{L}(\boldsymbol{\theta}) \equiv \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{y}) \coloneqq -\log p(\boldsymbol{y} \mid \boldsymbol{\theta}) = -\sum_i^N \log p(\boldsymbol{y}_i \mid \boldsymbol{\theta}). \tag{2.2}$$

This allows us to compute the loss as the sum of the losses for each individual observation. This loss function is defined in an optimisation problem to find the parameters $\boldsymbol{\theta}^*$ of the model which minimise this loss, thereby maximising the performance of the model on the data.

2.1.1 Gradient Descent

Gradient descent is a first-order optimisation algorithm which iteratively updates the parameters $\boldsymbol{\theta}$ in the direction of the negative gradient of the loss function $\mathcal{L}(\boldsymbol{\theta})$.[2] The update rule is given by

2: First-order optimisation methods only use the first derivative of the loss function.

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_t), \tag{2.3}$$

where η_t is the learning rate at iteration t. The learning rate η_t can be constant or adaptive, and is often chosen to be a non-increasing function of t. The gradient $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_t, \boldsymbol{x}_t, y_t)$ can be computed separately for each individual training example $\boldsymbol{x}_t, y_t$, as seen in Equation 2.1. Using the chain rule on the composition of the loss into the loss as a function of the model output $\mathcal{L}$ and the model output as a function of the parameters f such that $\mathcal{L}(\boldsymbol{\theta}_t, \boldsymbol{x}_t, y_t) = \mathcal{L}(y_t, f(\boldsymbol{x}_t, \boldsymbol{\theta}_t))$,we compute the gradient as

$$\nabla_f \mathcal{L}(\boldsymbol{\theta}_t, \boldsymbol{x}_t, y_t) = \nabla_{\boldsymbol{\theta}} \log p(y_t \mid \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} f_{\boldsymbol{\theta}_t}(\boldsymbol{x}_t), \tag{2.4}$$

where $\nabla_{\boldsymbol{\theta}} f_{\boldsymbol{\theta}_t}(\boldsymbol{x}_t)$ is the gradient of the neural network output with respect to its parameters at iteration t and $\nabla_{\boldsymbol{\theta}} \log p(y_t \mid \boldsymbol{\theta})$ is the gradient of the loss function with respect to the model output. Note that in this case, we perform one gradient update for each data point, which is why we iterate over the data points and update the parameters in the same step t. Thus, gradient descent only requires the gradient of the loss function with respect to the parameters of the neural network, which can be computed efficiently using automatic differentiation for a similar cost to a single forward pass through the network. However, gradient descent is prone to getting stuck in local minima and can be slow to converge. In practice, we cannot compute the gradient of the loss function for the entire dataset at once, since this would require storing the entire dataset in memory and computing the gradient over it. Instead, we compute the gradient of the loss for a subset of the data at each iteration (a batch), and then update the parameters based on this unbiased estimate of the gradient. This is known as stochastic gradient descent (SGD), and is the most common form of gradient descent used in practice. SGD helps avoid getting stuck in local minima, but, like gradient descent, does not take into account the curvature of the loss function, and so can overshoot the minimum.

2.1.2 Adam

Adam [21] is a popular alternative to SGD for training neural networks. It is an approximate second-order optimisation method, which means that it approximates the curvature of the loss. It is a variant of SGD that uses the first and second moments of the gradient of the loss to scale the learning rate of each parameter. In Adam, the first moment of the gradient is approximated by a moving average of the gradient and the second moment of the gradient is approximated by a moving average of the squared gradient. The moving averages are weighted by the parameters β_1 and β_2. The learning rate is then scaled by the ratio of the first and second moments. The update rule for Adam is thus

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \boldsymbol{m}_t \odot \left(\sqrt{\boldsymbol{v}_t} + \varepsilon\right)^{-1}, \tag{2.5}$$

$$\boldsymbol{m}_t = (1 - \beta_1^t)\left[\beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_t)\right], \tag{2.6}$$

$$\boldsymbol{v}_t = (1 - \beta_2^t)\left[\beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_t) \odot \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_t)\right], \tag{2.7}$$

where $\odot$ denotes the Hadamard product, ε is a small constant to prevent division by zero, and $\boldsymbol{m}_t$ and $\boldsymbol{v}_t$ are the first and second moments of the gradient respectively. The momentum term [22] takes inspiration from physics, where the momentum of an object is the product of its mass and velocity. In Adam, the first moment of the gradient is used to compute the momentum of the parameters, increasing the convergence speed of the

optimisation [23]. The second moment of the gradient is also known as the variance of the gradient. Adam approximates the second moment by the diagonal of the Fisher information matrix where the loss and model function are assumed to be linear. This is a common approximation in practice, as this approximation is easy to compute. The Fisher information matrix is discussed further in Subsection 2.3.2. The use of the second moment is inspired by optimisation methods such as Newton's method and the natural gradient [24], which use the second derivative of the loss to compute the optimal step size by scaling the step by the inverse of the curvature of the loss. These optimisation methods identify a single set of parameters as the optimal solution. In the next section, we will discuss how to learn a distribution over parameters instead.

2.2 Maximum a Posteriori Estimation

Bayesian deep learning is a framework for deep learning that allows for modelling the parameters as random variables instead of as single optima in the loss landscape. From a Bayesian perspective, we define a prior distribution $p(\boldsymbol{\theta})$ and a likelihood function $p(\boldsymbol{y} \mid \boldsymbol{\theta})$ such that, under maximum a posteriori estimation, the goal of optimisation is to find the parameters $\boldsymbol{\theta}^*$ which maximise the posterior distribution $p(\boldsymbol{\theta} \mid \boldsymbol{y})$. This way, we can quantify the uncertainty in our model parameters $\boldsymbol{\theta}$ by computing the posterior distribution $p(\boldsymbol{\theta} \mid \boldsymbol{y})$. It is obtained from Bayes' theorem, which allows for uncertainty quantification by

$$p(\boldsymbol{\theta} \mid \boldsymbol{y}) = \frac{p(\boldsymbol{y} \mid \boldsymbol{\theta})\, p(\boldsymbol{\theta})}{p(\boldsymbol{y})}, \tag{2.8}$$

where $p(\boldsymbol{\theta} \mid \boldsymbol{y})$ is the posterior distribution over the parameters, $p(\boldsymbol{\theta})$ is the prior distribution over parameters, $p(\boldsymbol{y} \mid \boldsymbol{\theta})$ is the likelihood of the data given the parameters, and $p(\boldsymbol{y})$ is the marginal likelihood of the data. The marginal likelihood is obtained by marginalising over the parameters, i.e., $p(\boldsymbol{y}) = \int p(\boldsymbol{y} \mid \boldsymbol{\theta})\, p(\boldsymbol{\theta})\, d\boldsymbol{\theta}$. Due to this integral, the marginal is often intractable. However, since the posterior distribution is proportional to the product of the likelihood by the prior, we can obtain the unnormalised posterior from the likelihood and the prior

Since the marginal likelihood is a normalisation constant and does not depend on the parameters, it does not affect the optimisation problem.

$$p(\boldsymbol{\theta} \mid \boldsymbol{y}) \propto p(\boldsymbol{y} \mid \boldsymbol{\theta})\, p(\boldsymbol{\theta}) =: \tilde{p}(\boldsymbol{\theta} \mid \boldsymbol{y}). \tag{2.9}$$

Since we want to determine the parameters which maximise the posterior distribution, we can typically use the same optimisation techniques as for frequentist learning, but with the negative log-posterior loss instead of the negative log-likelihood. Furthermore, since the unnormalised posterior from Equation 2.9 is proportional to the likelihood and the prior, the exact posterior itself need not be tractable. Similarly to the negative log-likelihood, we factorise the negative log-posterior

$$\begin{aligned} \mathcal{L}(\boldsymbol{\theta}) = -\log p(\boldsymbol{\theta} \mid \boldsymbol{y}) &= -\log p(\boldsymbol{y} \mid \boldsymbol{\theta}) - \log p(\boldsymbol{\theta}) \\ &= \underbrace{-\sum_{i}^{N} \log p(\boldsymbol{y}_i \mid \boldsymbol{\theta})}_{\text{Negative log-likelihood}} - \underbrace{\log p(\boldsymbol{\theta})}_{\text{Regularisation}}. \end{aligned} \tag{2.10}$$

Here the regularisation term is given by the prior distribution $p(\boldsymbol{\theta})$. Due to the black-box nature of neural networks, it is often difficult to define a prior distribution over the parameters using previous knowledge. Because of this, the prior is often chosen to be a Gaussian distribution with zero mean and a diagonal covariance matrix with some precision α, i.e., $p(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta} \mid \mathbf{0}, \alpha^{-1}\boldsymbol{I})$.[3] The likelihood term is defined as in the frequentist setting (see Section 2.1). However, the posterior distribution itself is typically intractable, so we approximate the posterior instead.

3: In the frequentist setting, this type of regularisation is known as weight decay or L_2-regularisation. Note that $-1/N \log p(\boldsymbol{y} \mid \boldsymbol{\theta}) - \gamma/2 \|\boldsymbol{\theta}\|^2$ is the typical formulation in this setting, where γ is the regularisation strength and is usually within the range of 10^{-2} to 10^{-4}. In our setting, we are instead computing the log-posterior $-\log p(\boldsymbol{y} \mid \boldsymbol{\theta}) - \alpha/2 \|\boldsymbol{\theta}\|^2$, and so we must multiply the regularisation strength by N to obtain α.

We use the posterior to quantify the uncertainty in our model parameters. Often, all that is needed is a set of samples from the posterior. From here, we can make predictions in a Bayesian setting by computing the posterior predictive distribution and using Monte Carlo sampling as

$$\begin{aligned} p(y^* \mid \boldsymbol{y}) &= \int p(y^* \mid \boldsymbol{x}^*, \boldsymbol{\theta})\, p(\boldsymbol{\theta} \mid \boldsymbol{y})\, d\boldsymbol{\theta}, \\ &\approx \frac{1}{S} \sum_{s=1}^{S} p(y^* \mid \boldsymbol{x}^*, \boldsymbol{\theta}_s), \quad \boldsymbol{\theta}_s \sim p(\boldsymbol{\theta} \mid \boldsymbol{y}), \end{aligned} \tag{2.11}$$

where y^* is the new data point and S is the number of Monte Carlo posterior samples $\boldsymbol{\theta}_s$. However, in some cases, it is useful to have a closed-form expression for the posterior distribution. There are several methods for approximating the posterior distribution, which we will discuss in the following sections. Some, like Markov chain Monte Carlo (MCMC) [25], are exact methods, which can be used to sample from the posterior distribution directly (though they are often computationally expensive). However, they cannot be used to compute the posterior distribution itself directly. Others, like variational inference (VI) [26, 27], are methods which attempt to find a tractable approximation to the posterior distribution by parametrising a family of approximate posteriors and optimising the parameters to minimise the KL divergence between the approximate posterior and the true posterior. The performance of these methods depends on the variational family used to approximate the posterior and the choice of the variational parameters. One common choice for selecting the variational family is to use a Gaussian distribution with diagonal covariance matrix, which is known as mean-field VI.[4] Many variational methods suffer from the difficulty of their optimisation problem, since they often attempt to train variance parameters, which are difficult to optimise [28]. In this project, we focus on the Laplace approximation, in which the posterior is approximated by a Gaussian distribution, but where the variance is deduced rather than optimised.

4: The mean-field assumption refers to the assumption that the parameters are independent of each other which leads to a diagonal covariance matrix.

2.3 The Laplace Approximation

In the Laplace approximation (LA) [8, 9, 29, 30], the posterior is approximated by a Gaussian, similarly to mean-field VI. However, instead of finding the optimal Gaussian distribution locations and scales by maximising the ELBO, the LA finds the location by computing the MAP solution $\boldsymbol{\theta}_{\text{MAP}}$ and the scale by approximating the log-posterior with a second degree Taylor expansion around this solution ($\boldsymbol{\theta}_0 = \boldsymbol{\theta}_{\text{MAP}}$) and determining the curvature via its Hessian matrix $\boldsymbol{\Lambda} = -\nabla^2_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta} \mid \boldsymbol{y})\big|_{\boldsymbol{\theta}_{\text{MAP}}}$. Since the Taylor expansion is performed around the MAP solution, the

first order derivative is zero, and the expansion is given by

$$\begin{aligned}\log p(\boldsymbol{\theta} \mid \boldsymbol{y}) \approx\ & \log p(\boldsymbol{\theta}_{\text{MAP}} \mid \boldsymbol{y}) + \\ & + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{MAP}})^\intercal \left(\nabla^2_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta} \mid \boldsymbol{y})\big|_{\boldsymbol{\theta}_{\text{MAP}}} \right) (\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{MAP}}) \end{aligned} \tag{2.12}$$

$$\Rightarrow \tilde{p}(\boldsymbol{\theta} \mid \boldsymbol{y}) \approx p(\boldsymbol{\theta}_{\text{MAP}} \mid \boldsymbol{y}) \exp\left(-\frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{MAP}})^\intercal \boldsymbol{\Lambda} (\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{MAP}}) \right), \tag{2.13}$$

where $\tilde{p}(\boldsymbol{\theta} \mid \boldsymbol{y})$ corresponds to the unnormalised posterior (Equation 2.9). Dividing it by the normalisation constant $p(\boldsymbol{y})$ gives the normalised posterior

This normalisation constant is computed as the marginal likelihood for the Laplace approximation in Section 3.1.

$$\begin{aligned} p(\boldsymbol{\theta} \mid \boldsymbol{y}) &\approx \sqrt{\frac{\det(\boldsymbol{\Lambda})}{(2\pi)^D}} \exp\left(-\frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{MAP}})^\intercal \boldsymbol{\Lambda} (\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{MAP}}) \right) \\ &= \mathcal{N}(\boldsymbol{\theta} \mid \boldsymbol{\theta}_{\text{MAP}}, \boldsymbol{\Lambda}^{-1}). \end{aligned} \tag{2.14}$$

The Laplace approximation can simply be trained to the MAP solution, where the Hessian is computed to obtain the posterior precision at inference time. This method is known as post-hoc Laplace. However, it is also possible to compute the posterior precision during training. This can be done either by sampling from the neural network at each iteration of training and evaluating the gradient at each sample [11] or by estimating the log-marginal likelihood [8, 31], enabling training of hyperparameters and improved model selection. In this work, we approach the latter, as well as post-hoc Laplace.

We have now shown that approximating the log-posterior with a second degree Taylor expansion around the $\boldsymbol{\theta}_{\text{MAP}}$ corresponds to approximating the posterior with a Gaussian distribution given by $\mathcal{N}(\boldsymbol{\theta} \mid \boldsymbol{\theta}_{\text{MAP}}, \boldsymbol{\Lambda}^{-1})$ where $\boldsymbol{\Lambda} = -\nabla^2_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta} \mid \boldsymbol{y})\big|_{\boldsymbol{\theta}_{\text{MAP}}}$. However, this approximation only holds when the posterior precision $\boldsymbol{\Lambda}$ is positive definite. For BNNs, this is not guaranteed as the loss function is not convex with respect to the model parameters, which means that the normal distribution assumed by the Laplace approximation may not be valid. Furthermore, the Hessian is only ensured to be positive semi-definite at the MAP. This may not be realistic when training neural networks, which tend to have unreliable convergence [32]. In the next sections, we will go into the computation of the Hessian for the negative log-posterior loss function from Section 2.2 and show how the generalised Gauss-Newton approximation can be used to obtain a positive definite Hessian approximation, even when the Hessian itself is not positive definite.

2.3.1 The Hessian

The Hessian is a matrix of second-order partial derivatives of a scalar function.[5] Suppose we have a function $\mathcal{L} : \mathbb{R}^D \to \mathbb{R}$ parametrised by $\boldsymbol{\theta} \in \mathbb{R}^D$. Then the Hessian can be interpreted as the Jacobian matrix of the gradient of the function, as per $J_{\boldsymbol{\theta}}(\nabla_{\boldsymbol{\theta}} \mathcal{L})$.

We are interested in the Hessian of the loss function $\mathcal{L}(\boldsymbol{\theta})$ with respect to the parameters $\boldsymbol{\theta}$. For a neural network with D parameters, the Hessian is therefore a $D \times D$ square matrix. Furthermore, if all the neural network's second partial derivatives are continuous, then the Hessian is symmetric.

5: From now on, we will use the notation $\nabla^2_{\boldsymbol{\theta}} f$ to represent the Hessian matrix of f, where each element is given by

$$\left(\nabla^2_{\boldsymbol{\theta}} f\right)_{i,j} = \frac{\partial^2 f}{\partial \theta_i \partial \theta_j}.$$

In this case, the Hessian is generally dominated by the block-diagonal [15]. However, the Hessian will only be positive definite if the loss function is a convex function in the model parameters $\boldsymbol{\theta}$.

Typically, the loss function is defined as the negative log-likelihood or negative log-posterior. In this framing, if we consider the negative log-posterior loss from Equation 2.10, by the linearity of the derivative, we obtain

$$\begin{aligned}\nabla^2_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}) &= \nabla^2_{\boldsymbol{\theta}}\left(-\sum_i^N \log p(\boldsymbol{y}_i \mid \boldsymbol{\theta}) - \log p(\boldsymbol{\theta})\right) \\ &= -\nabla^2_{\boldsymbol{\theta}}\sum_i^N \log p(\boldsymbol{y}_i \mid \boldsymbol{\theta}) - \nabla^2_{\boldsymbol{\theta}}\log p(\boldsymbol{\theta}) \\ &= -\sum_i^N \nabla^2_{\boldsymbol{\theta}}\log p(\boldsymbol{y}_i \mid \boldsymbol{\theta}) - \nabla^2_{\boldsymbol{\theta}}\log p(\boldsymbol{\theta}).\end{aligned} \tag{2.15}$$

In this construction, the first term is the Hessian of the negative log-likelihood, e.g., the Hessian of the MSE loss. For the common prior given by a zero-mean Gaussian $\boldsymbol{\theta} \sim \mathcal{N}(\mathbf{0}, \alpha^{-1}\boldsymbol{I})$, the log-prior regularisation term is given by $\nabla^2_{\boldsymbol{\theta}}\log p(\boldsymbol{\theta}) = -\alpha\boldsymbol{I}$. However, as we discussed previously, the Hessian of the negative log-likelihood is not always positive definite. In the next section, we will discuss how to obtain a positive definite Hessian approximation.

2.3.2 The Generalised Gauss-Newton Approximation

From the expression of the posterior precision as the Hessian of the negative log-posterior in Equation 2.15 we can apply the chain rule twice and the product rule once to obtain a simpler expression,

$$\begin{aligned}\nabla^2_{\boldsymbol{\theta}}\mathcal{L} &= J_{\boldsymbol{\theta}}(\nabla_{\boldsymbol{\theta}}\mathcal{L}) = J_{\boldsymbol{\theta}}\left(\nabla_f\mathcal{L}\cdot J_{\boldsymbol{\theta}}f\right) \\ &= J_{\boldsymbol{\theta}}(J_{\boldsymbol{\theta}}f)\cdot\nabla_f\mathcal{L} + J_{\boldsymbol{\theta}}\left(\nabla_f\mathcal{L}\right)\cdot J_{\boldsymbol{\theta}}f \\ &= \nabla^2_{\boldsymbol{\theta}}f\cdot\nabla_f\mathcal{L} + J_{\boldsymbol{\theta}}f^{\intercal}\cdot\nabla^2_f\mathcal{L}\cdot J_{\boldsymbol{\theta}}f,\end{aligned} \tag{2.16}$$

where ∇_f is the gradient with respect to $f(\boldsymbol{x}, \boldsymbol{\theta})$ and $J_{\boldsymbol{\theta}}$ is the Jacobian with respect to $\boldsymbol{\theta}$. More precisely, we obtain

$$\nabla^2_{\boldsymbol{\theta}}\mathcal{L}_{ij} = \sum_{n=1}^{O}\frac{\partial^2}{\partial\theta_i\theta_j}f_n\cdot\frac{\partial}{\partial f_n}\mathcal{L} + \sum_{n,m=1}^{O}\frac{\partial}{\partial\theta_i}f_n\cdot\frac{\partial^2}{\partial f_n f_m}\mathcal{L}\cdot\frac{\partial}{\partial\theta_j}f_m, \tag{2.17}$$

where f_n is the n^{th} output of the neural network function $f(\boldsymbol{x}, \boldsymbol{\theta})$.

Assume now we will approximate f using a first-order Taylor expansion. In this way, we are linearising our neural network function. This approximation is given by

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}) \approx f_{\boldsymbol{\theta}_0}(\boldsymbol{x}) + \nabla_{\boldsymbol{\theta}}f_{\boldsymbol{\theta}_0}(\boldsymbol{x})\cdot(\boldsymbol{\theta} - \boldsymbol{\theta}_0). \tag{2.18}$$

Notice that $\nabla^2_{\boldsymbol{\theta}}f_{\boldsymbol{\theta}_0}(\boldsymbol{x}) = 0$ under the Taylor expansion assumption of neural network linearity. By combining Equations 2.16 and 2.18 we then

obtain the generalised Gauss-Newton approximation of the Hessian,

$$\nabla^2_{\boldsymbol{\theta}} \mathcal{L} \approx \nabla_{\boldsymbol{\theta}} f^\intercal \cdot \nabla^2_f \mathcal{L} \cdot \nabla_{\boldsymbol{\theta}} f, \tag{2.19}$$

which corresponds to the second term in Equation 2.16. This means that the GGN approximation is an approximation to the Hessian of the composition of two functions, $\mathcal{L}(\boldsymbol{\theta}) \equiv (\mathcal{L} \circ f)(\boldsymbol{\theta})$ where we linearise the inner function f around a point $\boldsymbol{\theta}_0$. For a given negative log-likelihood loss $\mathcal{L}$, we thus have the GGN approximation of the Hessian

$$\begin{aligned} \nabla^2_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) &= \sum_i^N \nabla^2_{\boldsymbol{\theta}} \log p(\boldsymbol{y}_i \mid \boldsymbol{\theta}) \\ &\approx \sum_{i=1}^N \underbrace{J_{\boldsymbol{\theta}} f(\boldsymbol{x}_i, \boldsymbol{\theta})^\intercal}_{\boldsymbol{J}_i^\intercal} \underbrace{\left(\nabla^2_f \mathcal{L}(\boldsymbol{y}_i, f(\boldsymbol{x}_i, \boldsymbol{\theta})) \right)}_{\boldsymbol{H}_i} \underbrace{J_{\boldsymbol{\theta}} f(\boldsymbol{x}_i, \boldsymbol{\theta})}_{\boldsymbol{J}_i}. \end{aligned} \tag{2.20}$$

For a normal prior with precision α on $\boldsymbol{\theta}$, we obtain the GGN approximation of the posterior precision

$$\boldsymbol{\Lambda} \approx \sum_{i=1}^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i + \alpha \boldsymbol{I}, \tag{2.21}$$

where we have defined $\boldsymbol{J}_i = J_{\boldsymbol{\theta}} f(\boldsymbol{x}_i, \boldsymbol{\theta})$ and $\boldsymbol{H}_i = \nabla^2_f \mathcal{L}(\boldsymbol{y}_i, f(\boldsymbol{x}_i, \boldsymbol{\theta}))$.

For mean square error (MSE) loss (e.g. for regression), we obtain the simple expression for the GGN approximation of the Hessian

$$\boldsymbol{H}_i \approx (\boldsymbol{y}_i - f(\boldsymbol{x}_i, \boldsymbol{\theta}))^2 = 2\boldsymbol{I}, \tag{2.22}$$

$$\boldsymbol{\Lambda} \approx 2 \sum_{i=1}^N \boldsymbol{J}_i^\intercal \boldsymbol{J}_i + \alpha \boldsymbol{I}. \tag{2.23}$$

If the Hessian of the loss function with respect to the model outputs $\boldsymbol{H}_i$ is positive semidefinite, the GGN approximation is positive semidefinite, since each term in the sum $\boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i$ is positive semidefinite.[6] Furthermore, this is the case for most common loss functions, such as MSE loss and cross-entropy loss. For the case of the normal prior, the GGN approximation of the posterior precision will be positive definite, since the eigenvalues of $\alpha \boldsymbol{I}$ are all $\alpha > 0$ and the eigenvalues of $\sum_{i=1}^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i$ are all greater than or equal to zero.

6: This is equivalent to the loss function being a convex function of the model outputs.

It is also interesting to note that, for our definition of the loss, the GGN approximation of the Hessian is equivalent to the Fisher information matrix, which is a measure of the curvature of the log-likelihood function with respect to the model parameters. Here, the Fisher information matrix is given by

$$\mathcal{F} = \sum_{i=1}^N \mathrm{E}_{p(y \mid \boldsymbol{x}_i, \boldsymbol{\theta})} \left[\nabla_{\boldsymbol{\theta}} \log p(y \mid \boldsymbol{x}_i, \boldsymbol{\theta}) \cdot \nabla_{\boldsymbol{\theta}} \log p(y \mid \boldsymbol{x}_i, \boldsymbol{\theta})^\intercal \right], \tag{2.24}$$

where $\mathrm{E}_{p(y \mid \boldsymbol{x}_i, \boldsymbol{\theta})} [\bullet]$ denotes the expectation with respect to the likelihood function. The Fisher matrix (and its empirical equivalent) is relevant as it motivates second-order methods such as natural gradient descent (and thus the Adam optimiser) and the use of the GGN approximation.

2.3.3 Practical Considerations for the GGN

The generalised Gauss-Newton approximation requires that the function for which we would like to compute the Hessian, $h(x)$, can be framed as a composition of two functions $h(x) = g(f(x))$. However, there are many possible choices of functions f and g which, when composed, yield h, and these different compositions can yield wildly varying performance for the GGN approximation [33]. Given that, in our case, h is the loss function with respect to the model parameters of a neural network, a natural choice of functions (and the choice we have assumed above) are $g : \mathbb{R}^O \to \mathbb{R}$ and $f : \mathbb{R}^D \to \mathbb{R}^O$ where g is the loss function with respect to the model output and f is a function which maps the model parameters to a model output, for a given model input x. This construction allows us to capture the curvature of the loss with respect to the model output, which can often be easily computed in a simple closed form, without requiring the computation of the curvature of the model function with respect to its parameters; this therefore corresponds to linearising the model function with a Taylor expansion as in Equation 2.18. However, this is not the only possible construction.

Another choice is to linearise the whole likelihood function with respect to the model parameters. This yields $g(y) = -\log(y)$, and thus we obtain the GGN approximation of the Hessian with this composition as

$$\begin{aligned}\nabla^2_{\boldsymbol{\theta}}\mathcal{L} &= \nabla_{\boldsymbol{\theta}} f^\intercal \cdot \nabla^2_f g \cdot \nabla_{\boldsymbol{\theta}} f = \nabla_{\boldsymbol{\theta}} f^\intercal \cdot f(\boldsymbol{\theta})^{-2} \cdot \nabla_{\boldsymbol{\theta}} f \\ &= \left(f(\boldsymbol{\theta})^{-1} \cdot \nabla_{\boldsymbol{\theta}} f\right)^\intercal \cdot \left(f(\boldsymbol{\theta})^{-1} \cdot \nabla_{\boldsymbol{\theta}} f\right) \\ &= \left(-\nabla_f g \cdot \nabla_{\boldsymbol{\theta}} f\right)^\intercal \cdot \left(-\nabla_f g \cdot \nabla_{\boldsymbol{\theta}} f\right) \\ &= \left[\nabla_{\boldsymbol{\theta}}(g \circ f)\right]^\intercal \cdot \left[\nabla_{\boldsymbol{\theta}}(g \circ f)\right] = \nabla_{\boldsymbol{\theta}}\mathcal{L}^\intercal \cdot \nabla_{\boldsymbol{\theta}}\mathcal{L}.\end{aligned} \tag{2.25}$$

The diagonal of this matrix is given by the element-wise square of the gradient $\nabla_{\boldsymbol{\theta}}\mathcal{L} \odot \nabla_{\boldsymbol{\theta}}\mathcal{L}$. Notice that this is the second-order term used in the Adam optimiser, which we discussed in Subsection 2.1.2.

The choice of application of the generalised Gauss-Newton approximation is not based on having a reduced computational cost—both the Hessian and the GGN approximation require the same number of forward and backward passes. However, the GGN approximation is better behaved than the exact Hessian since it is guaranteed to be positive semidefinite for convex losses, thereby making it more suitable for practical applications. This is because the GGN approximation is designed to be positive semi-definite, while the exact Hessian can be vulnerable to negative curvature. This feature can limit the use of the exact Hessian for algorithms that require a positive definite Hessian, such as, in our case, the Laplace approximation.

In practice, as we explained in Chapter 1, we are looking for methods which access the product of the GGN matrix with a vector, rather than the GGN matrix itself. This then means we want to compute

$$\begin{aligned}\boldsymbol{\Lambda}\boldsymbol{v} &= \left(\sum_{i=1}^{N} \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i + \alpha \boldsymbol{I}\right)\boldsymbol{v} \\ &= \sum_{i=1}^{N} \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i \boldsymbol{v} + \alpha \boldsymbol{v},\end{aligned} \tag{2.26}$$

such that we actually compute the product of the Jacobian of the neural network (as defined in Equation 2.20) by a vector $\boldsymbol{v}$, i.e., $\boldsymbol{J}_i\boldsymbol{v}$.

2.3.4 The Spectrum of the GGN Approximation

When we construct the posterior precision for the Laplace approximation, we compute the posterior precision as in Equation 2.21. For methods which we will introduce later, we are interested in the spectrum (i.e., the set of eigenvalues) of the posterior precision. The eigenvalues of this precision will thus be the sum of the eigenvalues of $\sum_{i=1}^{N} \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i$ and of $\alpha \boldsymbol{I}$, where $\alpha > 0$ is the prior precision. The first term, being the positive semi-definite GGN matrix, will have non-negative eigenvalues. The second term, being a diagonal matrix with entries α, will have eigenvalues that are all equal to α. Therefore, the eigenvalues of the posterior precision will be the sum of the eigenvalues of the GGN matrix and α and will thus all be positive. This means that the prior precision α guarantees the positive definiteness of the posterior precision. For some applications, it is desirable to have a weak prior, so that the regularisation effect of the prior is not too strong. While, theoretically speaking, the eigenvalues of the likelihood GGN matrix are greater than or equal to zero, in practice, due to numerical issues, the eigenvalues can be slightly negative. As such, α must be chosen to be sufficiently large to ensure that the posterior precision is positive definite. While this may appear to detract from the effectiveness of the prior, it is important to note that, in most cases, the prior precision is large enough to ensure that the posterior precision is positive definite.

Since $\boldsymbol{J}_i$ is a rectangular matrix of shape $O \times D$, the rank of each $\boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i$ is at most O, where O is the number of outputs of the model. The rank of the sum of these matrices thus depends on the eigenvectors of each $\boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i$. If we add $\boldsymbol{J}_n^\intercal \boldsymbol{H}_n \boldsymbol{J}_n$ into the sum $\sum_{i=1}^{n-1} \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i$, then, if any of the eigenvectors of $\boldsymbol{J}_n^\intercal \boldsymbol{H}_n \boldsymbol{J}_n$ are not contained in the span of the eigenvectors of the sum, then the rank of the sum will increase. This is because the rank of a matrix is the number of linearly independent columns. Thus, non-rigorously, we have that the closer the eigenvectors (assuming that the eigenvectors have the same ordering between the two matrices) are between terms in the sum, the closer the largest eigenvalue will be to the sum of the largest eigenvalues of each term.[7] Practically speaking, this means that the rank of the sum of the GGN matrices can increase as we add more data points (but will not be greater than $N \cdot O$). Furthermore, as we add more data points, the eigenvalues of the sum of the GGN matrices will increase. Specifically, the largest eigenvalue of the sum of the GGN matrices will be the less than or equal to the sum of the largest eigenvalues of the GGN matrices. Additionally, the non-zero spectrum of the sum of the GGN matrices will itself get steeper as we add more data points. If the O non-zero eigenvectors are identical across the GGN matrices, then this non-zero spectrum of the sum of the GGN matrices will just be the sum of the spectrum of each GGN matrix and should thus not get steeper.[8] However, if these eigenvectors are not identical, then as the largest eigenvalue of the sum of the GGN increases and the number of non-zero eigenvalues of the sum of the GGN increases, the non-zero spectrum of the sum of the GGN matrices will get steeper. For these reasons, as we add more data points, the posterior precision will

7: By "same ordering" we mean that they are eigenvectors with respect to the same relative eigenvalue.

8: If the GGN matrices $\boldsymbol{J}^\intercal \boldsymbol{H} \boldsymbol{J}$ have an identical spectrum

$$\{\underbrace{s_1, s_2, \dots, s_O}_{O}, \underbrace{0, \dots, 0}_{D-O}\},$$

then the sum of GGN matrices as per $\sum^N \boldsymbol{J}^\intercal \boldsymbol{H} \boldsymbol{J} = N \boldsymbol{J}^\intercal \boldsymbol{H} \boldsymbol{J}$ has the spectrum

$$\{\underbrace{Ns_1, \dots, Ns_O}_{O}, \underbrace{0, \dots, 0}_{D-O}\}.$$

become more ill-conditioned.[9] If the sum of the GGN matrices is indeed low-rank, then the prior precision α will be the smallest eigenvalue of the posterior precision.

Empirically, we find that, for MNIST, the effective rank of the posterior precision is significantly less than $N \cdot O$ (see Figure 2.1). This suggests that the eigenvectors of the posterior precision are relatively similar across different observations, though not identical. However, this seems to greatly depend on the structure of the neural network (e.g., the filter size) in ways that are not yet clear.

9: A matrix is considered ill-conditioned when the ratio of its largest eigenvalue to its lowest eigenvalue, its *condition number* ($\kappa := \lambda_{\max}/\lambda_{\min}$), is very large.

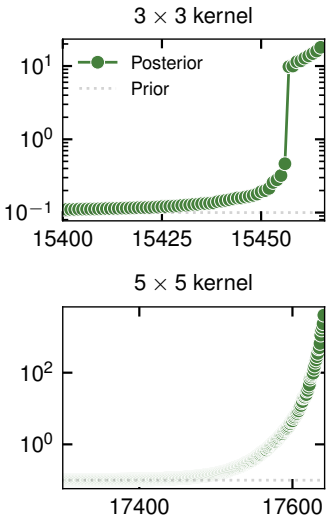


Figure 2.1: Spectrum of the Laplace posterior precision on 100 MNIST observations with a CNN. Note the log scale and the fact that we are zoomed in on the largest eigenvalues, as can be seen from the eigenvalue indices on the x-axis. For the 3×3 filter, there are two distinct clusters in the spectrum, with the first cluster being over an order of magnitude larger than the second. This first cluster consists of nine eigenvalues, and so the precision has an effective rank close to nine. For the 5×5 filter, we observe a hyper-exponential distribution of eigenvalues in this spectrum, which does not have any distinct clusters. This suggests that the effective rank of the precision is over 50, which is still significantly lower than the number of observations $N = 100$ times the number of outputs $O = 10$.

2.4 Limitations

One limitation of the Laplace approximation is that it can only approximate one mode of the posterior distribution (the MAP). However, most approximate posterior distributions, such as MCMC and variational inference, also struggle to approximate multimodal distributions. The standard way to solve this problem is to use a multimodal approximation such as a mixture, typically by training multiple models from different initialisations and then combining them in a deep ensemble. This is also possible with the Laplace approximation [34]. However, deep ensembles are quite expensive, as they require training multiple models. Additionally, it is not clear whether there is a significant benefit to using a multimodal Laplace approximation over a single mode Laplace approximation (i.e., whether generalisation improves).

Another limitation of the Laplace approximation is that it is not scalable to non-trivial problems for the full Laplace approximation without performing crude approximations. This is because the full Laplace approximation requires the computation of the posterior precision, which is a $D \times D$ matrix, where D is the number of parameters. Thus, in order to scale to large models and large datasets without approximating the posterior precision with, e.g., a diagonal matrix, we need to find a method to perform the Laplace approximation without instantiating the posterior precision. During training, we typically only need to train the model in a deterministic way (to find the MAP), and then we can use the Laplace approximation to sample from the approximate posterior post-hoc. Thus, for inference, we need to determine a method for sampling from the Gaussian approximate posterior without explicitly computing the posterior precision. We have so far only considered the Laplace approximation computed by maximising posterior (i.e., MAP estimation), but it is also possible to perform the Laplace approximation by training a model on its marginal likelihood (i.e., the likelihood of the data under the model). This would reduce overfitting and allow us to optimise hyperparameters via backpropagation. However, we must find a differentiable way to compute the Laplace approximate log-marginal likelihood without explicitly storing the posterior precision.

Training 3

In this chapter, we will discuss training neural networks by maximising the evidence under the Laplace approximation. We first motivate maximising the evidence as opposed to maximising the posterior and discuss how the Laplace approximation guarantees the feasibility of the evidence, as well as the issues that arise when computing the marginal likelihood with the full posterior precision. Next, we propose a novel method for computing an upper bound on the log-determinant of the full posterior precision matrix which avoids the aforementioned issues. Finally, we discuss the implementation of online Laplace with mini-batching using our upper bound.

3.1 Maximising the Evidence

As seen in Equation 2.8, the posterior probability for a given model is

$$p(\boldsymbol{\theta} \mid \boldsymbol{y}) = \frac{p(\boldsymbol{y} \mid \boldsymbol{\theta})\, p(\boldsymbol{\theta})}{p(\boldsymbol{y})}, \tag{3.1}$$

where $p(\boldsymbol{y})$ is known as the evidence. It is also known as the *marginal likelihood*, as it is obtained by marginalising $\boldsymbol{\theta}$ from the likelihood as

$$p(\boldsymbol{y}) = \int_{\boldsymbol{\theta}} p(\boldsymbol{y} \mid \boldsymbol{\theta})\, p(\boldsymbol{\theta})\, d\boldsymbol{\theta}. \tag{3.2}$$

This term is typically intractable, since it involves integrating over the entire parameter space. For this reason, we typically approximate the posterior by some tractable distribution, such as a Gaussian, and then use this to approximate the marginal likelihood.

Since we are integrating over the posterior, maximising the marginal likelihood will seek minima that are flat, since the posterior will be very broad in these regions. This is a desirable property, since it means that the model will be less likely to overfit the data, both from a theoretical standpoint [35] and in practice [7, 36, 37].

To further motivate training on the evidence term, Fong and Holmes [38] show that maximising the marginal (and consequently the log-marginal) is equivalent to performing leave-p-out cross validation for all values $p = 1, \ldots, \infty$ and choosing the model with the highest average posterior probability (across each of p folds and across all values of p). This result shows that the marginal likelihood is a good proxy for the true cross-validation performance of a model. Because of this, it can be used for model selection, since it will choose the model that performs best on average across all permutations of the data.

Additionally, unlike the posterior, the marginal likelihood can be used to optimise both model parameters and hyperparameters, since we can meaningfully minimise the log-marginal with respect to the hyperparameters (and compute the respective gradients). This is useful, since it allows

us to use gradient-based optimisation methods to find the hyperparameters that maximise the marginal likelihood, as has been commonly done in Gaussian processes [39, 40]. This provides an alternative to finding the hyperparameters by grid search, which is computationally expensive.

The Laplace approximation provides an approximation to the posterior which allows us to compute and maximise the marginal likelihood. This is known as *online* Laplace, and is an alternative to the *post-hoc* Laplace approach, which is the standard approach to training under the Laplace approximation [8]. Under this approximation, we approximate the posterior distribution as $p(\boldsymbol{\theta} \mid \boldsymbol{y}) \approx \mathcal{N}(\boldsymbol{\theta} \mid \boldsymbol{\theta}_{\text{MAP}}, \boldsymbol{\Lambda}^{-1})$, where the posterior precision $\boldsymbol{\Lambda} = \nabla^2_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_{\text{MAP}})$ is the Hessian of the negative log-posterior at the maximum a posteriori (MAP) estimate $\boldsymbol{\theta}_{\text{MAP}}$. As in Equation 2.12, we have the approximate Laplace log-posterior

$$\begin{aligned}\log p(\boldsymbol{\theta} \mid \boldsymbol{y}) &\overset{\text{LA}}{\approx} \log p(\boldsymbol{\theta}_{\text{MAP}} \mid \boldsymbol{y}) + \nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta}_{\text{MAP}} \mid \boldsymbol{y}) \cdot (\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{MAP}}) \\ &\quad + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{MAP}}) \nabla^2_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta}_{\text{MAP}} \mid \boldsymbol{y})(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{MAP}}) \\ &= \log p(\boldsymbol{\theta}_{\text{MAP}} \mid \boldsymbol{y}) - \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{MAP}}) \boldsymbol{\Lambda} (\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{MAP}}).\end{aligned} \tag{3.3}$$

In reality, we will not be at the MAP solution when optimising the marginal, neither during training nor after convergence (the marginal and the posterior have different minima). In this case, we cannot neglect the first derivative term in Equation 3.3. However, when this term is not zero, the Laplace posterior is $\mathcal{N}(\boldsymbol{\theta} \mid \boldsymbol{\theta}^*_{\text{MAP}}, \boldsymbol{\Lambda}^{-1})$, where $\boldsymbol{\theta}^*_{\text{MAP}} \coloneqq \boldsymbol{\theta}_{\text{MAP}} + \boldsymbol{\Lambda}^{-1} \nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta}_{\text{MAP}} \mid \boldsymbol{y})$. This corresponds to shifting the parameters to the MAP with a Newton step $\boldsymbol{\theta}^*_{\text{MAP}} \coloneqq \boldsymbol{\theta}_{\text{MAP}} - \nabla^2_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_{\text{MAP}})^{-1} \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_{\text{MAP}})$, since Newton's method finds the minimum in one step for quadratic functions (like the second order Taylor expansion to our loss).

The approximate Laplace marginal is then computed as

$$\begin{aligned}p(\boldsymbol{y}) &\overset{\text{LA}}{\approx} \int_{\boldsymbol{\theta}} p(\boldsymbol{\theta}_{\text{MAP}} \mid \boldsymbol{y}) \exp\left(-\frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{MAP}}) \boldsymbol{\Lambda} (\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{MAP}})\right) d\boldsymbol{\theta} \\ &= p(\boldsymbol{\theta}_{\text{MAP}} \mid \boldsymbol{y}) \int_{\boldsymbol{\theta}} \exp\left(-\frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{MAP}}) \boldsymbol{\Lambda} (\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{MAP}})\right) d\boldsymbol{\theta} \\ &= p(\boldsymbol{\theta}_{\text{MAP}} \mid \boldsymbol{y}) \, (2\pi)^{D/2} \, (\det \boldsymbol{\Lambda})^{-1/2}.\end{aligned} \tag{3.4}$$

Taking the log of this approximate marginal, we obtain

$$\begin{aligned}\log p(\boldsymbol{y}) &\overset{\text{LA}}{\approx} \log p(\boldsymbol{\theta}_{\text{MAP}} \mid \boldsymbol{y}) + \frac{D}{2} \log(2\pi) - \frac{1}{2} \log \det \boldsymbol{\Lambda} \\ &= \log p(\boldsymbol{y} \mid \boldsymbol{\theta}_{\text{MAP}}) + \log p(\boldsymbol{\theta}_{\text{MAP}}) + \frac{D}{2} \log(2\pi) - \frac{1}{2} \log \det \boldsymbol{\Lambda}.\end{aligned} \tag{3.5}$$

For the normal likelihood and normal prior, we have

$$p(\boldsymbol{y} \mid \boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{y} \mid f_{\boldsymbol{\theta}}(\boldsymbol{x}), \rho^{-1} \boldsymbol{I}) = \prod_{i=1}^{N} \mathcal{N}(\boldsymbol{y}_i \mid f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \rho^{-1} \boldsymbol{I}), \tag{3.6}$$

$$\begin{aligned}\log p(\boldsymbol{y} \mid \boldsymbol{\theta}) &= \sum_{i=1}^{N} \log \mathcal{N}(\boldsymbol{y}_i \mid f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \rho^{-1} \boldsymbol{I}) \\ &= \sum_{i=1}^{N} \left(-\frac{O}{2} \log(2\pi) - \frac{1}{2} \log \det(\rho^{-1} \boldsymbol{I}) - \frac{1}{2} \rho \left\| \boldsymbol{y}_i - f_{\boldsymbol{\theta}}(\boldsymbol{x}_i) \right\|^2\right) \\ &= -\frac{NO}{2} \log(2\pi) + \frac{NO}{2} \log \rho - \frac{1}{2} \rho \sum_{i=1}^{N} \left\| \boldsymbol{y}_i - f_{\boldsymbol{\theta}}(\boldsymbol{x}_i) \right\|^2,\end{aligned} \tag{3.7}$$

and

$$p(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta} \mid \boldsymbol{0}, \alpha^{-1} \boldsymbol{I}), \tag{3.8}$$

$$\log p(\boldsymbol{\theta}) = -\frac{D}{2} \log(2\pi) + \frac{D}{2} \log \alpha - \frac{1}{2} \alpha \left\| \boldsymbol{\theta} \right\|^2, \tag{3.9}$$

where α is the precision of the prior and ρ is the precision of the

likelihood. Combining Equations 3.5, 3.7 and 3.9, we expand the Laplace approximation to the log-marginal likelihood as

$$\begin{aligned}\log p(\boldsymbol{y}) \overset{\text{LA}}{\approx} & -\frac{NO}{2}\log(2\pi) + \frac{NO}{2}\log\rho - \frac{1}{2}\rho\sum_{i=1}^{N}\|\boldsymbol{y}_i - f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)\|^2 \\ & - \frac{D}{2}\log(2\pi) + \frac{D}{2}\log\alpha - \frac{1}{2}\alpha\|\boldsymbol{\theta}\|^2 \\ & + \frac{D}{2}\log(2\pi) - \frac{1}{2}\log\det\boldsymbol{\Lambda},\end{aligned} \tag{3.10}$$

where it can be seen that there are constant terms which do not depend on $\boldsymbol{\theta}$, α, or ρ. While these terms are not necessary for the optimisation problem, we choose to include them, as these terms allow us to interpret the log-marginal as a likelihood and do not significantly affect performance. From Equation 3.10, we can see that the Laplace log-marginal likelihood requires computation of the log-determinant of the posterior precision matrix. This log-determinant term gets minimised in our optimisation, which favours models with small eigenvalues in the Hessian of the loss, i.e., points of low curvature in the loss landscape, as we argued earlier in this section. This is a challenging computation, as there is no *cheap* closed-form expression for the log-determinant of a matrix without instantiating the matrix. However, we can compute an upper bound on the log-determinant of the posterior precision matrix, which is useful for our optimisation problem which minimises the log-determinant term.

3.2 The Determinant Bound

We want to compute an upper bound on the log-determinant of the posterior precision $\boldsymbol{\Lambda} \in \mathbb{R}^{D\times D}$ such that we have $\log\det\boldsymbol{\Lambda} \leq B_D(\mu_1, \mu_2, \beta)$. Bai and Golub [41] provide one such bound,

$$B_D(\mu_1, \mu_2, \beta) \coloneqq \begin{pmatrix}\log\beta & \log t\end{pmatrix}\begin{pmatrix}\beta & t \\ \beta^2 & t^2\end{pmatrix}^{-1}\begin{pmatrix}\mu_1 \\ \mu_2\end{pmatrix}, \tag{3.11}$$

with $\mu_1 = \operatorname{Tr}(\boldsymbol{\Lambda})$, $\mu_2 = \|\boldsymbol{\Lambda}\|_F^2 = \operatorname{Tr}\left(\boldsymbol{\Lambda}^2\right)$, $t = \frac{\beta\mu_1 - \mu_2}{\beta D - \mu_1}$, where β is an upper bound on the largest eigenvalue of $\boldsymbol{\Lambda}$. This bound amounts to approximating the log eigenvalues of $\boldsymbol{\Lambda}$ with a second-order polynomial. Since $B_D(\mu_1, \mu_2, \beta)$ is an upper bound on the log-determinant of $\boldsymbol{\Lambda}$, we are computing a lower bound on the log-marginal likelihood (which we maximise), as per Equation 3.10.

Assume $\sum_{i=1}^{N}\boldsymbol{J}_i^\intercal\boldsymbol{H}_i\boldsymbol{J}_i$ has rank $K \leq O \cdot N$. Then $\boldsymbol{\Lambda} = \sum_{i=1}^{N}\boldsymbol{J}_i^\intercal\boldsymbol{H}_i\boldsymbol{J}_i + \alpha\boldsymbol{I}$ has K distinct eigenvalues greater than α and the remaining $D - K$ eigenvalues are equal to α. We then decompose the log-determinant as

$$\begin{aligned}\log\det\boldsymbol{\Lambda} &= \sum_{k=1}^{K}\log(\lambda_k + \alpha) + \sum_{k=K+1}^{D}\log\alpha \\ &= \sum_{k=1}^{K}\log(\lambda_k + \alpha) + (D - K)\log\alpha,\end{aligned} \tag{3.12}$$

where $\lambda_1, \ldots, \lambda_K$ are the non-zero eigenvalues of $\sum_{i=1}^{N}\boldsymbol{J}_i^\intercal\boldsymbol{H}_i\boldsymbol{J}_i$. The first term is the log-determinant of a positive definite matrix with eigenvalues

$\{\lambda_1 + \alpha, \dots, \lambda_K + \alpha\}$, and the second term is a constant. The former can then be bounded using Equation 3.11 with an upper bound on its eigenvalues $\beta \geq \lambda_1 + \alpha$ as $B_D\left(\sum_{k=1}^K(\lambda_k + \alpha), \sum_{k=1}^K(\lambda_k + \alpha)^2, \beta\right)$. We then compute the trace of this term as in Equation 3.14 via

$$\operatorname{Tr}\mathbf{\Lambda} = \sum_{k=1}^{K}(\lambda_k + \alpha) + (D - K)\alpha \tag{3.13}$$

$$\begin{aligned}\Rightarrow \sum_{k=1}^{K}(\lambda_k + \alpha) &= \operatorname{Tr}\mathbf{\Lambda} - (D - K)\alpha \\ &= \operatorname{Tr}\left(\sum_{i=1}^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i\right) + K\alpha,\end{aligned} \tag{3.14}$$

and for the square trace we have

$$\sum_{k=1}^{K}(\lambda_k + \alpha)^2 = \operatorname{Tr}\left(\left(\sum_{i=1}^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i\right)^2\right) + 2\alpha \operatorname{Tr}\left(\sum_{i=1}^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i\right) + K\alpha^2. \tag{3.15}$$

We can thus bound $\log\det\mathbf{\Lambda}$ with Equations 3.12, 3.14 and 3.15 as

$$\log\det\mathbf{\Lambda} \leq B_K(\mu_1, \mu_2, \beta) + (D - K)\log\alpha, \tag{3.16}$$

$$\mu_1 = \operatorname{Tr}\left(\sum_{i=1}^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i\right) + K\alpha, \tag{3.17}$$

$$\mu_2 = \operatorname{Tr}\left(\left(\sum_{i=1}^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i\right)^2\right) + 2\alpha \operatorname{Tr}\left(\sum_{i=1}^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i\right) + K\alpha^2. \tag{3.18}$$

For the upper bound β on the eigenvalues of $\sum_{i=1}^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i$, we can exploit that this matrix is positive semidefinite and use $\beta = \operatorname{Tr}(\sum_{i=1}^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i) + \alpha$ as an upper bound.[1] From Equations 3.17 and 3.18, we can see that we need to calculate the traces $\operatorname{Tr}(\sum_{i=1}^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i)$ and $\operatorname{Tr}((\sum_{i=1}^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i)^2)$ to compute the upper bound on the log-determinant of the posterior precision.

1: Since the matrix is positive semidefinite, all eigenvalues are non-negative and thus the trace is bigger than the largest eigenvalue.

3.2.1 Hutchinson's Trace Estimator

To compute the upper bound on the log-determinant of the posterior precision, we need to compute μ_1 and μ_2, which depend on the traces $\operatorname{Tr}(\sum_{i=1}^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i)$ and $\operatorname{Tr}((\sum_{i=1}^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i)^2)$. To compute these, we can use Hutchinson's trace estimator [42]. Let $\boldsymbol{\varepsilon}_0 \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$. Since $\mathrm{E}\left[\boldsymbol{\varepsilon}_0 \boldsymbol{\varepsilon}_0^\intercal\right] = \boldsymbol{I}$, we compute μ_1 as

$$\mu_1 = \operatorname{Tr}(\textstyle\sum_{i=1}^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i) + K\alpha, \tag{3.19}$$

$$\begin{aligned}\operatorname{Tr}(\textstyle\sum_{i=1}^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i) &= \operatorname{Tr}\left(\mathrm{E}\left[\boldsymbol{\varepsilon}_0 \boldsymbol{\varepsilon}_0^\intercal\right]\left(\textstyle\sum_{i=1}^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i\right)\right) \\ &= \mathrm{E}\left[\operatorname{Tr}(\boldsymbol{\varepsilon}_0 \boldsymbol{\varepsilon}_0^\intercal \left(\textstyle\sum_{i=1}^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i\right))\right] \\ &= \mathrm{E}\left[\boldsymbol{\varepsilon}_0^\intercal \left(\textstyle\sum_{i=1}^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i\right) \boldsymbol{\varepsilon}_0\right].\end{aligned} \tag{3.20}$$

Similarly, we compute μ_2 as

$$\mu_2 = \operatorname{Tr}\left(\left(\textstyle\sum_{i=1}^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i\right)^2\right) + 2\alpha \operatorname{Tr}\left(\textstyle\sum_{i=1}^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i\right) + K\alpha^2, \tag{3.21}$$

$$\operatorname{Tr}\left(\left(\textstyle\sum_{i=1}^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i\right)^2\right) = \mathrm{E}\left[\boldsymbol{\varepsilon}_0^\intercal \left(\textstyle\sum_{i=1}^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i\right)\left(\textstyle\sum_{i=1}^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i\right)^\intercal \boldsymbol{\varepsilon}_0\right]. \tag{3.22}$$

We then apply a Monte Carlo approximation of the expectation by sampling S samples from $\boldsymbol{\varepsilon}_0 \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ to compute the traces. Fortunately, this means that we only need to compute one GGN-vector product

$\sum_{i=1}^{N} \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i \cdot \boldsymbol{\varepsilon}_0$ to estimate the μ_1 and μ_2 terms in Equation 3.16. Furthermore, we have that the trace can be rearranged and simplified to

$$\operatorname{Tr}\left(\sum_{i=1}^{N} \underbrace{\boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i}_{D \times D}\right) = \sum_{i=1}^{N} \operatorname{Tr}\left(\boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i\right) = \sum_{i=1}^{N} \operatorname{Tr}\left(\underbrace{\boldsymbol{H}_i \boldsymbol{J}_i \boldsymbol{J}_i^\intercal}_{O \times O}\right), \tag{3.23}$$

which means we can reduce the dimensionality of $\boldsymbol{\varepsilon}_0$ to the number of model outputs O instead of the (significantly larger) number of parameters D, reducing the variance and computational cost of the trace estimator. Empirically, we find that, without the trace rearrangement in Equation 3.23, the variance of the trace estimator is too high to be useful.

3.2.2 Scaling

In standard negative log-likelihood training, the only term in the loss is $-\log p(\boldsymbol{y} \mid \boldsymbol{\theta})$. For regression, the loss function which has a Bayesian interpretation as the negative log-likelihood is the sum of squared errors (SSE), though often the mean squared error is used instead, since computing the correct gradient just involves scaling the step size η up by the number of observations N. For negative log-posterior training, the same scaling can be applied with a Gaussian prior by dividing the prior precision by N. However, for the log-marginal likelihood, scaling the terms is no longer possible, since scaling the loss by a constant will not affect the gradients of the log-determinant term. This means that it is important to compute the negative log-likelihood correctly as the SSE, since the gradients of the log-marginal likelihood are not invariant to scaling the log-marginal by a constant. To compute the posterior precision using a mini-batch, we thus have to scale the likelihood term by N/B to account for the likelihood being computed on a mini-batch of size B instead of the full dataset of size N, yielding $N/B \sum_{i=1}^{B} \boldsymbol{J}_i \boldsymbol{H}_i \boldsymbol{J}_i + \alpha \boldsymbol{I}$. The log-determinant of this mini-batched precision is equivalent to

$$\begin{aligned} \log\det\left(\frac{N}{B}\sum_{i=1}^{B} \boldsymbol{J}_i \boldsymbol{H}_i \boldsymbol{J}_i + \alpha \boldsymbol{I}\right) &= \log\det\left(\frac{N}{B}\left(\sum_{i=1}^{B} \boldsymbol{J}_i \boldsymbol{H}_i \boldsymbol{J}_i + \tfrac{B}{N}\alpha \boldsymbol{I}\right)\right) \\ &= D \log\frac{N}{B} + \log\det\left(\sum_{i=1}^{B} \boldsymbol{J}_i \boldsymbol{H}_i \boldsymbol{J}_i + \tfrac{B}{N}\alpha \boldsymbol{I}\right). \end{aligned} \tag{3.24}$$

The scaling is thus important for determining the relative importance of the prior and likelihood terms in the eigenvalues. To correctly compute the log-determinant of the posterior precision, we scale the trace and square traces for Equations 3.17 and 3.18 as

$$\operatorname{Tr}\left(\frac{N}{B}\sum_{i=1}^{B} \boldsymbol{J}_i \boldsymbol{H}_i \boldsymbol{J}_i\right) = \frac{N}{B} \operatorname{Tr}\left(\sum_{i=1}^{B} \boldsymbol{J}_i \boldsymbol{H}_i \boldsymbol{J}_i\right), \tag{3.25}$$

$$\operatorname{Tr}\left(\left(\frac{N}{B}\sum_{i=1}^{B} \boldsymbol{J}_i \boldsymbol{H}_i \boldsymbol{J}_i\right)^2\right) = \frac{N^2}{B^2} \operatorname{Tr}\left(\left(\sum_{i=1}^{B} \boldsymbol{J}_i \boldsymbol{H}_i \boldsymbol{J}_i\right)^2\right). \tag{3.26}$$

By combining Equations 3.11, 3.16, 3.19, 3.21, 3.23, 3.25 and 3.26, we can therefore compute a differentiable upper bound on the log-marginal likelihood under the Laplace approximation using only Jacobian-vector products without needing to instantiate the posterior precision.

4 Sampling

We will now discuss how to perform inference by sampling from the Laplace posterior distribution. We first give an overview of how we typically sample from the Laplace posterior and discuss the memory constraints of this sampling method when using the full posterior precision matrix. In Section 4.2, we propose an algorithm for sampling from the Laplace posterior that does not require storing the full precision matrix, allowing us to scale sampling to larger models. Finally, we discuss how to mitigate some of the issues of our method (Section 4.3) and propose a practical technique for determining whether sampling was successful (Section 4.4).

4.1 Sampling from the Laplace Posterior

In many methods, such as the Laplace approximation (Section 2.3), we sample from a multivariate normal distribution parametrised by a mean vector $\boldsymbol{\mu}$ and a covariance matrix $\boldsymbol{\Sigma}$. To sample from a Gaussian distribution, you can apply the reparametrisation trick to compute a vector of samples as

$$\boldsymbol{\varepsilon} = \boldsymbol{\mu} + \boldsymbol{\Sigma}^{1/2}\boldsymbol{\varepsilon}_0, \tag{4.1}$$

where $\boldsymbol{\varepsilon}_0$ are samples drawn from a standard normal distribution, i.e., $\boldsymbol{\varepsilon}_0 \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$.

Using the Laplace approximation, we estimate the precision matrix as the Hessian matrix of the loss with respect to the parameters, as discussed in Section 2.3. Because of this, the covariance matrix is given by the inverse of this Hessian, and we thus sample from the Laplace-approximate posterior distribution as

$$\boldsymbol{\varepsilon} = \boldsymbol{\theta}_{\text{MAP}} + \boldsymbol{\Lambda}^{-1/2}\boldsymbol{\varepsilon}_0, \tag{4.2}$$

where $\boldsymbol{\Lambda} = \nabla^2_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta} \mid \boldsymbol{y})$. In Equation 4.1, the square root of a matrix yields another matrix which, when multiplied by itself, is equal to the original matrix. However, since an isotropic Gaussian distribution is invariant to rotation, in this special case, we just need to obtain a matrix which returns the original matrix *up to a rotation*.[1] We thus need to find a method which computes the inverse square root of the full $D \times D$ posterior precision matrix up to a rotation, where D is the number of model parameters.

Some methods exist which compute the inverse square root of a matrix up to a rotation, such as the Cholesky decomposition. However, these methods are not suitable for large matrices, as they require the instantiation of the full precision matrix, which is not feasible for large matrices. There are also methods which can compute the inverse square root of a low-rank matrix that is constructed as the inner product of two rectangular matrices by performing the inverse square root on the

1: That is, if we have a vector of standard normal samples $\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$, then, for an orthogonal matrix $\boldsymbol{R}$ (i.e., a rotation matrix), $\boldsymbol{R}^\intercal\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$. We then have that we can sample *non*-standard normal samples as $\boldsymbol{\Lambda}^{-1/2}\boldsymbol{R}^\intercal\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Lambda}^{-1})$.

outer product (which is low-dimensional for low-rank problems), such as the Woodbury matrix identity. However, the precision matrix we are interested in is not itself low-rank but a sum of low-rank matrices, and thus this method is not suitable for our problem. This is discussed in more detail in Sections 2.3 and 4.3.

Furthermore, methods which do not require computation of the square root of the covariance matrix typically attempt to exploit properties such as a block structure or sparsity of the covariance matrix [43]. However, we do not have a good understanding of the structure of the Laplace covariance matrix, and thus these methods are not suitable for our problem. When sampling from high-dimensional Gaussian distributions, it is common to use Markov chain Monte Carlo (MCMC) sampling methods such as Gibbs sampling [43]. However, these methods are tricky to tune and tend to converge very slowly for high-dimensional problems. As such, we do not use these methods for sampling from the Laplace approximation, and instead sample using the reparametrisation trick. For this, we need a method to compute the inverse square root using only matrix-vector products.

4.2 Contour Integral Quadrature

Pleiss et al. [44] propose one such method. Contour integral quadrature (CIQ) attempts to solve the problem of computing the inverse square root of a matrix, up to a rotation, by approximating the expression $f(\boldsymbol{K})\boldsymbol{\varepsilon}_0 = \boldsymbol{K}^{-1/2}\boldsymbol{\varepsilon}_0$ where $\boldsymbol{K}$ is a positive semi-definite matrix using Cauchy's integral formula. To approximate this integral, it exploits the Lanczos algorithm, Gaussian quadrature, and the msMINRES algorithm, methods which only require the computation of matrix-vector products $\boldsymbol{K}\boldsymbol{v}$. There are then essentially three steps to CIQ, namely:

1. Perform the Lanczos algorithm to compute a lower bound on the maximum eigenvalue and an upper bound on the minimum eigenvalue of $\boldsymbol{K}$.
2. Sample Q quadrature points from a complex-plane circle to approximate the contour integral from Cauchy's integral formula, using these quadrature points to create Q shifted linear systems.
3. Solve these Q shifted systems using the msMINRES algorithm.

Algorithm 1: Contour Integral Quad (CIQ)

Input : $\boldsymbol{K} > 0, \boldsymbol{b}, \boldsymbol{P} > 0, J > 0, Q > 0$
Output: $\boldsymbol{s} = \boldsymbol{K}^{-1/2}b$
$\_, \boldsymbol{\alpha}, \boldsymbol{\beta} \leftarrow \text{CG}(\boldsymbol{K}, \boldsymbol{b}, \boldsymbol{P}, J)$;
$\boldsymbol{M}_{\text{Lanczos}} \leftarrow \text{TriDiag}(\boldsymbol{\alpha}, \boldsymbol{\beta})$;
$\lambda_{\boldsymbol{M}} \leftarrow \lambda(\boldsymbol{M}_{\text{Lanczos}})$;
$\lambda_{\min} \leftarrow \min \lambda_{\boldsymbol{M}}$;
$\lambda_{\max} \leftarrow \max \lambda_{\boldsymbol{M}}$;
$w_1, \ldots, w_Q, t_1, \ldots, t_Q \leftarrow \text{Quadrature}(\lambda_{\min}, \lambda_{\max}, Q)$;
$\boldsymbol{c}_1, \ldots, \boldsymbol{c}_Q \leftarrow \text{msMINRES}(\boldsymbol{K}, \boldsymbol{b}, t_q, \boldsymbol{P}, J)$; /* $\left(t_q \boldsymbol{I} + \boldsymbol{K}\right)^{-1} \boldsymbol{b}$. */
$\boldsymbol{s} \leftarrow \sum_{q=1}^{Q} w_q \boldsymbol{c}_q$;

Since this procedure (see Algorithm 1) computes the approximate inverse square root product $\boldsymbol{K}$ by $\boldsymbol{\varepsilon}_0$ up to a rotation, i.e.,

$$\boldsymbol{K}^{-1/2}\boldsymbol{\varepsilon}_0, \tag{4.3}$$

it is then also trivial to compute the square root product by multiplying $\boldsymbol{K}$ onto this vector, as per

$$\boldsymbol{K}^{1/2}\boldsymbol{\varepsilon}_0 = \boldsymbol{K} \cdot \underbrace{\boldsymbol{K}^{-1/2}\boldsymbol{\varepsilon}_0}_{\text{Equation 4.3}} . \tag{4.4}$$

The first step of CIQ invokes Cauchy's integral formula. Cauchy's integral formula is a central theorem in complex analysis which states that, for a holomorphic function f, there exists a closed contour Γ in the complex plane (see Figure 4.1) which encloses the eigenvalues of $\boldsymbol{K}$ such that $f(\boldsymbol{K})$ can be approximated as

$$f(\boldsymbol{K}) = \frac{1}{2\pi i} \oint_\Gamma f(\tau)\,(\tau \boldsymbol{I} - \boldsymbol{K})^{-1}\, d\tau, \tag{4.5}$$

where $\oint_\Gamma$ is the contour integral along the contour Γ parametrised in the complex variable τ. Since, in our case, the eigenvalues of $\boldsymbol{K}$ will all be real-valued, a circle centred on the real axis which encloses the minimum and maximum eigenvalues of $\boldsymbol{K}$ will suffice. To estimate these values, we can use the Lanczos algorithm, a Krylov subspace method which adapts the power iteration method, to compute a lower bound on the maximum eigenvalue and an upper bound on the minimum eigenvalue of $\boldsymbol{K}$.[2]

We then apply Cauchy's integral formula to $f(\boldsymbol{K}) = \boldsymbol{K}^{-1/2}$, apply a change of variable, and then approximate this contour integral using the quadrature rule with Q quadrature points (see Figure 4.2), as per

$$\begin{aligned} \boldsymbol{K}^{-1/2} &= \frac{1}{2\pi i} \oint_\Gamma \tau^{-1/2}\,(\tau \boldsymbol{I} - \boldsymbol{K})^{-1}\, d\tau \\ &\approx \frac{1}{2\pi i} \sum_{q=1}^{Q} \tilde{w}_q \tau_q^{-1/2} \left(\tau_q \boldsymbol{I} - \boldsymbol{K}\right)^{-1}, \end{aligned} \tag{4.6}$$

where we have the sampled quadrature points $\tau_1, \ldots, \tau_Q \in \Gamma$ and the quadrature weights $\tilde{w}_1, \ldots, \tilde{w}_Q \in \mathbb{R}$. Since we will be sampling a fixed number of quadrature points from the circle, the closer the contour is to the singular points (i.e., the eigenvalues), the better the approximation to the integral will be.

We could sample these quadrature points from the circle uniformly, which corresponds to the regular trapezoid quadrature rule. However, the convergence for this quadrature is linear with regard to the condition number of $\boldsymbol{K}$ [44, 45]. Since the precision matrix (which will be computed from the Laplace approximation as the GGN matrix) can often have a low rank and poor conditioning, uniform sampling would then require a very large number of quadrature points Q and would therefore be inadequate [44].

Since our GGN precision matrix $\boldsymbol{\Lambda} = \sum_{i=1}^{N} \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i$ is often low-rank, we would instead prefer to oversample quadrature points which lie close to the minimum eigenvalue. This is accomplished by applying a change of

The inverse square root function is holomorphic for all positive definite real matrices.

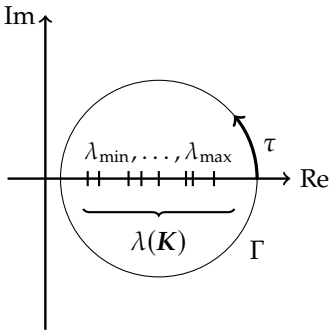


Figure 4.1: Integration circle Γ contained within the domain of f as defined according to the Cauchy integral formula.

If a real-valued matrix is symmetric, its eigenvalues are real. In our case, $\boldsymbol{K}$ will be the precision matrix obtained from the Laplace approximation.

2: Krylov subspace methods are a class of iterative methods which estimate certain properties (e.g., the rank, eigenvalues) of a matrix $\boldsymbol{K}$ with a Krylov subspace $\mathcal{K}_j(\boldsymbol{K}, \boldsymbol{v})$ for some vector $\boldsymbol{v}$. This subspace is spanned by the set of images of $\boldsymbol{v}$ under the first j powers of $\boldsymbol{K}$, $\mathcal{K}_j(\boldsymbol{K}, \boldsymbol{v}) = \text{span}\{\boldsymbol{v}, \boldsymbol{K}\boldsymbol{v}, \ldots, \boldsymbol{K}^{j-1}\boldsymbol{v}\}$.

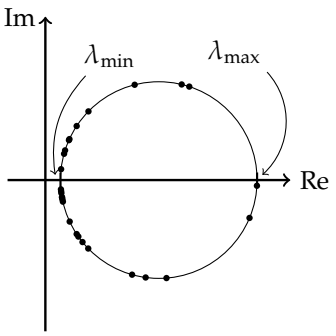


Figure 4.2: The shifts $\tau_1, \ldots, \tau_Q$ are sampled from a circle with its centre on the real axis which intersects the real axis at the points $\lambda_{\min}$ and $\lambda_{\max}$. Since we use Jacobi elliptic functions, we over-sample quadrature points that are closer to the minimum eigenvalue $\lambda_{\min}$.

variable from τ to $\sigma = \tau^{1/2}$ and sampling the quadrature points from σ. To account for this change of variable, we reformulate Equation 4.6 as

$$\begin{aligned} \boldsymbol{K}^{-1/2} &= \frac{1}{2\pi i} \oint_{\Gamma} \tau^{-1/2} (\tau \boldsymbol{I} - \boldsymbol{K})^{-1} \, d\tau \\ &= \frac{1}{\pi i} \oint_{\Gamma_\sigma} \left(\sigma^2 \boldsymbol{I} - \boldsymbol{K}\right)^{-1} d\sigma \\ &\approx \frac{1}{\pi i} \sum_{q=1}^{Q} \tilde{w}_q \left(\sigma_q^2 \boldsymbol{I} - \boldsymbol{K}\right)^{-1}. \end{aligned} \tag{4.7}$$

We can now adapt this equation to allow for computation using only matrix-vector products. From here, the inverse square root vector product is given by

$$\boldsymbol{K}^{-1/2}\boldsymbol{v} \approx \frac{1}{\pi i} \sum_{q=1}^{Q} \tilde{w}_q \underbrace{\left(\sigma_q^2 \boldsymbol{I} - \boldsymbol{K}\right)^{-1} \boldsymbol{v}}_{Q \text{ system solves}}. \tag{4.8}$$

We can then solve the Q systems of equations to obtain the solutions $\boldsymbol{c}_q = \left(\tau_q \boldsymbol{I} - \boldsymbol{K}\right)^{-1} \boldsymbol{v}$ for $q \in 1, \ldots, Q$.

Since the integrand is symmetric with respect to the real axis, only the imaginary component of Γ_σ is required.[3] Since σ is then imaginary (its real part is zero), $\tau \coloneqq \sigma^2$ is real such that $\tau = -\operatorname{Im}(\sigma)^2$. As such, since $\operatorname{Im}(\sigma)^2$ must be positive, τ must then be real-valued and negative. In practice, we incorporate the $1/(\pi i)$ term into the quadrature weights and, since $\tilde{w}_q$ also happens to be real-valued and negative, we can redefine the quadrature weights $w_q = -\tilde{w}_q$ and the quadrature points $t_q = -\sigma_q^2$ for simplicity. This yields the quadrature rule result $\boldsymbol{K}^{-1/2}\boldsymbol{v} \approx \sum_{q=1}^{Q} w_q \boldsymbol{c}_q$, where $\boldsymbol{c}_q = (t_q \boldsymbol{I} + \boldsymbol{K})^{-1}$.

3: This is because the circle with its centre on the x-axis is symmetric with respect to said x-axis.

With this redefinition, both $w_q > 0$ and $t_q > 0$. This then means that $\left(t_q \boldsymbol{I} + \boldsymbol{K}\right)$ is positive definite.

This change of variables from τ to σ yields a set of quadrature weights $w_1, \ldots, w_Q$ given by

$$w_q = \frac{2\sqrt{\lambda_{\min}}}{\pi Q} \mathcal{K}'(k) \cdot \operatorname{cn}(i u_q \mathcal{K}'(k) \mid k) \cdot \operatorname{dn}(i u_q \mathcal{K}'(k) \mid k), \tag{4.9}$$

and a set of quadrature points (shifts) given by

$$\sigma_q^2 = \lambda_{\min} \cdot \operatorname{sn}(i u_q \mathcal{K}'(k) \mid k)^2, \tag{4.10}$$

where sn, cn, and dn are the Jacobi elliptic functions and $\mathcal{K}'(k)$ is the complete elliptic integral, such that $\mathcal{K}'(k) = \mathcal{K}(k') = \mathcal{K}(\sqrt{1-k^2})$. The complete elliptic integral of the first kind is given by

$$\mathcal{K}(p) = \int_0^{\pi/2} \left(1 - m \sin(t)^2\right)^{-1/2} dt, \tag{4.11}$$

where $p = 1 - m$, and is defined in the domain $0 < p \leq 1$.[4] This integral is approximated by

$$\mathcal{K}(p) \approx P(p) - \log(p) Q(p), \tag{4.12}$$

where P and Q are tenth-order polynomials. Additionally, we need to compute the Jacobian elliptic functions $\operatorname{sn}(u \mid m)$, $\operatorname{cn}(u \mid m)$, and

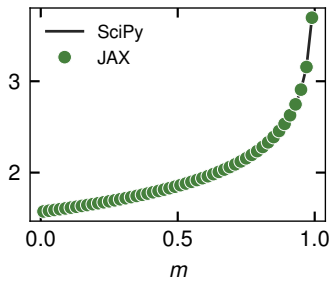


Figure 4.3: Comparison of SciPy and JAX `ellipk` implementations. For values for which the complete elliptic integral $\mathcal{K}(p)$ is well-defined, both implementations yield the same results. Furthermore, due to its static nature, JAX implementation is compatible with *just-in-time* compilation.

4: If $p > 1$, then what we compute is instead the identity $\mathcal{K}(p) = \mathcal{K}(1/p)/\sqrt{p}$.

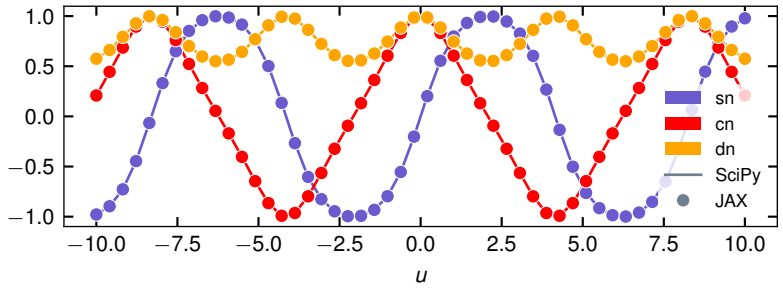


Figure 4.4: Comparison of SciPy and JAX `ellipj` implementations. For values for which the functions are well-defined, both implementations yield the same results. Plotted are $\mathrm{sn}(u \mid m = 0.7)$, $\mathrm{cn}(u \mid m = 0.7)$, and $\mathrm{dn}(u \mid m = 0.7)$. Furthermore, due to its static nature, JAX implementation is compatible with *just-in-time* compilation.

$\mathrm{dn}(u \mid m)$. To compute these values, we first calculate the elliptic modulus $k = \kappa(\boldsymbol{K})^{-1/2}$ (where $\kappa(\boldsymbol{K})$ denotes the condition number of $\boldsymbol{K}$) and $u_q = (q - 1/2)/Q$. These functions are not implemented in JAX, and SciPy uses the Cephes numerical library in C to compute them. Because of this, we ported the Cephes Jacobi elliptical functions to JAX, and Figures 4.3 and 4.4 show the functions evaluated for the two implementations. Our static JAX implementation allows for efficient *just-in-time* compilation, which is not possible with the SciPy implementation. We can thereby now compute the quadrature weights and shifts as described in Algorithm 2.

The last step involves solving the Q linear systems defined in Equation 4.8 to determine $\boldsymbol{c}_q$. As mentioned above, this is done by using msMINRES, an algorithm for solving linear systems of the form $(\boldsymbol{K} + t_q \boldsymbol{I})\boldsymbol{c}_q = \boldsymbol{b}$ where $\boldsymbol{K}$ is symmetric. Like other Krylov methods, msMINRES computes the solutions to the linear systems by constructing a Krylov subspace $\mathcal{K}_q$ of $\boldsymbol{K} + t_q \boldsymbol{I}$ and then iteratively minimises the norm of the residual in $\mathcal{K}_q$. While msMINRES only requires that the matrices $\boldsymbol{K} + t_q \boldsymbol{I}$ be symmetric, and not necessarily also positive definite (since the shifts t_q are positive and $\boldsymbol{K}$ is positive semidefinite), the sum is also positive definite. However, the CIQ implementation from Pleiss et al. [44] uses msMINRES, and we follow their implementation. There is also evidence suggesting that msMINRES converges faster than, for example, the conjugate gradient method (CG) [46], a Krylov method that requires the matrices to be positive definite.

Since the Lanczos algorithm only requires the matrix-vector product $\boldsymbol{K}\boldsymbol{v}$ with a random vector $\boldsymbol{v}$ to estimate a bound on the eigenvalues, we can use the same bound for all Q linear systems. As such, the computational bottleneck of the contour integral quadrature is performing the msMINRES for the shifted systems. Additionally, since msMINRES is an iterative Krylov method, its convergence bound is sensitive to the conditioning of the linear systems, i.e., the conditioning of $\boldsymbol{K}$. To mitigate the effect of ill conditioning, we use a preconditioner $\boldsymbol{P}$ such that $\boldsymbol{P}^{-1}\boldsymbol{K}$ is well-conditioned.

Algorithm 2: Quadrature

Input : $\lambda_{\min}, \lambda_{\max}, Q > 0$
Output: $w_q, t_q \mid q \in 1, \ldots, Q$

$k^2 \leftarrow \lambda_{\min}/\lambda_{\max}$;
$k'^2 \leftarrow \sqrt{1 - k^2}$;
$K' \leftarrow \mathcal{K}(k'^2)$;
for $q \leftarrow 1$ **to** Q **do**
  $u_q \leftarrow (q - 1/2)/Q$;
  $\overline{\mathrm{sn}}_q \leftarrow \mathrm{sn}(u_q K' \mid k'^2)$;
  $\overline{\mathrm{cn}}_q \leftarrow \mathrm{cn}(u_q K' \mid k'^2)$;
  $\overline{\mathrm{dn}}_q \leftarrow \mathrm{dn}(u_q K' \mid k'^2)$;
  $\mathrm{sn}_q \leftarrow i(\overline{\mathrm{sn}}_q / \overline{\mathrm{cn}}_q)$;
  $\mathrm{dn}_q \leftarrow (\overline{\mathrm{dn}}_q / \overline{\mathrm{cn}}_q)$;
  $\mathrm{cn}_q \leftarrow (1/\overline{\mathrm{cn}}_q)$;
  $w_q \leftarrow \lambda_{\min}^{1/2} K' \,\mathrm{cn}_q \,\mathrm{dn}_q$;
  $w_q \leftarrow 2w_q/(\pi Q)$;
  $t_q \leftarrow -\lambda_{\min} \,\mathrm{sn}_q^2$;

4.3 Preconditioning

One drawback of CIQ is that the accuracy its of approximation of $\boldsymbol{\Lambda}^{-1/2}\boldsymbol{\varepsilon}_0$ suffers when the precision matrix $\boldsymbol{\Lambda}$ is poorly conditioned. In the case of

the Laplace approximation, the precision matrix is given by

$$\boldsymbol{\Lambda} = \sum_{i}^{N} \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i + \alpha \boldsymbol{I}, \tag{4.13}$$

$$\boldsymbol{\theta} \sim \mathcal{N}(\boldsymbol{\theta}_{\text{MAP}}, \boldsymbol{\Lambda}^{-1}), \tag{4.14}$$

where N is the number of observations in your dataset.

For preconditioning, we need to find a preconditioner matrix $\boldsymbol{P}$ and its inverse $\boldsymbol{P}^{-1}$ such that the matrix $\boldsymbol{P}^{-1}\boldsymbol{\Lambda}$ is well-conditioned. This is done by choosing $\boldsymbol{P}$ such that the two matrices are "similar", i.e., they have close eigenvalues and eigenvectors. In this way, the preconditioner acts as an initial best guess of the value to be computed. We are thus seeking a preconditioner $\boldsymbol{P}$ such that

$$\boldsymbol{P}^{-1}\boldsymbol{A} \approx \boldsymbol{I} \Rightarrow \boldsymbol{P} \approx \boldsymbol{A}, \tag{4.15}$$

$$\boldsymbol{P}^{-1}\boldsymbol{\Lambda} = \boldsymbol{P}^{-1}\left(\textstyle\sum_i^N \boldsymbol{J}_i^\intercal \boldsymbol{H} \boldsymbol{J}_i + \alpha \boldsymbol{I}\right), \tag{4.16}$$

where $\boldsymbol{\Lambda}$ is the posterior precision matrix given by Equation 4.13 to improve convergence of the msMINRES and Lanczos algorithms. The eigenvalues of our GGN approximation are thus given by

$$\lambda\left(\textstyle\sum_i^N \boldsymbol{J}_i^\intercal \boldsymbol{H} \boldsymbol{J}_i + \alpha \boldsymbol{I}\right) = \Big\{ \underbrace{s_1, \ldots, s_K}_{K \leq N \cdot O}, \underbrace{\alpha, \ldots, \alpha}_{D-K} \Big\}, \tag{4.17}$$

where $\{s_1, \ldots, s_K\}, K \leq N \cdot O$ are the non-zero eigenvalues of the rank-K matrix $\sum_i^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i$. If a preconditioner approximates all eigenvalues and eigenvectors of a matrix, then the preconditioner and the matrix are equal and the preconditioner is exact. However, we often do not have a preconditioner which is exact, but rather an approximation of the matrix. In this case, there is a tradeoff between the accuracy of the preconditioner and its computational cost. Furthermore, the inverse of the preconditioner is required for preconditioning, which is often not available. To approximate the inverse of the preconditioner, we can use the conjugate gradient method, though this method is then sensitive to the condition number of the preconditioner; and if the original matrix $\boldsymbol{\Lambda}$ is ill-conditioned, then a good preconditioner will also tend to be ill-conditioned.

As such, the condition number of our preconditioner also becomes an issue, since we have two conflicting objectives: to have a preconditioner which is as similar to $\boldsymbol{\Lambda}$ as possible and to have a preconditioner which we can invert easily. The first point lends itself toward the use of preconditioners with a similar condition number as $\boldsymbol{\Lambda}$. However, to invert an arbitrary positive semi-definite matrix, we will often resort to using the conjugate gradient method which is sentitive to the condition number of the preconditioner. Thus, there is an intrinsic balance to be struck between a preconditioner which is very similar to $\boldsymbol{\Lambda}$ (but whose condition number is problematic) and a preconditioner which is less similar to $\boldsymbol{\Lambda}$ (and will therefore not improve the conditioning of the problem).

Let us consider the toy example of a preconditioner which we can guarantee to have the same eigenvectors as $\boldsymbol{\Lambda}$ and whose eigenvalues we can freely choose. To do so, one could hypothetically attempt to compute the eigenvectors of $\boldsymbol{\Lambda}$ and then use these eigenvectors as the columns of the

preconditioner. If we have a rank-one preconditioner whose only eigenvector is the largest eigenvector of $\mathbf{\Lambda}$, then the eigenvalue corresponding to this eigenvector becomes one and the second-largest eigenvalue becomes the largest (assuming that the second-largest eigenvalue is greater than one). Thus, in the best-case scenario, you can then eliminate the largest eigenvalue of $\mathbf{\Lambda}$ by using a rank-one preconditioner. Thus, if we have a rank-K preconditioner, then we can eliminate at most the K challenging (largest or smallest) eigenvalues of $\mathbf{\Lambda}$ by using a rank-K preconditioner. The steeper the spectrum of $\mathbf{\Lambda}$, the more effective this preconditioning will be (however, in this case, $\mathbf{\Lambda}$ will probably also tend to be worse conditioned). For a preconditioner with a condition number of κ', we can at best obtain a preconditioned system with conditioning κ/κ', where κ is the condition number of $\mathbf{\Lambda}$. This is because we want the highest eigenvalue of the preconditioner to neutralise (i.e., align with) the highest eigenvalue of $\mathbf{\Lambda}$, and the same for the lowest eigenvalues.

In CIQ we will, in each iteration, perform one matrix-vector product $\mathbf{\Lambda}\boldsymbol{v}$ and one preconditioner-vector product, as seen in Algorithm 1.As such, if the preconditioner-vector product is similarly or more expensive to compute compared to the matrix-vector product, then preconditioning will significantly slow down the algorithm. However, we usually choose preconditioners which are relatively easy to compute relative to the posterior precision itself. Additionally, an effective preconditioner can significantly improve the convergence speed of CIQ, and this effect can outweigh the cost of computing the preconditioner. Thus, we will often choose to use a preconditioner which is not exact, but which is still close to $\mathbf{\Lambda}$ and is cheap to compute. We will now discuss some potential choices for preconditioners to sample using CIQ.

4.3.1 Fully Linear GGN

One of the simplest preconditioners is similar to the second moment estimation used in Adam (Subsection 2.1.2). If we calculate the GGN approximation of the Hessian by linearizing over the whole likelihood (as described in Subsection 2.3.3), we get

$$\boldsymbol{P} = \nabla\nabla^{\mathsf{T}} + \alpha\boldsymbol{I} \tag{4.18}$$

$$\boldsymbol{P}^{-1} = \beta \cdot \nabla\nabla^{\mathsf{T}} + \alpha^{-1}\boldsymbol{I} \tag{4.19}$$

$$\beta = \left(\alpha \left\|\nabla\right\|^2 + \left\|\nabla\right\|^4\right)^{-1} - \left(\alpha \left\|\nabla\right\|^2\right)^{-1}, \tag{4.20}$$

where ∇ is the gradient of the loss with regard to the model parameters. Since Equation 4.18 is in the form of a diagonal perturbation to a vector outer product, we have the closed-form inverse given by Equations 4.19 and 4.20. The problem with using this approximation is that the rank of the outer product is one and this term determines the largest eigenvalue of the preconditioner. This means that, best-case scenario, we can only reduce the highest eigenvalue of the Hessian. This occurs when the gradient corresponds to the direction of the highest precision (this may not be the case). This would then leave the second-highest eigenvalue unaffected. We therefore need to find a higher-rank approximation of the posterior precision.

4.3.2 Sub-Sampling the Data

We could just choose to calculate the true posterior precision over fewer observations and invert it using the conjugate gradient method to use it as a preconditioner. This would have the advantage of being guaranteed to converge to the true posterior precision as the number of preconditioner observations increases. The spectrum of this preconditioner would thus be similar to that of the true posterior precision. Since the lowest eigenvalue of the posterior precision is the prior precision α, we can (even with few preconditioner observations) increase the lowest eigenvalue of the preconditioned system to be equal to one.

Alternatively, we can estimate the eigenvectors and eigenvalues of $N/B \sum_i^B J_i^\intercal J_i$, where B is the number of preconditioner observations. This would *hopefully* have an eigenbasis that is close to that of $\sum_i^N J_i^\intercal J_i$ without needing to calculate the precision matrix over the whole dataset (we may need to guarantee all classes are represented in $x_i, i \sim \mathcal{B}$).[5] However, scaling the preconditioner up to the size of the entire dataset will lead to a preconditioner which is ill-conditioned, making inversion via conjugate gradient difficult.

5: Empirically, it seems likely that the eigenvectors of the posterior precision are relatively similar across different observations—see Figure 2.1 and Subsection 2.3.4.

As mentioned before, each msMINRES iteration performs one matrix-vector product and one preconditioner-vector product. However, even though the preconditioner is constructed in nearly the same way as the sum of GGN matrices in the posterior precision, the preconditioner is significantly cheaper to compute. This is because the preconditioner-vector product will cost $B/N \cdot t_{\text{GGN}}$, where t_{GGN} is the computational cost of performing the GGN-vector product.[6] Since we will have $N \gg B$, then the cost of computing the preconditioner will be negligible compared to that of the GGN-vector product.

6: This analysis assumes that the computation of the GGN-vector product scales linearly in the number of observations used in the GGN matrix. Mathematically speaking, this is a valid assumption, due to the number of elementary operations that are performed in the algorithm. However, the efficient JAX XLA-compiled implementation of Jacobian-vector products can scale differently to naïve implementations, so the validity of the assumption is less clear in our case.

4.3.3 Using the Woodbury Matrix Identity

The Woodbury matrix identity [47] is a useful identity for computing the inverse of a matrix which is the sum of two matrices of which one is low-rank and the other can easily be inverted. The identity is given by

$$(\boldsymbol{A} + \boldsymbol{BCD})^{-1} = \boldsymbol{A}^{-1} - \boldsymbol{A}^{-1}\boldsymbol{B}\left(\boldsymbol{C}^{-1} + \boldsymbol{D}\boldsymbol{A}^{-1}\boldsymbol{B}\right)^{-1}\boldsymbol{D}\boldsymbol{A}^{-1}, \tag{4.21}$$

where $\boldsymbol{A}$ is a $D \times D$ matrix and can easily be inverted, $\boldsymbol{B}$ is a $D \times O$ matrix, $\boldsymbol{C}$ is a $O \times O$ matrix, and $\boldsymbol{D}$ is a $O \times D$ matrix. This identity can be used to compute the inverse of the GGN matrix with the prior precision as

$$\left(\alpha\boldsymbol{I} + \boldsymbol{J}^\intercal\boldsymbol{H}\boldsymbol{J}\right)^{-1} = \alpha^{-1}\boldsymbol{I} - \alpha^{-2}\boldsymbol{J}^\intercal\left(\boldsymbol{H}^{-1} + \alpha^{-1}\boldsymbol{J}\boldsymbol{J}^\intercal\right)^{-1}\boldsymbol{J}, \tag{4.22}$$

where α is the prior precision and $\boldsymbol{H} \in \mathbb{R}^{O\times O}$ and $\boldsymbol{J} \in \mathbb{R}^{O\times D}$ are the Hessian and Jacobian from Equation 2.20 evaluated at a single observation.

Here, to exploit the low-rank structure of the preconditioner, we only compute the GGN matrix over one observation. For this preconditioner to be effective, we assume that the eigenvectors of the GGN matrix are approximately constant across the dataset. This may be a reasonable assumption for many problems, but is not guaranteed to be true.

However, even if the eigenvectors of the GGN matrix are constant across the dataset, the eigenvalues of this preconditioner should be large enough to reduce the condition number of the posterior precision matrix. To do so, we can scale the outer product $\boldsymbol{J}\boldsymbol{J}^\intercal$ term by some factor, e.g., the number of observations, N. Since, by scaling up the outer product term we are effectively scaling the eigenvalues of $\boldsymbol{J}\boldsymbol{J}^\intercal$ by a constant, the condition number of this outer product will remain the same. In this way, we can increase the maximum eigenvalues of the preconditioner without increasing the condition number of the outer product $\boldsymbol{J}\boldsymbol{J}^\intercal$ which we want to invert using the conjugate gradient method.

Before the Woodbury matrix identity can be used, the Hessian of the loss with respect to the model outputs $\boldsymbol{H}$ must first be inverted. To use the conjugate gradient method to invert this Hessian, we would have to perform nested conjugate gradient to also compute the inverse of the outer product term $\boldsymbol{H}^{-1} + \alpha^{-1}\boldsymbol{J}\boldsymbol{J}^\intercal$, which could be very expensive, depending on the number of outputs. For regression tasks where the Hessian is diagonal (Equation 2.22), this is not a problem. For classification tasks using the cross-entropy loss, we can compute the inverse of the Hessian as a diagonal and outer product matrix, so this is also not a problem. For models with a very large number of outputs, such as autoencoders, the matrix $\boldsymbol{H}$ will be fairly large, and so computing its inverse manually can become expensive. However, in the case of autoencoders, the loss is usually the MSE loss, and so has a diagonal Hessian which can be inverted easily.

Thus, if $\boldsymbol{H}$ is positive definite (which is necessary for the Laplace approximation to be valid) and easy to invert, then it should be possible to use the Woodbury matrix identity to compute the inverse of this preconditioner. This preconditioner is fast to compute, but may not be effective when the eigenvectors of the GGN matrix are not constant across the dataset. However, due to the effectiveness of the sub-sampled GGN preconditioner and the assumption of stationary eigenvectors, we did not end up using this preconditioner. Additionally, this preconditioner cannot assume a higher rank than the number of outputs O, since adding more observations to the preconditioner will increase its rank to some unknown value and prevent the application of the Woodbury matrix identity. Ideally, we would like to be able to find a preconditioner for which we can choose the rank of the low-rank approximation $\boldsymbol{A} \approx \sum_{i=1}^{N} \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i$ without simply computing the GGN on a single observation ($\boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i \approx \sum_{i=1}^{N} \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i$), while still allowing us to apply the Woodbury matrix identity, unlike in the case of the sub-sampling preconditioner in Subsection 4.3.2 ($\sum_{i=1}^{B} \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i \approx \sum_{i=1}^{N} \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i$).

4.3.4 Pivoted Cholesky

The pivoted Cholesky factorisation is a method for computing a low-rank approximation to the Cholesky decomposition of a matrix [48, 49]. This decomposition has been successfully applied to preconditioning for kernel matrices in Gaussian processes [50], but not yet to preconditioning for the posterior precision of neural networks using the Laplace approximation. Given the posterior precision matrix $\boldsymbol{\Lambda}$ from the Laplace

approximation, we can compute the pivoted Cholesky factorisation as

$$\begin{aligned}\boldsymbol{\Lambda} &= \sum_{i}^{N} \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i + \alpha \boldsymbol{I} \\ &\approx \boldsymbol{L}^\intercal \boldsymbol{L} + \alpha \boldsymbol{I},\end{aligned} \tag{4.23}$$

where $\boldsymbol{L} \in \mathbb{R}^{K \times D}$, with K being the chosen rank of the pivoted Cholesky factorisation and D being the number of parameters. We can then use this approximation as a preconditioner by computing its inverse via the Woodbury matrix identity, similarly to Subsection 4.3.3, as

$$\boldsymbol{P} = \boldsymbol{L}^\intercal \boldsymbol{L} + \alpha \boldsymbol{I}, \tag{4.24}$$

$$\boldsymbol{P}^{-1} = \alpha^{-1}\boldsymbol{I} - \alpha^{-2}\boldsymbol{L}^\intercal \left(\boldsymbol{I} + \alpha^{-1}\boldsymbol{L}\boldsymbol{L}^\intercal\right)^{-1} \boldsymbol{L}. \tag{4.25}$$

We then obtain the preconditioner-vector and inverse preconditioner-vector products as in Equation 4.26. The most expensive computation in the inverse preconditioner is inverting a $K \times K$ matrix, where K is the rank of our pivoted Cholesky factorisation (which we choose).

$$\boldsymbol{P}\boldsymbol{v} = \boldsymbol{L}^\intercal(\boldsymbol{L}\boldsymbol{v}) + \alpha \boldsymbol{v}, \tag{4.26}$$

$$\boldsymbol{P}^{-1}\boldsymbol{v} = \alpha^{-1}\boldsymbol{v} - \alpha^{-2}\boldsymbol{L}^\intercal(\underbrace{\boldsymbol{I} + \alpha^{-1}\boldsymbol{L}\boldsymbol{L}^\intercal}_{K \times K})^{-1}(\boldsymbol{L}\boldsymbol{v}). \tag{4.27}$$

Here it is possible to precompute the inverse factorised product given by $\boldsymbol{L}^\intercal(\boldsymbol{I} + \alpha^{-1}\boldsymbol{L}\boldsymbol{L}^\intercal)^{-1} \in \mathbb{R}^{D \times K}$, meaning that we only need to perform two matrix multiplications for both the preconditioner- and inverse preconditioner-vector products. This would then require storing the factorisation $\boldsymbol{L} \in \mathbb{R}^{K \times D}$ and the inverse factorised product $\mathbb{R}^{D \times K}$ matrix.[7]

7: We can also precompute the inverse $(\boldsymbol{I} + \alpha^{-1}\boldsymbol{L}\boldsymbol{L}^\intercal)^{-1} \in \mathbb{R}^{K \times K}$, which means we would only need to store the factorisation $\boldsymbol{L} \in \mathbb{R}^{K \times D}$ and this inverse $K \times K$ matrix, in which case we would just need to perform one more matrix multiplication (multiplying $\boldsymbol{L}^\intercal$ with this $K \times K$ matrix) in every preconditioner call.

As explained in Subsection 2.3.4, as we sum over multiple low-rank GGN matrices, we will likely find that the minimum non-zero eigenvalue gets progressively smaller compared to the largest. The condition number will therefore get progressively worse. The preconditioning of $\sum_{i=1}^{N} \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i$ (which should be low-rank) will progressively become *less* low-rank and its condition number will progressively become higher.

Regarding the non-zero eigenvalues and eigenvectors, we consider a best-case scenario. Assume that the pivoted Cholesky factorisation finds the K largest eigenvectors of the precision matrix. If the eigenspectrum of this factorisation is steep enough, then the largest eigenvalue s_1 will be significantly larger than the smallest eigenvalue s_K (i.e., there will be a steep drop-off from the largest to the smallest non-zero eigenvalue). Since the eigenvectors of each per-observation term $\boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i$ will not be the same, when we sum over them, the eigenvalues of the sum will not be the sum of the eigenvalues of the individual Hessians. However, some of the eigenvectors may be close to each other, and so the spectrum of the GGN sum may be approximately the sum of the GGN eigenvalues. Nevertheless, the rank of the total Hessian will increase with the number of observations and so the number of non-zero eigenvalues consequently increase as well. These eigenvalues will be smaller than the summed eigenvalues (because the eigenvectors may not be contained in the same subspace), but they will be larger than the smallest eigenvalue of the per-observation Hessian. This will then lead to a steeper spectrum and a larger condition number of the total Hessian. Thus, when the batch

size B is large, the condition number of the sum of GGN Hessians will increase and the preconditioner will become less effective. How much the condition number increases with the batch size depends on the eigenspectrum of the per-observation Hessians (i.e., how close the eigenvectors are to each other).

If we want to use the pivoted Cholesky decomposition to approximate $\sum_i^B \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i$, we are largely trying to approximate the higher eigenvectors of the sum. In order for this to be the case, we will want to oversample vectors from this matrix which correspond to the largest eigenvalues. A common approach to sampling the largest eigenvalues is to sample vectors with probability proportional to the diagonal value [48]. However, it is not possible to generally and efficiently determine the diagonal of a matrix via matrix-vector products. Often, it is necessary to multiply a vector of zeros with a value of one at the index at which to find the diagonal. This operation needs to be repeated as many times as there are elements in the diagonal with no shared computation between each element, which is very expensive. Alternatively, we can approximate the diagonal of the GGN matrix as the element-wise product of the gradient with itself, similarly to the preconditioner described in Subsection 4.3.1 and the GGN approximation computed in Equation 2.25, though this may not be the most accurate approximation.

In conclusion, to find the optimal parameters for the pivoted Cholesky preconditioner (the number of observations B and the rank of the pivoted Cholesky factorisation K), there are some rules of thumb to consider. Notably, the quality of the preconditioner depends largely on the ease of inverting the $\boldsymbol{I} + \alpha^{-1}\boldsymbol{L}\boldsymbol{L}^\intercal$ matrix, which depends on its condition number. In practice, increasing the rank of the pivoted Cholesky factorisation will increase the condition number of this matrix, since we are approximating more of the largest eigenvectors and so we will capture progressively smaller eigenvalues.

We have found that it is most effective to keep K low, such that the $K \times K$ outer product of the pivoted Cholesky factorisation matrix L approximates few enough of the largest eigenvectors that the eigenvalue of the largest is close to that of the smallest (meaning this outer product is well-conditioned). We thus need to choose K such that it is large enough to approximate enough large eigenvectors and small enough to ensure the condition number of the outer product is low. Furthermore, you want enough observations B to be included in the preconditioner such that it is possible to accurately approximate the eigenvectors of the problem—it may be optimal to use the whole dataset, though this may worsen the conditioning of the outer product.

Furthermore, the pivoted Cholesky decomposition is not implemented in JAX. Since the pivoted Cholesky decomposition involves incrementally adding columns to the factorisation, and JAX is not friendly towards mutable variables, this preconditioner is not straightforward to implement. Additionally, the pivoted Cholesky decomposition traditionally uses the diagonal of the matrix to determine which columns to add to the factorisation, which is difficult without instantiating the matrix, so we would need to approximate the diagonal of the GGN matrix.

Table 4.1: Comparison of various approximations of the likelihood contribution $\sum_i^N \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i$ to the Laplace posterior precision for use in preconditioning. We compare the rank of the approximation, the speed of computing preconditioner-vector products, the closeness of the approximation, and the method used for inverting the preconditioner.

	Rank	Speed	Closeness	Inversion
$\boldsymbol{\nabla}^\intercal \boldsymbol{\nabla}$	1	Very fast	Bad	Closed-form
$\sum_i \boldsymbol{J}_i^\intercal \boldsymbol{H}_i \boldsymbol{J}_i$	Flexible, implicit	Slow	Arbitrarily close	Conjugate Gradient
$\boldsymbol{J}^\intercal \boldsymbol{H} \boldsymbol{J}$	O	Fast	Okay	Woodbury
$\boldsymbol{L}^\intercal \boldsymbol{L}$	Flexible, explicit	Fast (precomputable)	Good	Woodbury

4.3.5 Other Preconditioners

The randomly pivoted Cholesky [51] preconditioner is simple and inexpensive variant of the partial Cholesky decomposition family of algorithms. Similarly to the pivoted Cholesky preconditioner, the randomly pivoted Cholesky preconditioner requires the diagonal of the GGN matrix in order for the decomposition to be close to the GGN. We can thus approximate this term as we do in the pivoted Cholesky preconditioner (see Subsection 4.3.4). Early tests of the randomly pivoted Cholesky preconditioner were not successful, and so it was not pursued further.

Additionally, we considered potential preconditioners to be computed as traditional approximations of the posterior precision, like the diagonal approximation or the Kronecker factorisation. These methods have been shown to be relatively effective approximations of the posterior precision in the context of Bayesian neural networks [8], and so they may contain enough information about the full posterior precision to be effective preconditioners.

However, due to the speed, effectiveness, and theoretical properties of other preconditioners proposed in this section, these methods were not used. Overall, both the sub-sampled preconditioner and the pivoted Cholesky preconditioners were found to be effective under the correct settings. However, the pivoted Cholesky preconditioner is not easy to implement in JAX and was more difficult to tune than the sub-sampled preconditioner. The sub-sampled preconditioner is thus the preferred preconditioner in this project. We provide a high-level comparison of the various preconditioners we have presented in this chapter in Table 4.1.

4.4 Sampling Evaluation

As we have explained, sampling through CIQ is sensitive to the condition number of the posterior precision. As such, we need a technique for evaluating the quality of the samples drawn from the approximate posterior to determine if sampling occurred correctly. This means evaluating whether the samples are drawn from a normal distribution with the correct mean and precision.

What we would like to fundamentally test is whether the CIQ algorithm converged correctly and thus whether the samples were generated successfully. However, for cases when sampling fails to converge, this is typically caused by ill-conditioning of the posterior precision matrix. Since we only have access to the precision matrix and computing the

covariance involves inverting the precision matrix using, for example, the conjugate gradient method (which is itself sensitive to ill-conditioning), we cannot directly evaluate the covariance of the samples. Instead, we require a method to evaluate the quality of the samples without access to the covariance matrix, which is what we will discuss in the next section.

4.4.1 The Chi-Squared Distribution

As per the definition of the chi-squared distribution, for D-dimensional standard normal samples $\boldsymbol{\varepsilon}_0$, the sum of square deviations from the mean is chi-squared distributed with D degrees of freedom, as per

$$\boldsymbol{\varepsilon}_0{}^\mathsf{T}\boldsymbol{\varepsilon}_0 \sim \chi^2(D). \tag{4.28}$$

Samples $\boldsymbol{\varepsilon}$ which are normally distributed but do not have a mean of zero and a standard deviation of one can be standardised and their sum of square deviations be calculated by computing their squared Mahalanobis distances, which will be chi-squared distributed with D degrees of freedom, as per

$$\begin{aligned} \boldsymbol{d} &= \sqrt{(\boldsymbol{\varepsilon}-\boldsymbol{\mu})^\mathsf{T}\boldsymbol{\Lambda}(\boldsymbol{\varepsilon}-\boldsymbol{\mu})} \\ &\Rightarrow \boldsymbol{d}^\mathsf{T}\boldsymbol{d} \sim \chi^2(D). \end{aligned} \tag{4.29}$$

This has the advantage of only requiring the computation of the product of the precision matrix $\boldsymbol{\Lambda}$ with a vector, which we can compute efficiently, as it does not require instantiation of the matrix. For the Laplace approximation, this can easily be computed as $\boldsymbol{\Lambda} = -\nabla^2_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta} \mid \boldsymbol{y})$. Since this precision matrix can be computed exactly, this calculation can be performed to evaluate the whether a set of samples $\boldsymbol{\varepsilon}$ are drawn from a distribution $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Lambda}^{-1})$ while only implicitly accessing this precision matrix in the evaluation via matrix-vector products. To do so, we can compute the squared Mahalanobis distance of each sample $\boldsymbol{\varepsilon}_i$ from the mean $\boldsymbol{\mu}$ and compare it to the chi-squared distribution with D degrees of freedom. We do this by comparing the histogram of these squared Mahalanobis distances to the appropriate PDF and by visualising the sample empirical percentiles against the theoretical percentiles in a quantile-quantile plot. As such, this method can be used to evaluate whether the approximate normal samples obtained from CIQ are correctly distributed. However, this method does not provide a quantitative measure of the quality of the samples. In order to run experiments using the Laplace approximation without manually checking the plots of every set of samples we generate, we also need a method to evaluate the quality of the samples quantitatively. Our current methods do not provide this, and so we need to evaluate the quality of the samples in a different way.

The Kolmogorov-Smirnov test is a non-parametric test for comparing two continuous distributions. It is a test of whether two samples are drawn from the same distribution. Since the chi-squared distribution is continuous, we can use the Kolmogorov-Smirnov test to compare the chi-squared distribution with D degrees of freedom to the squared Mahalanobis distances of the samples. This test is performed by computing the Kolmogorov-Smirnov statistic, which is the maximum difference between the empirical and theoretical CDFs. The null hypothesis of the test is

that the samples are drawn from the same distribution, and so the test is rejected if the Kolmogorov-Smirnov statistic is greater than the critical value for the test. This means that sample normality is rejected if the p-value of the test is less than the significance level of the test. However, this test is very sensitive to the number of samples used to compute the statistic, and so a large number of samples will lead to a very low p-value even if the samples are drawn from the same distribution. Since we can sample an arbitrarily large number of samples from the approximate posterior, this may lead to low p-values even when sampling succeeds.

5 Ablation Experiments

5.1 Hessian-Vector Products

In this project, we propose performing the Laplace approximation by only storing the computational graph for the implicit Hessian-vector product, thereby avoiding the explicit computation and storage of the quadratically-scaling Hessian matrix. How significant of an effect does this have, though? To test this, we perform Hessian-vector products $\boldsymbol{K}\boldsymbol{v}$ and inverse Hessian-vector products $\boldsymbol{K}^{-1}\boldsymbol{v}$, where $\boldsymbol{K} \in \mathbb{R}^{N\times N}$, and compare their performance. Specifically, we apply a manual instantiation and inverse (or solve) of the Hessian matrix with a random vector $\boldsymbol{v}$ and compare this to the performance of the JAX efficient Hessian-vector products (which we will call HVPs) and GGN-vector products (which we will call GVPs). In reality, JAX computes these products implicitly by performing two Jacobian-vector products (for the HVP) and one Jacobian-vector product (for the GVP). To compare performance with regard to memory usage and runtime, we perform these products for an increasing problem dimensionality $N \in \{10^1, 10^2, \ldots, 10^9\}$.

It is difficult to benchmark memory consumption in JAX, since the just-in-time compiler optimises computations using XLA. This then prevents us from stopping the program and measuring memory usage at a single point in time, as many computations will be performed at once. Ideally, it would be possible to measure the *peak* memory usage by JAX during the running of the program. This is not trivial, though, as it would require querying the memory usage from the GPU asynchronously at a very high frequency and "hope" the peak usage point was captured. Instead, we simply measure the value of N at which the program fails from an out-of-memory error instead, as a proxy for memory usage. This experiment should then answer two questions:

1. How much larger problems can we solve if we do not need to instantiate the whole Hessian?
2. How much faster is it to compute the GVP versus, say, the HVP or the explicit Hessian-vector product?

To perform this experiment, we define the the loss function composition $\mathcal{L}(\boldsymbol{x}) \equiv (g \circ f)(\boldsymbol{x}) \coloneqq \sum_{i=1}^{N}(2x_i + 0.5)^2$ as $f(\boldsymbol{x}) \coloneqq 2\boldsymbol{x} + 0.5$, $g(\boldsymbol{x}) \coloneqq \sum_{i=1}^{N} x_i^2$. This is equivalent to the sum-of-square-error loss g for a simple linear model f. Since $f(x)$ is a linear function, the GGN matrix will exactly equal the actual Hessian.[1] We then initialise $\boldsymbol{x}$ to a random vector and compute the matrix-vector product $\boldsymbol{K}\boldsymbol{v}$ where $\boldsymbol{v}$ is a random vector using three Hessian-vector products: the JAX efficient HVP and GVP functions and a manual Hessian instantiation and multiplication. We then measure the total wall-clock time for computing this Hessian product by $\boldsymbol{v}$.

Furthermore, since we are looking to compute some variation of the matrix inverse of $\boldsymbol{K}$, and inverting a high-dimensional matrix can also be very expensive, we also compare the performance of the *inverse* Hessian-vector product $\boldsymbol{K}^{-1}\boldsymbol{v}$. To compute the inverse for the efficient implicit HVP

1: This allows us to compare our implementation of the GGN-vector product to the Hessian-vector product. While this test is not perfect (since it only holds for linear functions), it is a good sanity check. We find that the two are indeed the same in these experiments.

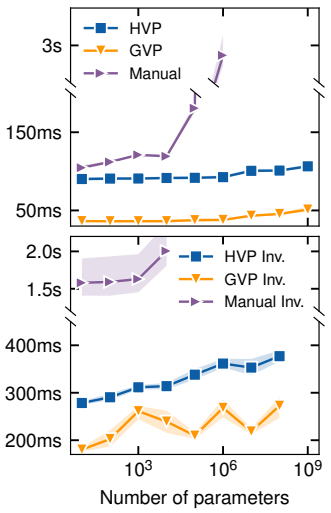


Figure 5.1: Top: comparison of the performance for manual Hessian-vector products, HVPs, and GVPs for increasing numbers of parameters D. Bottom: comparison of the performance of inverse-vector products for the same methods. Implicit inverse is performed via conjugate gradient for HVP and GVP. We can see that the explicit inversion fails for $D > 10k$ and that explicit Hessian-vector products fail for $D > 1M$ (and, for $D = 1M$, the manual product is extremely slow). Implicit products, however, can be computed so long as the output vector fits in memory, while implicit inversion fails for $D = 1B$. Notice the broken axis and thus, in particular, the performance difference between the explicit and implicit inversion methods.

and GVP functions, we use the conjugate gradient method, which only requires the use of matrix-vector multiplications to compute this inverse product. To compute the explicit inverse, we use the `jax.scipy.linalg` package, which performs the inverse using LAPACK (on CPU) or cuSOLVER (on GPU). These experiments were run on a single NVIDIA A100 GPU.

The results for this experiment have been visualised in Figure 5.1. Additionally, the raw results for this experiment have been summarised in the appendix, in Table 1. We can see that the explicit Hessian-vector product fails for $D > 1M$ parameters, while the explicit inversion fails for $D > 10k$ parameters. This is not surprising, as the explicit Hessian is a $D \times D$ matrix, and so the memory requirements for this matrix depend on the D^2 parameters. Additionally, while the wall clock time for computing the Hessian-vector product is sublinear with D, explicit Hessian-vector product is over an order of magnitude slower for $D = 1$M than for $D = 100$k. This may be due to the explicit Hessian, at this size, being too large to fit in the cache of the GPU.

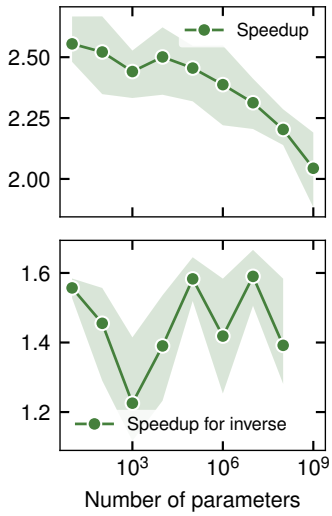


Figure 5.2: Comparison of the speedup from computation of the Hessian-vector product (HVP) versus the GGN-vector product (GVP) (top) and inverse HVP versus inverse GVP using the conjugate gradient method (bottom). Speedup is calculated as the ratio of the wall-clock time of the HVP to the GVP. We can see that the two methods have the same time complexity (up to a linear factor).

The largest number of parameters tested is 1 B. Above 1 B parameters, even implicit methods fail, although this failure occurs in the instantiation of a single random vector of that size (so it would fail on this hardware for any method, since we can not even store the v nor the product $\boldsymbol{K}v$). Notably, however, these implicit Hessian-vector product methods allowed for computation of matrix products for problems three orders of magnitude larger, and computation of solves for problems four orders of magnitude larger, than the explicit methods. This is a significant improvement in scalability, and is a key reason why we use these methods in our experiments. Additionally the implicit inverse products are significantly faster than explicit inversion, even for small problems. Performance also appears identical regardless of whether the explicit inverse is computed by actual inversion or by solving a system of linear equations.

However, due to the scale of the figure, it is not clear how the performance of the Hessian-vector product compares to the performance of the GGN-vector product. We can then plot the speedup from going from the HVP to the GVP as a function of the number of model parameters. This can be calculated as the ratio of the wall-clock time of the HVP to the wall-clock time of the GVP. Thus, larger values suggest greater performance gains for the GVP, while values closer to one suggest there is no performance gain. These results can be seen in Figure 5.2. The number of parameters on the x-axis are exponentially increasing. We can see that the computation of the GVP is always faster than the computation of the HVP, and that the speedup is approximately linear in the number of parameters. This is in line with what we expected (see Subsection 2.3.3). We can also see a slight decrease in the speedup as the number of parameters increases for the Hessian-vector products. However, it is not clear why this is the case.

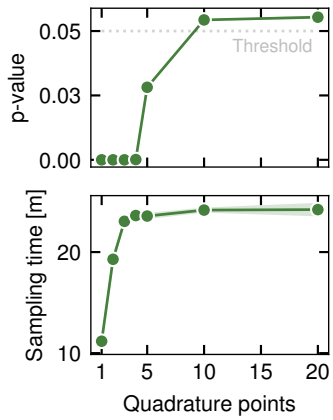


Figure 5.3: Ablation over quadrature points for MNIST. We can see that sampling fails unambiguously for $Q < 5$ and succeeds for $Q \geq 10$.

5.2 Sampling Ablation

Contour integral quadrature (CIQ) converges to the exact solution for $\lim_{Q\to\infty} \boldsymbol{s}_Q^{\text{CIQ}} = \boldsymbol{K}^{-1/2}\boldsymbol{v}$. However, CIQ is typically close to the exact

solution for low values of Q.[2] We now attempt to determine the number of quadrature points Q required to compute an adequate posterior samples.

We vary $Q \in \{1, \ldots, 20\}$ to determine the number of quadrature points required to effectively sample from the Laplace approximation. We then compute the Mahalanobis distance between the samples and the true posterior mean, and compare this to the chi-squared distribution with D degrees of freedom. By comparing the empirical CDF of the Mahalanobis distances to the theoretical χ^2_D CDF, we can perform the Kolmogorov-Smirnov test for each value of Q. This p-value is then used to determine whether the null hypothesis that the samples are multivariate normal can be rejected based on some significance threshold. Sample evaluation is explained in more detail in Section 4.4. The results of this experiment are shown in Table 5.1 and Figure 5.3. Quantile–quantile plots for each value of Q are shown in the appendix, in Figure 2. These experiments were, again, run on a single NVIDIA A100 GPU.

The p-values in Table 5.1 suggest that sampling is successful for $Q \geq 10$ with a significance threshold of 0.05. However, significance testing can be sensitive to the number of samples used, and, for large numbers of samples, these tests can excessively reject the null hypothesis. In our case, the number of samples is generally fixed as a function of the number of parameters in the model and the amount of memory available. As such, it is important to select an appropriate significance threshold for the test based on the number of posterior samples that are required. Inspection of Figure 2 in the appendix suggests that there is not a significant difference between using 5, 10, or 20 quadrature points. Using less than 4 quadrature points, however, results in sampling failure, while using exactly 4 quadrature points results in a borderline success. For 200 posterior samples, a more appropriate significance threshold of 0.01 or lower would reject sampling for $Q \leq 4$, since the cases where sampling fails completely have p-values that are below single floating point precision (i.e., they are effectively zero).

In taking as much as 20 to 30 minutes to compute 200 posterior samples for an MNIST model with 15 k parameters, the time required to compute the samples is not negligible. In practice, it is likely that only a small number of samples will be obtained (possibly around 5–15), since for each sample it is necessary to store a parameter vector of size D. However, the sampling procedure still takes over 20 minutes to compute 10 samples. As such, the time required to compute the samples may be a significant factor in the overall time required to compute the Laplace approximation. In particular, for models with a very large number of parameters, the time required to compute the samples may be prohibitive when compared when the near-instant time required to sample from the diagonal Laplace approximation. Even if the sampling time scales sublinearly with the number of parameters, it would still prevent the application in Laplace methods that require the Laplace approximation to be computed at each iteration from being used in practice. However, this sampling time constraint is also significant for MCMC methods, which are the most common method for sampling from the posterior distribution of a BNN.

2: Pleiss et al. [44] find that $Q = 20$ is approximately sufficient for most problems.

Table 5.1: Sampling time and p-values for Kolmogorov-Smirnov with varying quadrature points Q on MNIST with prior precision $\alpha = 0.1$, for 200 posterior samples. Lower p-value indicates a higher probability of rejecting normality. Bold indicates sampling was successful based on visual inspection of quantile–quantile plots. Since, for values of $Q \geq 5$, samples appear visually to be multivariate normal, we can conclude that we should choose a significance threshold of 0.01 or lower.

Q	p-value	Time [m]
1	0.0000	10.6
2	0.0000	18.8
3	0.0000	23.4
4	0.0001	23.4
5	**0.0284**	23.9
10	**0.0542**	24.4
20	**0.0561**	24.9

Results for the experiments in this chapter run on CPU are shown in the appendix, in Figure 1.

6 Experiments

In Chapter 5 we tested whether the approximate sampling scheme is able to produce correct posterior samples on an arbitrary network. In this chapter, we use our custom bound on the marginal likelihood to train a Bayesian neural network on the marginal via online Laplace. We then use our approximate sampling scheme to sample from the posterior and propagate the samples through the network to produce a predictive distribution. In this way, we test the entire Bayesian neural network pipeline, from training to inference.

6.1 The Sine Function

We first use a simple sine function as a toy example to demonstrate the Bayesian neural network pipeline for regression. First, we must train the network on the marginal likelihood to demonstrate that the model can learn a good posterior distribution. We then sample from the posterior and visualise the posterior predictive samples to showcase the quality of the predictive distribution.

In Figure 6.1 we show the posterior predictive distribution for a fully connected neural network modelling a sine curve $f(x) = \sin(5x + 1)$ and compare it to a neural network trained on the MSE loss, i.e., maximum likelihood for regression. We vary the prior precision α to show how the posterior predictive distribution changes. We have the same prior precision for all weights and biases, with values $\alpha = 70$ for the top figure and $\alpha = 1$ for the bottom figure. Figure 3 in the appendix shows the posterior predictive samples for this experiment.

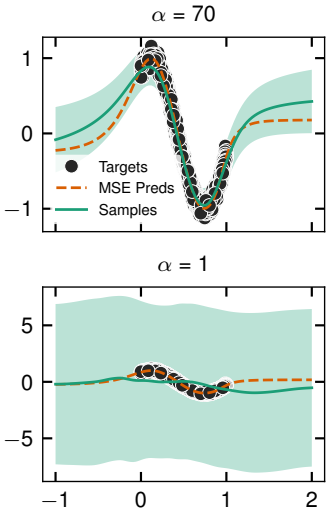


Figure 6.1: Laplace predictive posterior (with a confidence interval given by one standard deviation) for a multilayer perceptron modelling $f(x) = \sin(5x + 1)$ with parameters trained on the marginal likelihood for a fixed prior precision $\alpha = 70$ (top) and $\alpha = 1$ (bottom). We compute the marginal using our approximate lower bound and sample from the posterior using our approximate sampling scheme. The choice of prior precision has a large effect on the posterior uncertainty, but, when it is chosen appropriately, our approximate methods are able to learn a reasonable predictive distribution (i.e., more uncertainty in the regions with no data) with a similar mean prediction to the MSE-optimised network.

We can see from Figure 6.1 that for $\alpha = 70$ the posterior predictive distribution has a low uncertainty in the regions with data, but a high uncertainty in the regions without data. Furthermore, for this value of α the mean of the posterior predictive distribution is relatively close to the mean of the MSE-optimised network.

For a very low prior precision $\alpha = 1$, the posterior predictive distribution has a high uncertainty throughout the whole function domain. This is because the prior uncertainty is very high, so the posterior uncertainty is also high. Furthermore, because of the noise in the posterior samples, the mean of the posterior predictive distribution is not close to that of the MSE-optimised network.

We can thus see that maximising the marginal likelihood is able to learn a reasonable predictive distribution (i.e., more uncertainty in the regions with no data) with a similar mean prediction to the MSE-optimised network, when the prior precision is chosen appropriately. However, if the prior precision is too low, the posterior predictive distribution will have an extremely high uncertainty and the mean of the posterior predictive distribution will not be close to that of the MSE-optimised network. In these experiments, we have chosen the prior precision to be

fixed, but in practice, we would choose the prior precision by maximising the marginal likelihood or by cross-validation.

Figure 6.2 show the loss curves from training the model parameters and prior precision with initialisation $\alpha = 70$ on the negative log-marginal likelihood loss $-\log p(\boldsymbol{y})$ for our sine function. First, we train the model parameters on the MSE loss for 10 epochs, and then start training on the marginal. As per Equation 3.5, the Laplace-approximate negative log-marginal likelihood loss during training can be decomposed as

$$-\log p(\boldsymbol{y}) \overset{\text{LA}}{\approx} -\log p(\boldsymbol{y} \mid \boldsymbol{\theta}) - \log p(\boldsymbol{\theta}) + \frac{1}{2} \log \det \boldsymbol{\Lambda} + \text{const.} \tag{6.1}$$

These terms are shown in Figure 6.2 (top), where the negative log-likelihood $-\log p(\boldsymbol{y} \mid \boldsymbol{\theta})$ and negative log-marginal $-\log p(\boldsymbol{y})$ terms are both strongly smoothed by a moving average. We can see that all three terms (and therefore also the negative log-marginal loss itself) decrease during training. The negative log-likelihood term dominates the overall decrease in the negative log-marginal. Since we are optimising the prior precision hyperparameter α with the marginal likelihood, we also plot its variation throughout training in Figure 6.2 (bottom). We see that α decreases linearly during training. In practice, if we keep training, we find that α reaches a minimum value below 1. As we showed in Figure 6.1, this is a poor prior precision, since it leads to an unreasonably high posterior predictive uncertainty. However, training the hyperparameters on the marginal likelihood is a sensitive procedure, and so we have not yet managed to successfully optimise the prior precision. Because of this, we choose a fixed prior precision in our other experiments.

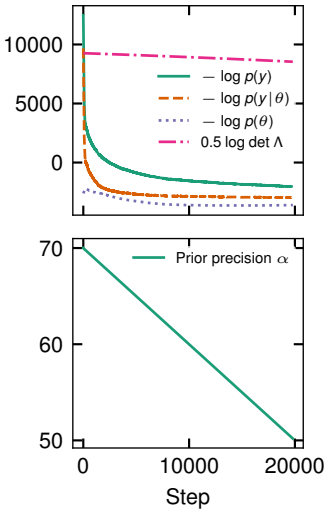


Figure 6.2: Loss curves (top) from training model parameters and prior precision with initialisation $\alpha = 70$ (bottom) on the negative log-marginal likelihood loss $-\log p(\boldsymbol{y})$ with a 10 epoch warmup period of training on MSE loss. Loss curves show the individual terms that make up the negative log-marginal likelihood loss, as per Equation 6.1. Negative log-marginal likelihood (in green) and negative log-likelihood (in orange) are both strongly smoothed by a moving average. All four terms can be seen to decrease during training, with the negative log-likelihood term dominating the decrease. We see that the prior precision α decreases linearly during training.

In Figure 6.3 we show the chi-squared sample plot for the posterior samples on this sine regression task to demonstrate that the posterior samples are multivariate normal with the correct covariance, as we explain in Section 4.4. We choose a low value for the prior precision of $\alpha = 1$ to demonstrate that our sampling works even when the condition number of the posterior precision is high. Figure 6.3 shows that the sample square Mahalanobis distances for these experiments appear chi-squared distributed, so the samples themselves are multivariate normal with the Laplace posterior precision. We deliberately choose an extremely large number of posterior samples (1000) to demonstrate the accuracy of the posterior samples. However, it would not be practical to compute this many posterior samples for larger problems, since we would have to either store them all in memory or compute them in batches.

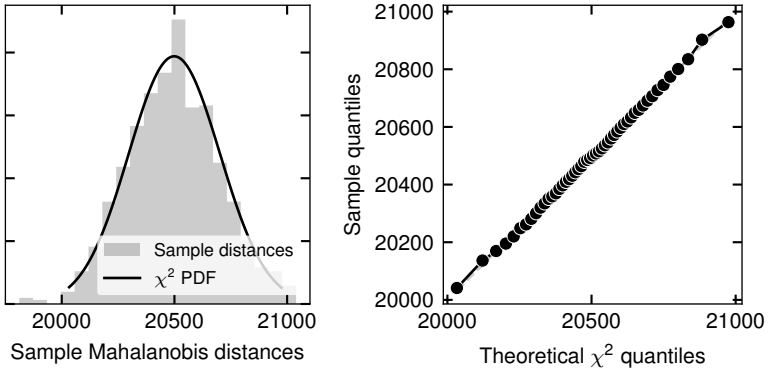


Figure 6.3: Chi-squared sample plot for visualising normality of 1000 posterior samples from the full Laplace approximation for a fully connected neural network modelling $f(x) = \sin(5x + 1)$, using CIQ. Since the sample square Mahalanobis distances appear chi-squared distributed, the samples themselves are multivariate normal with the correct (known) covariance.

6.2 MNIST

We now attempt to use our approximate sampling method on a more realistic problem, namely, the classification of handwritten digits from the MNIST dataset. To demonstrate that our approximate sampling method can be used on a more complex network, we train a convolutional neural network on MNIST to compute the Laplace approximation. Currently, the inclusion of convolutions in the network causes a major slowdown in our approximate log-marginal training procedure, so we maximise the posterior during training (i.e., performing post-hoc Laplace) instead of the marginal likelihood. Because of this, these experiments have been performed with a fixed posterior precision α which may not be the best choice for the posterior precision.

Unfortunately, evaluating the quality of the posterior predictive distribution for MNIST is not as straightforward as for the sine curve. The reason for this is that we cannot easily visually determine the appropriate magnitude of the uncertainty in the posterior predictive distribution for MNIST (since the output is a 10-dimensional vector). We instead only evaluate the quality of the posterior distribution by comparing the posterior samples to the theoretical posterior distribution, as we did for the sine function in Figure 6.3. For the sine function, we sampled 1000 posterior samples. Since, in practice, we would not be able to store this many posterior samples for large-scale problems, we instead sample 200 parameter vectors from the posterior for MNIST, to demonstrate what these plots look like for a more realistic number of samples. In reality, for large-scale problems, we would not be able to store even 200 posterior samples, but between 5 and 20. We can see these results with $\alpha = 0.1$ in Figure 6.4.

Figure 6.4 shows that sampling succeeds for values of α as low as 0.1. This value of the prior precision is much lower than the value of α we would likely use in practice. Additionally, as we noted in Subsection 2.3.4, since α is the lowest eigenvalue of the posterior precision, the condition number is inversely proportional to α. Because of this, by sampling with a low value of α, we are sampling from the most ill-conditioned posterior precision we can expect to encounter. As such, this experiment allows us to conclude that our approximate sampling method would successfully be able to sample from the posterior distribution for a wide range of values of α on MNIST without being too ill-conditioned for our method.

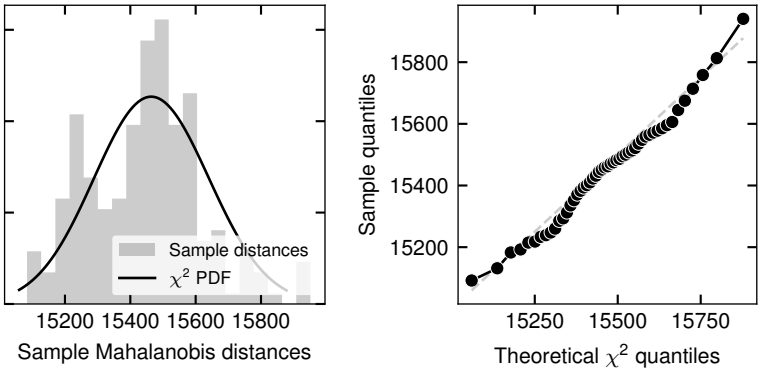


Figure 6.4: Chi-squared sample plot for visualising normality of 200 posterior samples from the full Laplace approximation for a convolutional neural network trained on MNIST with prior precision $\alpha = 0.1$, using CIQ. Though the sample square Mahalanobis distances do not appear *exactly* chi-squared distributed, this is because of the relatively low sample size used to compute the empirical histogram and quantiles. Thus, the plots themselves strongly suggest that the samples are multivariate normal with the correct (known) covariance.

6.3 Discussion

The choice of prior precision α is non-trivial. Furthermore, since the prior precision is the lowest eigenvalue of the posterior precision matrix, we must choose α to be large enough to ensure that the posterior precision matrix is positive definite (i.e., the prior precision must counteract small numerical issues which may lead to some negative eigenvalues). Additionally, the condition number of the posterior precision matrix is proportional to α^{-1}, so α should be large enough to ensure that the posterior precision matrix is well-conditioned. While we can "choose" α to be a value large enough to reduce conditioning issues, this somewhat invalidates the Bayesian interpretation of the prior, which should theoretically be chosen based on prior knowledge of the problem, can also be learned from the data. Choosing the prior in order to reduce ill-conditioning of our posterior precision means we sacrifice some of the benefits of the prior. As such, we would like to learn α from the data by maximising the marginal likelihood. Current issues with training prevent us from reliably learning α from the data across models of different sizes.

Conclusion 7

7.1 Summary and Key Results

In this project, we aimed to develop a scalable full Laplace approximation for Bayesian neural networks. In doing so, we developed a novel method for approximately sampling from the full Laplace posterior and methods for preconditioning the Hessian of the negative log-posterior, which involved attempting to better understand the spectrum of the generalised Gauss-Newton Laplace posterior precision. To test the quality of our sampling procedure, we showed how to evaluate the quality of the samples it produces while only accessing the samples themselves and the desired posterior precision, enabling us to choose a suitable preconditioner for it. This evaluation method showed that, with an appropriate preconditioner, we can reliably produce samples which are Gaussian-distributed with the correct mean and covariance, as desired. As such, we have shown that our approximate sampling method is a valid (albeit slow) method for sampling from the full Laplace posterior.

We also developed a novel method for training the full Laplace approximation by maximising the Laplace-approximate marginal likelihood of the data. To compute this evidence term, we developed a method for computing an upper bound on the log-determinant of the Laplace posterior precision without storing the posterior precision itself. This method allows us to compute a differentiable log-marginal likelihood bound, which we can train with standard gradient descent optimisation methods. We showed that this method is a valid method for training the full Laplace approximation, and that it can be used to train the full Laplace approximation on a convolutional neural network. However, we also found that the marginal training procedure to optimise the hyperparameters of the full Laplace approximation is not yet feasible, since the hyperparameters converge to unreasonable values.

Finally, we showed that it is possible to perform the full Laplace approximation with marginal likelihood training, and sample from its posterior. We demonstrated on a simple sine wave regression problem and an MNIST classification problem that our method can be used to train the full Laplace approximation and yield predictive performance which rivals deterministic training. However, we found that the sampling procedure is too slow to sample outside of a post-hoc setting, and may become a significant bottleneck for inference even under post-hoc Laplace on large-scale problems.

Overall, while we have developed novel methods for training and inference in the full Laplace approximation, we have yet to demonstrate its improved performance over the diagonal Laplace approximation nor its practical application to large-scale problems.

7.2 Outlook and Future Developments

As mentioned in the previous section, we have not yet demonstrated the practicality of our method for full Laplace approximation, particularly for large-scale problems. Hyperparameter selection typically requires performing a grid search over a range of hyperparameters, which is computationally expensive. As such, the practicality of our method depends on the ability to perform marginal training to optimise hyperparameters via backpropagation. Additionally, marginal training is prohibitively slow for training convolutions, and we have not yet found a way to speed it up. Furthermore, estimation of the rank of the posterior precision is important to guarantee the closeness of the upper bound on the log-determinant term in the marginal, and this is still an open problem. This is closely related to the fact that we do not have a solid understanding of how much the eigenvectors of the posterior precision matrix vary across different observations nor what affects the shape of the spectrum of this Hessian. This point is also related to the structure of the Hessian in the Laplace approximation more generally. As such, there is still much work to be done to understand the methods we have developed and to improve them.

Since we have not yet been able to train hyperparameters on the log-marginal likelihood, we have not yet been able to perform the full Laplace pipeline, which includes training the model parameters and hyperparameters and sampling from the posterior. Because of this, we have not yet run experiments which attempt to demonstrate that the full Laplace approximation shows improved predictive performance or uncertainty calibration when compared to the diagonal or Kronecker-factored Laplace approximations. Furthermore, the full Laplace approximation should be compared to other methods for Bayesian inference, such as variational approximations, deep ensembles, Monte Carlo dropout, and Markov chain Monte Carlo—both in terms of performance and runtime.

Furthermore, the feasibility of the full Laplace approximation must be demonstrated on a large-scale problem. It is not clear how our method for the full Laplace approximation will scale with the number of parameters, data points, and model outputs. This concerns not only the actual predictive performance and uncertainty estimation of the method, but also the post-hoc inference time for the approximate posterior. As such, important experiments must be performed to determine the scalability of the full Laplace approximation. This includes experiments on a large-scale problem, such as a large-scale image classification problem on a dataset with hundreds of thousands to millions of rows, with significantly higher resolution than the MNIST dataset, and more than 10 classes, such as in the case of zero-shot learning. These experiments would thus also require models with many parameters, such as a ResNet.

Bibliography

[1] Laurent Valentin Jospin et al. 'Hands-on Bayesian Neural Networks—A Tutorial for Deep Learning Users'. In: *IEEE Computational Intelligence Magazine* 17.2 (2022), pp. 29–48 (cited on page 1).

[2] Qing Rao and Jelena Frtunikj. 'Deep Learning for Self-Driving Cars: Chances and Challenges'. In: *Proceedings of the 1st International Workshop on Software Engineering for AI in Autonomous Systems*. 2018, pp. 35–38 (cited on page 1).

[3] Justin Ker et al. 'Deep Learning Applications in Medical Image Analysis'. In: *IEEE Access* 6 (2017), pp. 9375–9389 (cited on page 1).

[4] Rodolfo C Cavalcante et al. 'Computational Intelligence and Financial Markets: A Survey and Future Directions'. In: *Expert Systems with Applications* 55 (2016), pp. 194–211 (cited on page 1).

[5] Yarin Gal and Zoubin Ghahramani. 'Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning'. In: *International Conference on Machine Learning*. 2016 (cited on page 2).

[6] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. 'Simple and Scalable Predictive Uncertainty Estimation Using Deep Ensembles'. In: *Advances in Neural Information Processing Systems*. 2017 (cited on page 2).

[7] Wesley J Maddox et al. 'A Simple Baseline for Bayesian Uncertainty in Deep Learning'. In: *Advances in Neural Information Processing Systems*. 2019 (cited on pages 2, 15).

[8] Erik Daxberger et al. 'Laplace Redux-Effortless Bayesian Deep Learning'. In: *Advances in Neural Information Processing Systems*. 2021 (cited on pages 2, 8, 9, 16, 32).

[9] Pierre-Simon Laplace. 'Mémoire Sur La Probabilité Des Causes Par Les Événements'. In: *Mém. De Math. Et Phys. Présentés à l'Acad. Roy. Des Sci* 6 (1774), pp. 621–656 (cited on pages 2, 8).

[10] Muhammad Izzatullah et al. 'Laplace HypoPINN: Physics-Informed Neural Network for Hypocenter Localization and Its Predictive Uncertainty'. In: *Machine Learning: Science and Technology* 3.4 (2022), p. 045001 (cited on page 2).

[11] Marco Miani et al. 'Laplacian Autoencoders for Learning Stochastic Representations'. In: *Advances in Neural Information Processing Systems*. 2022 (cited on pages 2, 9).

[12] Yann LeCun, John Denker, and Sara Solla. 'Optimal Brain Damage'. In: *Advances in Neural Information Processing Systems*. 1989 (cited on page 2).

[13] John Denker and Yann LeCun. 'Transforming Neural-Net Output Levels to Probability Distributions'. In: *Advances in Neural Information Processing Systems*. 1990 (cited on page 2).

[14] Tom Heskes. 'On "Natural" Learning and Pruning in Multilayered Perceptrons'. In: *Neural Computation* 12.4 (2000), pp. 881–901 (cited on page 2).

[15] James Martens and Roger Grosse. 'Optimizing Neural Networks With Kronecker-Factored Approximate Curvature'. In: *International Conference on Machine Learning*. 2015 (cited on pages 2, 10).

[16] Aleksandar Botev, Hippolyt Ritter, and David Barber. 'Practical Gauss-Newton Optimisation for Deep Learning'. In: *International Conference on Machine Learning*. 2017 (cited on page 2).

[17] Hippolyt Ritter, Aleksandar Botev, and David Barber. 'A Scalable Laplace Approximation for Neural Networks'. In: *International Conference on Learning Representations*. Vol. 6. 2018 (cited on page 2).

[18] Hippolyt Ritter, Aleksandar Botev, and David Barber. 'Online Structured Laplace Approximations for Overcoming Catastrophic Forgetting'. In: *Advances in Neural Information Processing Systems*. Vol. 31. 2018 (cited on page 2).

[19] James Bradbury et al. *JAX: Composable Transformations of Python+NumPy Programs*. Version 0.3.13. 2018 (cited on page 3).

[20] Amit Sabne. 'XLA: Compiling Machine Learning for Peak Performance'. In: (2020) (cited on page 3).

[21] Diederik P Kingma and Jimmy Ba. 'Adam: A Method for Stochastic Optimization'. In: *International Conference on Learning Representations*. 2015 (cited on page 6).

[22] Boris T Polyak. 'Some Methods of Speeding Up the Convergence of Iteration Methods'. In: *USSR Computational Mathematics and Mathematical Physics* 4.5 (1964), pp. 1–17 (cited on page 6).

[23] Ilya Sutskever et al. 'On the Importance of Initialization and Momentum in Deep Learning'. In: *International Conference on Machine Learning*. 2013 (cited on page 7).

[24] Magnus Rattray, David Saad, and Shun-ichi Amari. 'Natural Gradient Descent for On-Line Learning'. In: *Physical Review Letters* 81.24 (1998), p. 5461 (cited on page 7).

[25] Andrew Gelman et al. *Bayesian Data Analysis*. Chapman and Hall/CRC, 1995 (cited on page 8).

[26] Michael I Jordan et al. 'An Introduction to Variational Methods for Graphical Models'. In: *Machine Learning* 37 (1999), pp. 183–233 (cited on page 8).

[27] Martin J Wainwright, Michael I Jordan, et al. 'Graphical Models, Exponential Families, and Variational Inference'. In: *Foundations and Trends® in Machine Learning* 1.1–2 (2008), pp. 1–305 (cited on page 8).

[28] Nicki Skafte, Martin Jørgensen, and Søren Hauberg. 'Reliable Training and Estimation of Variance Networks'. In: *Advances in Neural Information Processing Systems*. 2019 (cited on page 8).

[29] Christopher Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995 (cited on page 8).

[30] David MacKay. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003 (cited on page 8).

[31] Alexander Immer et al. 'Scalable Marginal Likelihood Estimation for Model Selection in Deep Learning'. In: *International Conference on Machine Learning*. 2021 (cited on page 9).

[32] Anna Choromanska et al. 'The Loss Surfaces of Multilayer Networks'. In: *Artificial Intelligence and Statistics*. PMLR. 2015, pp. 192–204 (cited on page 9).

[33] Frederik Kunstner, Philipp Hennig, and Lukas Balles. 'Limitations of the Empirical Fisher Approximation for Natural Gradient Descent'. In: *Advances in Neural Information Processing Systems*. 2019 (cited on page 12).

[34] Runa Eschenhagen et al. 'Mixtures of Laplace Approximations for Improved Post-Hoc Uncertainty in Deep Learning'. In: *arXiv preprint arXiv:2111.03577* (2021) (cited on page 14).

[35] Sepp Hochreiter and Jürgen Schmidhuber. 'Flat Minima'. In: *Neural Computation* 9.1 (1997), pp. 1–42 (cited on page 15).

[36] Nitish Shirish Keskar et al. 'On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima'. In: *arXiv preprint arXiv:1609.04836* (2016) (cited on page 15).

[37] Yiding Jiang et al. 'Fantastic Generalization Measures and Where to Find Them'. In: *International Conference on Learning Representations*. 2020 (cited on page 15).

[38] Edwin Fong and Chris C Holmes. 'On the Marginal Likelihood and Cross-Validation'. In: *Biometrika* 107.2 (2020), pp. 489–496 (cited on page 15).

[39] Christopher KI Williams and Carl Edward Rasmussen. *Gaussian Processes for Machine Learning*. Vol. 2. 3. MIT Press Cambridge, 2006 (cited on page 16).

[40] Andreas Damianou and Neil D Lawrence. 'Deep Gaussian Processes'. In: *Artificial Intelligence and Statistics*. PMLR. 2013, pp. 207–215 (cited on page 16).

[41] Zhaojun Bai and Gene H Golub. 'Bounds for the Trace of the Inverse and the Determinant of Symmetric Positive Definite Matrices'. In: *Annals of Numerical Mathematics* 4 (1996), pp. 29–38 (cited on page 17).

[42] Michael F Hutchinson. 'A Stochastic Estimator of the Trace of the Influence Matrix for Laplacian Smoothing Splines'. In: *Communications in Statistics-Simulation and Computation* 19.2 (1990), pp. 433–450 (cited on page 18).

[43] Maxime Vono, Nicolas Dobigeon, and Pierre Chainais. 'High-Dimensional Gaussian Sampling: A Review and a Unifying Approach Based on a Stochastic Proximal Point Algorithm'. In: *SIAM Review* 64.1 (2022), pp. 3–56 (cited on page 22).

[44] Geoff Pleiss et al. 'Fast Matrix Square Roots With Applications to Gaussian Processes and Bayesian Optimization'. In: *Advances in Neural Information Processing Systems*. 2020 (cited on pages 22, 23, 25, 37).

[45] Nicholas Hale, Nicholas J Higham, and Lloyd N Trefethen. 'Computing A^{α}, $\log(A)$, and Related Matrix Functions by Contour Integrals'. In: *SIAM Journal on Numerical Analysis* 46.5 (2008), pp. 2505–2523 (cited on page 23).

[46] David Chin-Lung Fong and Michael Saunders. 'CG Versus MINRES: An Empirical Comparison'. In: *Sultan Qaboos University Journal for Science [SQUJS]* 17.1 (2012), pp. 44–62 (cited on page 25).

[47] Max Woodbury. *Inverting Modified Matrices*. Statistical Research Group, 1950 (cited on page 28).

[48] Helmut Harbrecht, Michael Peters, and Reinhold Schneider. 'On the Low-Rank Approximation by the Pivoted Cholesky Decomposition'. In: *Applied Numerical Mathematics* 62.4 (2012), pp. 428–440 (cited on pages 29, 31).

[49] Francis Bach. 'Sharp Analysis of Low-Rank Kernel Matrix Approximations'. In: *Conference on Learning Theory*. 2013 (cited on page 29).

[50] Jacob Gardner et al. 'GPyTorch: Blackbox Matrix-Matrix Gaussian Process Inference With GPU Acceleration'. In: *Advances in Neural Information Processing Systems*. 2018 (cited on page 29).

[51] Yifan Chen et al. 'Randomly Pivoted Cholesky: Practical Approximation of a Kernel Matrix with Few Entry Evaluations'. In: *arXiv preprint arXiv:2207.06503* (2022) (cited on page 32).

Appendix

Algorithms

Algorithm 3: Multi-Shift Minimum Residual (msMINRES)

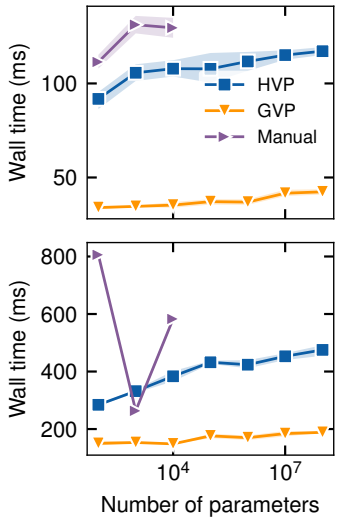
Input : $\boldsymbol{K} \succ 0, \boldsymbol{b}, \boldsymbol{P} \succ 0, t_1, \ldots, t_Q, J > 0$
Output: $\boldsymbol{c}_1 = (\boldsymbol{K} + t_1)^{-1}\boldsymbol{b}, \ldots, \boldsymbol{c}_Q = (\boldsymbol{K} + t_Q)^{-1}\boldsymbol{b}$

$\|\boldsymbol{b}\|_2 \leftarrow (\boldsymbol{b} \cdot \boldsymbol{b})^{1/2}$;
$\boldsymbol{b} \leftarrow \boldsymbol{b}/\|\boldsymbol{b}\|_2$;
$\boldsymbol{p} \leftarrow \boldsymbol{K}\boldsymbol{b}$;
$\boldsymbol{z}_{-1} \leftarrow \boldsymbol{0}$;
$\boldsymbol{z}_0 \leftarrow \boldsymbol{b}$;
$\boldsymbol{q}_0 \leftarrow \boldsymbol{P}\boldsymbol{z}_0$;
$\beta_0 \leftarrow (\boldsymbol{z}_0 \cdot \boldsymbol{q}_0)$;
for $q \leftarrow 1$ **to** Q **do**
  $\boldsymbol{c}_0^{(q)} \leftarrow \boldsymbol{0}$;
  $\boldsymbol{d}_0^{(q)}, \boldsymbol{d}_{-1}^{(q)} \leftarrow \boldsymbol{0}$;
  $\cos_0^{(q)}, \cos_{-1}^{(q)} \leftarrow 1$;
  $\sin_0^{(q)}, \sin_{-1}^{(q)} \leftarrow 0$;
  $\varphi_0^{(q)} \leftarrow \beta_0$;
for $j \leftarrow 1$ **to** J **do**
  $\boldsymbol{p} \leftarrow \boldsymbol{K}\boldsymbol{q}_{j-1}$;
  $\alpha_j \leftarrow \boldsymbol{p} \cdot \boldsymbol{q}_{j-1}$;
  $\boldsymbol{z}_j \leftarrow \boldsymbol{p} - \alpha_j \boldsymbol{z}_{j-1} - \beta_{j-1}\boldsymbol{z}_{j-2}$;
  $\boldsymbol{q}_j \leftarrow \boldsymbol{P}\boldsymbol{z}_j$;
  $\beta_j \leftarrow (\boldsymbol{z}_j \boldsymbol{q}_j)^{1/2}$;
  $\boldsymbol{z}_j \leftarrow \boldsymbol{z}_j/\beta_j$;
  $\boldsymbol{q}_j \leftarrow \boldsymbol{q}_j/\beta_j$;
  for $q \leftarrow 1$ **to** Q **do**
    $\varepsilon_j^{(q)} \leftarrow \sin_{j-2}^{(q)} \beta_{j-1}$;
    $\zeta_j^{(q)} \leftarrow \cos_{j-2}^{(q)} \beta_{j-1}$;
    $\alpha_j^{(q)} \leftarrow \alpha_j + t_q$;
    $\eta_j^{(q)} \leftarrow \cos_{j-1}^{(q)} \alpha_j^{(q)} - \sin_{j-1}^{(q)} \zeta_j^{(q)}$;
    $\zeta_j^{(q)} \leftarrow \cos_{j-1}^{(q)} \zeta_j^{(q)} + \sin_{j-1}^{(q)} \alpha_j^{(q)}$;
    $r_j^{(q)} \leftarrow (\eta_j^{(q)2} + \beta_j^2)^{1/2}$;
    $\cos_j^{(q)} \leftarrow \eta_j^{(q)}/r_j^{(q)}$;
    $\sin_j^{(q)} \leftarrow \beta_j^{(q)}/r_j^{(q)}$;
    $\eta_j^{(q)} \leftarrow \eta_j^{(q)} \cos_j^{(q)} + \sin_j^{(q)} \beta_j$;
    $\varphi_j^{(q)} \leftarrow -\varphi_{j-1}^{(q)} \sin_j^{(q)}$; $\varphi_{j-1}^{(q)} \leftarrow \varphi_{j-1}^{(q)} \cos_j^{(q)}$;
    $\boldsymbol{d}_j^{(q)} \leftarrow (\boldsymbol{q}_{j-1} - \zeta_j^{(q)} \boldsymbol{d}_{j-1}^{(q)} - \varepsilon_j^{(q)} \boldsymbol{d}_{j-2}^{(q)})/\eta_j^{(q)}$;
    $\Delta\boldsymbol{c}_j^{(q)} \leftarrow \boldsymbol{d}_j^{(q)} \varphi_{j-1}^{(q)}$;
    $\boldsymbol{c}_j^{(q)} \leftarrow \boldsymbol{c}_{j-1}^{(q)} + \Delta\boldsymbol{c}_j^{(q)}$;
for $q \leftarrow 1$ **to** Q **do**
  $\boldsymbol{c}_q \leftarrow \boldsymbol{c}_J^{(q)} \|\boldsymbol{b}\|_2$;

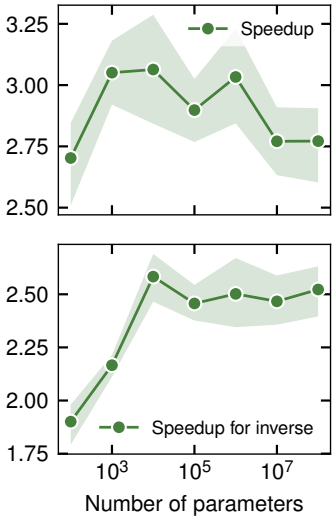
Additional Results

Table 1: Comparison of the wall-clock time for Hessian-vector, GGN-vector, and manual matrix-vector products, as well as the equivalent inverse-vector products, on a toy problem. Time measured in milliseconds, measured on an A100 GPU across 5 runs that are reported as a mean and standard deviation. Implicit inverse performed via conjugate gradient for HVP and GVP. We highlight in bold the extreme slowdown of computing the manual Hessian-vector product for a 1 M-dimensional problem. We can see that manual computations fail above small problems, while the implicit GVP and HVP functions and their inverses scale well. We also see that the GVP is slightly faster than the HVP, both both are significantly faster than the manual Hessian-vector product.

	HVP		GVP		Manual		HVP Inverse		GVP Inverse		Manual Inverse	
D	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
10	91.2	3.6	35.7	1.9	105.6	5.2	277.8	7.9	178.6	9.0	1703.6	646.7
100	91.6	2.7	36.5	2.3	111.6	2.1	288.3	8.7	200.8	26.9	1732.7	672.1
1 k	89.5	3.5	36.8	3.3	119.9	2.3	308.6	7.5	256.5	34.3	1767.3	653.4
10 k	91.4	3.8	36.7	3.4	118.0	2.4	312.4	11.1	228.9	38.1	2165.0	720.8
100 k	89.7	1.9	36.7	3.3	179.9	13.5	325.8	5.3	206.2	8.5	—	—
1 M	91.3	1.5	38.5	3.0	**2961.8**	**1060.9**	351.7	6.6	252.0	33.0	—	—
10 M	101.2	5.1	43.8	2.5	—	—	343.4	13.5	216.5	12.6	—	—
100 M	100.0	2.4	45.5	2.1	—	—	372.3	13.9	271.5	34.4	—	—
1 B	105.9	4.8	52.4	7.5	—	—	—	—	—	—	—	—

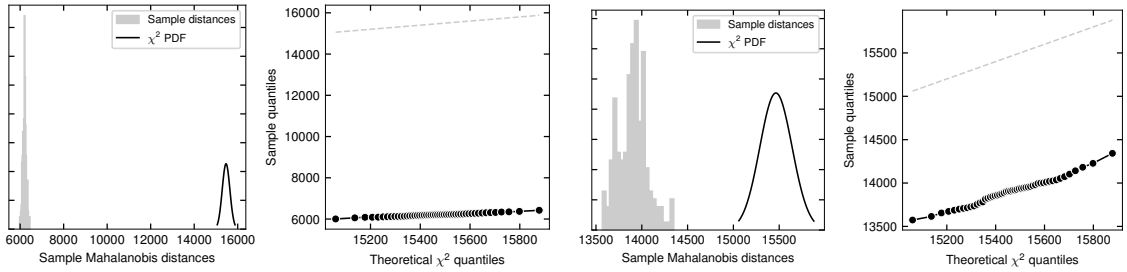


(a) Top: comparison of the performance for manual matrix-vector, Hessian-vector, and GGN-vector products. Bottom: comparison of the performance of inverse-vector products for the same methods. Implicit inverse performed via conjugate gradient for HVP and GVP. We can see that the explicit Hessian-vector product fails for problems with more than 10 000 dimensions and that the GVP is faster than the HVP.

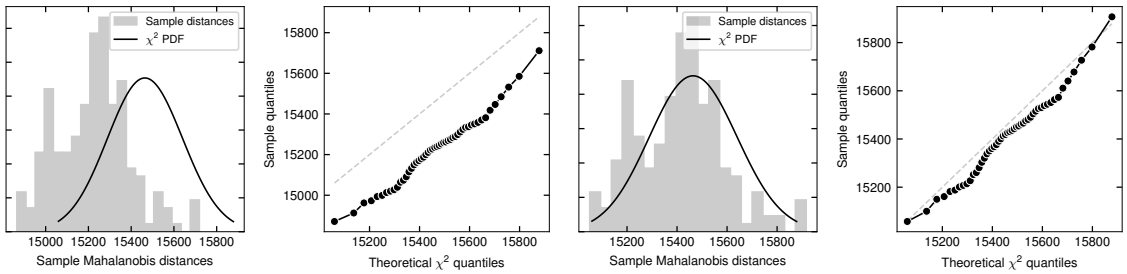


(b) Comparison of the speedup from computation of the Hessian-vector product (HVP) versus the GGN-vector product (GVP) (top) and inverse HVP versus inverse GVP using the conjugate gradient method (bottom). Speedup is calculated as the ratio of the wall-clock time of the HVP to the GVP. We can see that the two methods have the same time complexity (up to a linear factor).

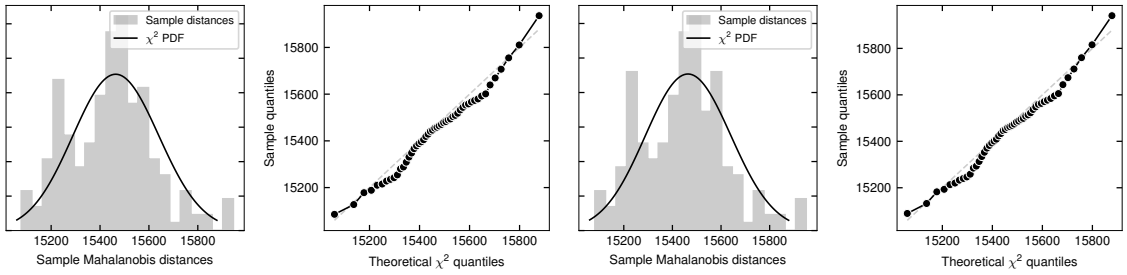
Figure 1: Performance comparison of different Hessian-vector products on CPU.



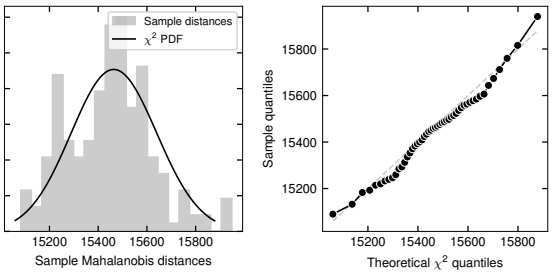
(a) $Q = 1$ **(b)** $Q = 2$



(c) $Q = 3$ **(d)** $Q = 4$

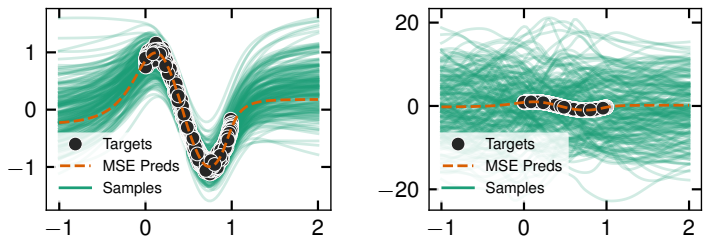


(e) $Q = 5$ **(f)** $Q = 10$



(g) $Q = 20$

Figure 2: Chi-squared sample plots for visualising normality of 200 posterior samples from the full Laplace approximation on MNIST for a varying number of quadrature points Q. Samples appear correctly distributed for $Q \geq 5$, incorrectly distributed for $Q < 4$, and borderline for $Q = 4$.



(a) $\alpha = 70$ **(b)** $\alpha = 1$

Figure 3: Laplace posterior predictive samples for a sine curve for different prior precisions α.